STATIC EXECUTION TIME ANALYSIS OF PARALLEL SYSTEMS

Andreas Gustavsson

2016
STATIC EXECUTION TIME ANALYSIS OF PARALLEL SYSTEMS

Andreas Gustavsson

Akademisk avhandling

som för avläggande av teknologie doktorsexamen i datavetenskap vid Akademin för innovation, design och teknik kommer att offentligen försvaras måndagen den 30 maj 2016, 13.15 i Beta, Mälardalens högskola, Västerås.

Fakultetsopponent: Associate Professor David Broman, KTH Royal Institute of Technology, Stockholm, Sweden
Abstract

The past trend of increasing processor throughput by increasing the clock frequency and the instruction level parallelism is no longer feasible due to extensive power consumption and heat dissipation. Therefore, the current trend in computer hardware design is to expose explicit parallelism to the software level. This is most often done using multiple, relatively slow and simple, processing cores situated on a single processor chip. The cores usually share some resources on the chip, such as some level of cache memory (which means that they also share the interconnect, e.g., a bus, to that memory and also all higher levels of memory). To fully exploit this type of parallel processor chip, programs running on it will have to be concurrent. Since multi-core processors are the new standard, even embedded real-time systems will (and some already do) incorporate this kind of processor and concurrent code.

A real-time system is any system whose correctness is dependent both on its functional and temporal behavior. For some real-time systems, a failure to meet the temporal requirements can have catastrophic consequences. Therefore, it is crucial that methods to derive safe estimations on the timing properties of parallel computer systems are developed, if at all possible.

This thesis presents a method to derive safe (lower and upper) bounds on the execution time of a given parallel system, thus showing that such methods must exist. The interface to the method is a small concurrent programming language, based on communicating and synchronizing threads, that is formally (syntactically and semantically) defined in the thesis. The method is based on abstract execution, which is itself based on abstract interpretation techniques that have been commonly used within the field of timing analysis of single-core computer systems, to derive safe timing bounds in an efficient (although, over-approximative) way. The thesis also proves the soundness of the presented method (i.e., that the estimated timing bounds are indeed safe) and evaluates a prototype implementation of it.
Abstract

The past trend of increasing processor throughput by increasing the clock frequency and the instruction level parallelism is no longer feasible due to extensive power consumption and heat dissipation. Therefore, the current trend in computer hardware design is to expose explicit parallelism to the software level. This is most often done using multiple, relatively slow and simple, processing cores situated on a single processor chip. The cores usually share some resources on the chip, such as some level of cache memory (which means that they also share the interconnect, e.g., a bus, to that memory and also all higher levels of memory). To fully exploit this type of parallel processor chip, programs running on it will have to be concurrent. Since multi-core processors are the new standard, even embedded real-time systems will (and some already do) incorporate this kind of processor and concurrent code.

A real-time system is any system whose correctness is dependent both on its functional and temporal behavior. For some real-time systems, a failure to meet the temporal requirements can have catastrophic consequences. Therefore, it is crucial that methods to derive safe estimations on the timing properties of parallel computer systems are developed, if at all possible.

This thesis presents a method to derive safe (lower and upper) bounds on the execution time of a given parallel system, thus showing that such methods must exist. The interface to the method is a small concurrent programming language, based on communicating and synchronizing threads, that is formally (syntactically and semantically) defined in the thesis. The method is based on abstract execution, which is itself based on abstract interpretation techniques that have been commonly used within the field of timing analysis of single-core computer systems, to derive safe timing bounds in an efficient (although, over-approximative) way. The thesis also proves the soundness of the presented method (i.e., that the estimated timing bounds are indeed safe) and evaluates a prototype implementation of it.
Acknowledgments

I would like to thank my advisors, Björn Lisper, Andreas Ermedahl and Jan Gustafsson, for accepting me as a doctoral student and also for their patience and invaluable guidance during my education. Without you, this thesis would not exist. A special thank you goes to Vesa Hirvisalo for putting a lot of effort and time into getting acquainted with, and suggesting improvements on, my research.

I would also like to thank all my friends and colleagues with whom I have shared many laughs and experiences during coffee breaks (I still have not learned to enjoy the taste of coffee), trips and various free-time activities. Lastly, I would like to express my deepest love and gratitude to all my friends and family who have been there for me on my journey through life and the thesis-writing process. Without your love, friendship and support, I would never have finished this thesis.

Thank you all!

Andreas Gustavsson
Idre, April, 2016

The research presented in this thesis was funded partly by the Swedish Research Council (Vetenskapsrådet) through the project “Worst-Case Execution Time Analysis of Parallel Systems” and partly by the Swedish Foundation for Strategic Research (SSF) through the project “RALF3 – Software for Embedded High Performance Architectures.”
Acknowledgments

I would like to thank my advisors, Björn Lisper, Andreas Ermedahl and Jan Gustafsson, for accepting me as a doctoral student and also for their patience and invaluable guidance during my education. Without you, this thesis would not exist. A special thank you goes to Vesa Hirvisalo for putting a lot of effort and time into getting acquainted with, and suggesting improvements on, my research.

I would also like to thank all my friends and colleagues with whom I have shared many laughs and experiences during coffee breaks (I still have not learned to enjoy the taste of coffee), trips and various free-time activities.

Lastly, I would like to express my deepest love and gratitude to all my friends and family who have been there for me on my journey through life and the thesis-writing process. Without your love, friendship and support, I would never have finished this thesis.

Thank you all!

Andreas Gustavsson
Idre, April, 2016

The research presented in this thesis was funded partly by the Swedish Research Council (Vetenskapsrådet) through the project “Worst-Case Execution Time Analysis of Parallel Systems” and partly by the Swedish Foundation for Strategic Research (SSF) through the project “RALF3 – Software for Embedded High Performance Architectures.”
## Contents

1 **Introduction**  
1.1 Real-Time Systems .......................... 1  
1.2 Execution Time Analysis ...................... 3  
1.3 Research Method ............................. 8  
1.4 Research Goal & Research Questions ............ 9  
1.5 Pilot Study ................................ 10  
1.6 Approach .................................. 11  
1.7 Contributions ............................... 14  
1.8 Included Publications ....................... 15  
1.9 Thesis Outline .............................. 16

2 **Related Work**  
2.1 Static WCET Analysis for Sequential Systems .... 19  
2.2 Static WCET Analysis for Parallel Systems ...... 20  
2.3 WCET Analysis of Parallel Systems Using Model Checking 22  
2.4 Multi-Core Analyzability ........................ 23

3 **Preliminaries**  
3.1 Partially Ordered Sets & Complete Lattices .... 26  
3.2 Constructing Complete Lattices .................. 28  
3.3 Galois Connections & Galois Insertions .......... 30  
3.4 Constructing Galois Connections ................. 34  
3.5 Constructing Galois Insertions .................. 41  
3.6 The Interval Domain ........................... 43

4 **PPL: a Concurrent Programming Language**  
4.1 States & Configurations ........................ 50
## Contents

4.2 Semantics ........................................ 52
4.3 Collecting Semantics ............................ 63

5 Abstractly Interpreting PPL .....................
5.1 Arithmetical Operators for Intervals ............. 66
5.2 Abstract Register States .......................... 66
5.3 Abstract Evaluation of Arithmetic Expressions . 69
5.4 Boolean Restriction for Intervals ................. 69
5.5 Abstract Variable States ......................... 80
5.6 Abstract Lock States ............................ 100
5.7 Abstract Configurations .......................... 104
5.8 Abstract Semantics ............................ 110

6 Safe Execution Time Analysis by Abstract Execution
6.1 Abstract Execution ............................... 163
6.2 Execution Time Analysis ......................... 187

7 Examples .............................................
7.1 Communication .................................. 193
7.2 Synchronization – Deadlock ...................... 198
7.3 Synchronization – Deadline Miss .................. 201
7.4 Data Parallel Loop .............................. 202

8 An Implementation of the Execution Time Analysis
8.1 Choosing an Implementation Language .......... 209
8.2 UPPL: a User-Friendly Version of PPL ........... 211
8.3 Generating Initial Configurations ................. 212
8.4 Implementation Architecture ..................... 215
8.5 Runtime Options ................................ 216
8.6 Verifying the Implementation .................... 221

9 Evaluation .......................................... 225
9.1 Benchmark Programs ............................. 225
9.2 Benchmark Timing Models ....................... 231
9.3 Benchmarking Setups ............................ 233
9.4 Measured Analysis Running Times ............... 235
9.5 Numbers of Derived Transitions ................. 253
9.6 Derived Execution Time Bounds .................. 277
9.7 Summary ......................................... 298
## 10 Conclusions

10.1 The Underlying Architecture ........................................... 299
10.2 Algorithmic Structure & Complexity ................................ 300
10.3 Nonterminating Transition Sequences ............................... 304
10.4 The Research Questions .................................................. 305
10.5 Other Applications of the Analysis .................................... 306
10.6 Future Work ................................................................. 307

### Bibliography

A Notation & Nomenclature .................................................. 323
B List of Assumptions ......................................................... 327
C List of Definitions .......................................................... 329
D List of Figures ............................................................... 333
E List of Tables ............................................................... 337
F List of Algorithms .......................................................... 343
G List of Lemmas .............................................................. 345
H List of Theorems ............................................................ 347

Index .................................................................................. 349
Chapter 1

Introduction

This chapter starts by introducing the fundamental concepts used within the field of the thesis. It then states the asked research questions, the approach used to answer the questions and the resulting contributions of the thesis. This chapter also presents the papers included in the thesis and a pilot study on using model checking for timing analysis of parallel real-time systems.

1.1 Real-Time Systems

As computers have become smaller, faster, cheaper and more reliable, their range of use has rapidly increased. Today, virtually every technical item, from wrist watches to airplanes, are computer-controlled. This type of computers are commonly referred to as embedded computers or embedded systems; i.e., one or more controller chips with accompanying software are embedded within the product. It has been approximated that over 99 percent of the worldwide production of computer chips are destined for embedded systems [17].

A real-time system is often an embedded system for which the timing behavior is of great importance. More formally, the Oxford Dictionary of Computing gives the following definition of a real-time system [62].

"Any system in which the time at which output is produced is significant. This is usually because the input corresponds to some movement in the physical world, and the output has to relate to that same movement. The lag from input time to output time must be sufficiently small for acceptable timeliness."
Chapter 1

Introduction

This chapter starts by introducing the fundamental concepts used within the field of the thesis. It then states the asked research questions, the approach used to answer the questions and the resulting contributions of the thesis. This chapter also presents the papers included in the thesis and a pilot study on using model checking for timing analysis of parallel real-time systems.

1.1 Real-Time Systems

As computers have become smaller, faster, cheaper and more reliable, their range of use has rapidly increased. Today, virtually every technical item, from wrist watches to airplanes, are computer-controlled. This type of computers are commonly referred to as embedded computers or embedded systems; i.e., one or more controller chips with accompanying software are embedded within the product. It has been approximated that over 99 percent of the worldwide production of computer chips are destined for embedded systems [17].

A real-time system is often an embedded system for which the timing behavior is of great importance. More formally, the Oxford Dictionary of Computing gives the following definition of a real-time system [62].

“Any system in which the time at which output is produced is significant. This is usually because the input corresponds to some movement in the physical world, and the output has to relate to that same movement. The lag from input time to output time must be sufficiently small for acceptable timeliness.”
Chapter 1. Introduction

The word “timeliness” refers to the total system and can be dependent on mechanical properties like inertia. One example is the compensation of temporary deviations in the supporting structure (e.g., a twisting frame) when firing a missile to keep the missile’s exit path constant throughout the process. Another example is to fire the airbag in a colliding car. This should not be done too soon, or the airbag will have lost too much pressure upon the human impact, and not too late, or the airbag could cause additional damage upon impact; i.e., the inertia of the human body and the retardation of the colliding car both impact on the timeliness of the airbag system. It should thus be apparent that the correctness of a real-time system depends both on the logical result of the performed computations and the time at which the result is produced.

Real-time systems can be divided into two categories: hard and soft real-time systems. Hard real-time systems are such that failure to produce the computational result within certain timing bounds could have catastrophic consequences. One example of a hard real-time system is the above-mentioned airbag system. Soft real-time systems, on the other hand, can tolerate missing these deadlines to some extent and still function properly. One example of a soft real-time system is a video displaying device. Missing to display a video frame within the given bounds will not be catastrophic, but perhaps annoying to the viewer if it occurs too often. The video will still continue to play, although with reduced displaying quality.

The ever increasing demand for performance in computer systems has historically been satisfied by increasing the speed (i.e., clock frequency) and complexity (e.g., using pipelines and caches) of the processor. It is however no longer possible to continue on this path due to the high power consumption and heat dissipation that these techniques infer. Instead, the current trend in computer hardware design is to make parallelism explicitly available to the programmer. This is often done by placing multiple processing cores on the same chip while keeping the complexity of each core relatively low. This strategy helps increasing the chip’s throughput (i.e., performance) without hitting the power wall since the individual processing cores on the multi-core chip are usually much simpler than a single core implemented on the equivalent chip area [100].

A problem with the multi-core design is that the cores typically share some resources, such as some level of on-chip cache memory. This introduces dependencies and conflicts between the cores; e.g., simultaneous accesses from two or more cores to shared resources will introduce delays for some of the cores. Processor chips of this kind of multi-core architecture are currently being used in real-time systems within, for example, the automotive industry.
To fully utilize the multi-core architecture, algorithms will have to be parallelized over multiple tasks, e.g., threads. This means that the tasks will have to share resources and communicate and synchronize with each other. There already exist techniques for explicitly parallelizing sequential code automatically. One example is the OpenMP [94] extension to the C/C++ and Fortran programming languages. The conclusion is that concurrent software running on parallel hardware is already available today and will probably be the standard way of computing in the future, also for real-time systems.

When proving the correctness of, and/or the schedulability of the tasks in, a real-time system, it is, as far as the author knows, always assumed that safe (i.e., not under-approximated) bounds on the timing behavior of all tasks in the system are known. The timing bounds are, for example, used as input to algorithms that prove or falsify the schedulability of the tasks in the system [6, 40, 79]. Therefore, it is of crucial importance that methods for deriving safe timing bounds for this type of parallel computational systems are defined.

This thesis presents a method that derives safe estimates on the timing bounds for parallel systems in which tasks share memory and can execute blocks of code in a mutually exclusive manner. The method mainly targets hard real-time systems. However, it can be applied to any computer system fitting the assumptions made in the upcoming chapters.

### 1.2 Execution Time Analysis

A program’s execution time (i.e., the amount of time it takes to execute the entire program from its entry point to its exit point) on a given processor is not constant in the general case; the execution time is dependent on the initial system state. This state includes the input to the program (i.e., the values of its arguments), the hardware state (e.g., cache memory contents) and the state of any other software that is executing on the same hardware. However, for any program and any set of initial states, at least one of the resulting execution times will be equal to the shortest possible execution time for the given program and set of initial states. The shortest possible execution time, provided that no other software is executing on the same hardware, is referred to as the Best-Case Execution Time (BCET). Likewise, at least one of the resulting execution times will be equal to the longest possible execution time for the given program and set of initial states. The longest possible execution time, provided that no other software is executing on the same hardware, is referred to as the Worst-Case Execution Time (WCET). Note that both the BCET and the WCET could
possibly be infinite,\footnote{One example for which both the BCET and WCET of a program are infinite is when the program always enters some nonterminating loop along all possible paths. Another example of an infinite WCET is when a program could deadlock.} though.

Figure 1.1 illustrates the relation between the possible execution times a program might have, and safe bounds on those execution times: any estimation of the WCET that is greater than the actual WCET is a safe bound on the actual WCET; likewise, any estimation of the BCET that is smaller than the actual BCET is a safe bound on the actual BCET. The figure also shows that measuring the execution time will always give a time between, and including, the BCET and WCET of the considered program. It is thus very difficult to guarantee that the actual BCET and WCET are found by measuring the execution time of the program. This is since a huge number of possible initial system states must typically be considered in the general case.

A trivial solution to the execution time analysis problem is to always say that the BCET of a given program is greater than or equal to 0 and that its WCET is less than or equal to infinity, which must be true for any program running on any hardware. This trivial solution should be avoided to make the analysis at all meaningful; instead a tight (i.e., not too over-approximate) estimation of the (BCET and/or) WCET of a given program should be derived. To achieve a tight execution time analysis, context-sensitive estimations of the
best-case and worst-case execution times for each single instruction, or a block of instructions, are often traditionally derived.

However, a trade-off must typically be made between the tightness and the efficiency of the analysis. When introducing multi-core architectures with shared memory (or even complex single-core architectures), the hardware does most likely suffer from timing anomalies regardless of how simple the processor cores are [2, 82, 108]. This greatly aggravates the problem of finding tight estimations of the BCET and WCET of programs executing on such hardware in an efficient way.

In *dynamic* WCET analysis, measurements of the actual execution time of the software running on the target hardware are performed. The largest measured time is typically the resulting WCET estimate. This method is often efficient but is not guaranteed to execute the program’s worst-case path, which could, for example, include some error-handling routine that is only rarely executed. Thus, the WCET might be gravely under-estimated; i.e., there might exist paths through the code with considerably worse (i.e., longer) execution times than the worst execution time detected by the measurements.

In *probabilistic* WCET analysis, extreme value theory is used to derive probabilistic estimations of the WCET of programs. The approach can be based on analyzing a mathematical model of the system, or on measuring the execution time of a given program running on an architecture incorporating randomness (one example of incorporating randomness in hardware is to use the random replacement policy for the caches in the processor). In the first case, timing probability distributions of individual operations, or even components, are mathematically determined and then used to upper bound the execution time of the program with a given certainty. In the second case, the randomness in the architecture makes the measured data independently distributed which allows the application of extreme value theory on the collected data to upper bound the execution time with a given certainty [19].

In *static* WCET analysis, the program code and the properties of the target hardware are analyzed without actually executing the program. Instead, the analysis is based on the semantics of the programming language constructs used to define the program and a (timing) model of the target hardware. Static methods usually try to find a tight estimation of the WCET, but always safely over-estimate it.

Static WCET analyses are normally split into three subtasks: the *flow analysis* (formerly known as the *high-level analysis*), which constrains the possible paths through the code; the *processor-behavior analysis* (formerly known as the *low-level analysis*), which attempts to find safe timing estimates for execu-
Chapter 1. Introduction

Figure 1.2: The three phases in traditional WCET analysis as commonly implemented in WCET analysis tools. [126]

The traditional three-phase approach assumes that the analyzed program consists of a single flow of control; i.e., is sequential. In a concurrent program, there are several flows of control (commonly referred to as threads or processes), possibly with dependencies among them. Such dependencies typically occur when the threads or processes communicate or synchronize with each other. Thus, it should be obvious that problems such as race conditions, blocking of threads accessing shared resources, and deadlocks can occur. The consequence is that the processor behavior analysis is no longer compositional, which means that the traditional three-phase approach is not directly applicable when analyzing arbitrary concurrent programs executing on parallel shared-memory architectures.

Today, there exist several algorithms and tools that strive to derive a safe and tight estimate of the WCET of a sequential task targeted for sequential hardware. Some examples of such tools are aiT [36, 126], Bound-T [57, 126], Chronos [74, 126], Heptane [126], OTAWA [9], RapiTime [107, 126], SWEET [33, 126], SymTA/P [126] and TuBound [102, 126]. aiT, Bound-T and RapiTime are commercial tools while the others are primarily research prototypes. aiT, Bound-T, Chronos, Heptane, OTAWA and TuBound are purely static tools while SWEET and SymTA/P mainly use static WCET analysis techniques, but also dynamic techniques to some extent, thus making the tools rely on hybrid analysis techniques. RapiTime is heavily based on dynamic techniques but can utilize statically derived flow information.

This thesis presents a static method that derives safe estimations of the BCET and WCET of a concurrent program consisting of dependent threads, for which race conditions, blocking of threads and deadlocks hence possibly can occur. The three traditional analysis phases are combined into one single phase; i.e., the method directly calculates the timing bound estimates while analyzing the semantic behavior of the program, based on a (safe) timing model of the underlying architecture. The definition of the timing model is out of the scope of this thesis but it is assumed to safely approximate the timing of all possible phenomena, including timing anomalies.

Note that solving the problem of finding the actual WCET in the general case is comparable to solving the halting-problem (i.e., determining whether the program will terminate), which is an undecidable problem [68]. Thus, the...
1.2 Execution Time Analysis

For the calculation phase, there exist several possible techniques for combining the information retrieved from the flow analysis and processor-behavior analysis to derive a safe estimation of the WCET. These techniques are further discussed and referenced in Section 2.1.

The traditional three-phase approach assumes that the analyzed program consists of a single flow of control; i.e., is sequential. In a concurrent program, there are several flows of control (commonly referred to as threads or processes), possibly with dependencies among them. Such dependencies typically occur when the threads or processes communicate or synchronize with each other. Thus, it should be obvious that problems such as race conditions, blocking of threads accessing shared resources, and deadlocks can occur. The consequence is that the processor behavior analysis is no longer compositional, which means that the traditional three-phase approach is not directly applicable when analyzing arbitrary concurrent programs executing on parallel shared-memory architectures.

Today, there exist several algorithms and tools that strive to derive a safe and tight estimate of the WCET of a sequential task targeted for sequential hardware. Some examples of such tools are aiT [36, 126], Bound-T [57, 126], Chronos [74, 126], Heptane [126], OTAWA [9], RapiTime [107, 126], SWEET [33, 126], SymTA/P [126] and TuBound [102, 126]. aiT, Bound-T and RapiTime are commercial tools while the others are primarily research prototypes. aiT, Bound-T, Chronos, Heptane, OTAWA and TuBound are purely static tools while SWEET and SymTA/P mainly use static WCET analysis techniques, but also dynamic techniques to some extent, thus making the tools rely on hybrid analysis techniques. RapiTime is heavily based on dynamic techniques but can utilize statically derived flow information.

This thesis presents a static method that derives safe estimations of the BCET and WCET of a concurrent program consisting of dependent threads, for which race conditions, blocking of threads and deadlocks hence possibly can occur. The three traditional analysis phases are combined into one single phase; i.e., the method directly calculates the timing bound estimates while analyzing the semantic behavior of the program, based on a (safe) timing model of the underlying architecture. The definition of the timing model is out of the scope of this thesis but it is assumed to safely approximate the timing of all possible phenomena, including timing anomalies.

Note that solving the problem of finding the actual WCET in the general case is comparable to solving the halting-problem (i.e., determining whether the program will terminate), which is an undecidable problem [68]. Thus, the
space of possible system states that a WCET analysis must search through could be extremely large, or even infinite, in the general case. This means that the analysis itself might not terminate in the general case. Therefore, techniques to increase the probability of, or even more desirable, guarantee, analysis termination should be derived. For many of the traditional methods using abstract interpretation for analyzing sequential programs, there are ways to guarantee termination using widening techniques (which are often used in conjunction with narrowing techniques to increase the precision of the analysis) [91]. These techniques are not directly applicable to the method presented in this thesis, though. Therefore, other more primitive, timeout-based techniques are suggested and used.

1.3 Research Method

An overview of the overall used research method is depicted in Figure 1.3. The first performed activity includes a study of the state-of-the-art literature (such as books and journal and conference publications) within the given research area. It also includes attending courses, workshops, conferences, summer schools and seminars related to the topic. This is done to get an idea about what the exact problem is, what others have already done to solve the problem and what is missing or not so promising about the other approaches. The most relevant related research is presented in Chapter 2.

When the exact problem to solve is identified, a research goal is defined. This goal is then refined into a set of research questions. The identified research goal and derived research questions are presented in Section 1.4. Since the research presented in this thesis is novel, the research goal is focused toward the research settings “feasibility”, “characterization” and “method/means” [116]. This is to allow for a feasible result to be derived within a reasonable amount of time.

A pilot study, as further discussed in Section 1.5, is the result of the initially proposed overall solution. An iterative approach [29] is then used based on an evaluation of an implementation of that solution to finally derive the solution presented in this thesis (hence the cycles defined in the graph shown in Figure 1.3), which is introduced in Section 1.6. While deriving solution propositions, the state-of-the-art literature is further studied to find guidelines and inspiration.

When implementing and validating the parts (i.e., algorithms etc.) of the solutions presented in this thesis, a deductive approach [29] is naturally used.
1.4 Research Goal & Research Questions

Within the different parts, iterative, (mathematically) inductive and recursive approaches [29] are used. This is apparent in the presented results, which are summarized in Section 1.7, and in the structure of the upcoming chapters.

The research performed in the fourth, fifth and sixth steps to answer the guiding research questions from step three resulted in one or more research publications. The relevant publications are presented in Section 1.8.

1.4 Research Goal & Research Questions

The main goal of this thesis is the following.

_To show that the execution times of concurrent programs consisting of communicating and synchronizing software threads, executing on parallel architectures providing shared memory and primitives for mutually exclusive execution, can be safely bounded in a non-trivial way with tightness in mind._

To reach this goal, the aim is to develop, implement and evaluate a method for analyzing the above mentioned system. The hypothesis is that it is indeed possible to achieve the goal. The result is however not expected to be optimal.
in any sense, but should give hints on how to proceed and improve the derived method.

The following research questions are derived in order to find a suitable path for reaching the goal. The overall question to be answered is Question 1. The other questions concern specific problems arising when analyzing concurrent programs consisting of dependent tasks.

**Question 1:** “Can safe and tight bounds on the execution time of concurrent programs consisting of dependent tasks be derived?”

**Question 2:** “How can the timing of communicating tasks be safely and tightly estimated?”

**Question 3:** “How can the timing of synchronizing tasks be safely and tightly estimated?”

**Question 4:** “How can programs suffering from deadlocks and other types of nonterminating programs be handled?”

### 1.5 Pilot Study

Model checking is a technique for verifying properties of a model of some system. The idea of using model checking to perform WCET analysis has been investigated and shown to be adequate for analyzing parts of a single-core system [28, 60, 88].

Timed automata\(^2\) can be used to model real-time systems [4]. An automaton can be viewed as a state machine with locations and edges [66]. A state represents certain values of the variables in the system and which location of an automaton that is active, while the edges represent the possible transitions from one state to another [66]. (Continuous) time is expressed as a set of real-valued variables called clocks. UPPAAL\(^3\) [10, 72, 124] is a tool used to model, simulate and verify properties of networks of timed automata [10, 11, 66].

An example application of using timed automata to model and verify real-time systems is given by the TIMES tool [5]. This tool is based on timed automata theory and the verifier component of UPPAAL and is mainly used to verify the schedulability of a modeled system of software tasks.

---

\(^2\)Other literature present the formal syntax and semantics of timed automata [3, 66].

\(^3\)Other literature present an introduction to UPPAAL [10] and the formal semantics of networks of timed automata [66].
The approach presented in this thesis is based on abstract interpretation \[27, 41, 91\] which is a method for safely approximating the semantics of a program and can be used to obtain a set of possible abstract states for each point in the program. An abstract entity collects, and most often over-approximates, the information given by a set of concrete entities. An entity could for example be the value of a register, which in the abstract domain often is referred to as an abstract value; a collection of such information (e.g., a mapping from register names to their corresponding values), which is often referred to as a state; or even a transition between states. By collecting the information given by a set of concrete entities into a single abstract entity, an analysis based on the abstract entities (i.e., an analysis based on abstractly interpreting the semantics of a program) can become less complex and more efficient, but might suffer from imprecision, compared to an analysis based on the concrete entities. Note that, in general, some form of abstraction of the concrete semantics has to be done since the analysis otherwise will become too complex due to the enormous number of entities/states that must otherwise be handled.

The concrete semantics of an arbitrary programming language can be abstracted in many different ways. The choice of abstraction is done by defining an abstract domain. An abstract domain is essentially the set of all possible abstract states that fit the definition of the domain. A provably safe abstraction is often achieved by establishing a Galois connection between a concrete domain, \( C \), and an abstract domain, \( A \), as depicted in Figure 1.4. A Galois connection
is basically a pair of two functions: the abstraction function, \( \alpha \), and the concretization function, \( \gamma \). The essence of Galois connections is that an abstraction of a concrete entity always safely approximates the information given by the concrete entity: if an abstraction of a concrete entity within the concrete domain is performed, followed by a concretization of the resulting abstract entity, then the resulting concrete entity will contain at least the information given by the original concrete entity. The details and properties of Galois connections are presented in Section 3.3.

The semantics of a program is basically a set of equations based on concrete states. A solution to these equations can be found by iterating on transitions between states until the least fixed point is found; this solution is often referred to as the collecting semantics of the program. Given a safe abstraction of the program semantics, the equations can always be defined and solved in the abstract domain. The resulting abstract solution is a safe approximation of the concrete solution (i.e., of the concrete collecting semantics).

An example of an abstract domain is \( \text{Intv} \), defined as \{ \([z_1, z_2] \mid -\infty \leq z_1 \leq z_2 \leq \infty \land z_1, z_2 \in \mathbb{Z} \cup \{-\infty, \infty\}\}; i.e., the set of all integer intervals that “fit inside” \([-\infty, \infty]\). This domain can be used to over-approximate the concrete domain \( \mathcal{P}(\{z \in \mathbb{Z} \cup \{-\infty, \infty\} \mid -\infty \leq z \leq \infty\}) = \mathcal{P}(\mathbb{Z} \cup \{-\infty, \infty\}); i.e., the set of all possible sets of integers between (and including) \(-\infty\) and \(\infty\). In other words, a set of integers can be approximated using an interval. Note that \( \text{Intv} \) is completely defined, and that a Galois connection is established between \( \text{Intv} \) and the concrete domain mentioned above, in Section 3.6.

Assume that the program variable \( x \) can have the value \( v \), such that \( v \in \{1, 2, 5, 8\} \), in a given point of the program according to the concrete semantics (i.e., \( x \) has four possible values in the given program point). In the abstract

![Galois connection](image-url)
domain, the value of \( x \) could safely be represented by \([1, 8]\). This is an over-
approximation since turning the abstract value into a set of concrete values yields \([1, 8] \rightarrow \{1, 2, 3, 4, 5, 6, 7, 8\} \supseteq \{1, 2, 5, 8\}\). It can be noted that \([1, 8]\) is
the best (tightest) approximation of the values of \( x \), since \([1, 8]\) is the smallest
interval containing all the possible concrete values of \( x \).

Abstract execution (AE) [41, 46] was originally designed as a method to
derive program flow constraints [126] on imperative sequential programs, like
bounds on the number of iterations in loops and infeasible program path con-
straints. This information can be used by a subsequent execution time (WCET)
analysis [126] to compute a safe WCET bound. AE is based on abstract in-
terpretation, and is basically a very context-sensitive value analysis [91, 126]
which can be seen as a form of symbolic execution [41] (i.e., sets of possible
abstract values for the program variables etc. in the visited program points are
found). Note that AE is in fact a technique for iterating on semantic transitions
until a fixed point is found; i.e., a technique based on fixed point iteration. AE
is very context-sensitive because the possible states at a specific program point
considered in different iterations of the analysis do not necessarily have any
obvious correlation to each other (e.g., the derived states of a given program
point are not necessarily joined before used in future iterations). The program
is hence executed in the abstract domain; i.e., abstract versions of the program
operators are executed and the program variables have abstract values, which
thus correspond to sets of concrete values.

The main difference between AE and a traditional value analysis is that
in the former, an abstract state is not necessarily calculated for each program
point. Instead, the abstract state is propagated on transitions in a way similar
to the concrete state for concrete executions of the program. Note that since
values are abstracted, a state can propagate to several new states on a single
transition, e.g., when both branches of a conditional statement could be taken
given the abstract values of the program variables in the current abstract state.
Therefore, a worklist algorithm that collects all possible transitions is needed
to safely approximate all concrete executions.

There is a risk that AE does not terminate. However, if it terminates then
all final states of the concrete executions have been safely approximated [41].
Nontermination can be dealt with by setting a “timeout,” e.g., as an upper limit
on the number of abstract transitions.

If timing bounds on the statements of the program are known, then AE is
easily extended to calculate BCET and WCET bounds by treating time as a
regular program variable that is updated on each state transition – as with all
other variables, its set of possible final values is then safely approximated when
the algorithm terminates [34].

The approach used in this thesis is to statically calculate safe BCET and WCET estimations by abstractly executing the analyzed program using a safe timing model of the underlying architecture. To use this approach, some suitable concurrent programming language, preferably with a formally defined semantics, must be considered and its semantics must be abstracted. There are many different language alternatives that could be considered for use in real-time systems. Some examples of concurrent programming languages are Ada [65], Cilk [16], ERLANG [8], and Scala [93]. Other examples (of extensions to sequential programming languages) are the OpenMP [94] API for C/C++ and Fortran, and the POSIX thread library standard [18, 61] for C/C++.

However, there is already an extensive amount of research on how to handle features of a programming language that are not strictly concurrency-related, such as functions and pointer variables [91]. Including those features in the language only adds to the complexity and difficulty of performing the abstraction of the language semantics.

In this thesis, focus is put on analyzing features of concurrent programming languages that have not been thoroughly investigated before within WCET-related research: communication and synchronization between the concurrent entities of execution (referred to as threads). Therefore, a prototype concurrent programming language based on threads and with a formally defined semantics is presented. The language does not include features such as functions or pointer variables but focuses on allowing communication using shared memory and synchronization using mutually exclusive locks.

Basically, the only assumption made on the underlying architecture is that it provides (or can simulate) a shared memory address space, that can be used for communication, and mutually exclusive shared resources, that can be used for synchronization. One example of such an architecture is a multi-core CPU. Another example is a virtualization environment that runs on top of a distributed system and provides a shared memory view. Yet another example is any real-time operating system; e.g., VxWorks [128].

1.7 Contributions

The main contributions of this thesis are the following.

1. PPL: a formally defined, rudimentary, concurrent programming language for real-time systems, including shared memory and synchroniza-
1.8 Included Publications

This thesis is based on the material presented in the following papers. Andreas Gustavsson is the main author of all the listed publications and has alone contributed with all the technical material presented in them.

Paper A

Worst-Case Execution Time Analysis of Parallel Systems
Andreas Gustavsson.
Presented at the RTiS workshop, 2011 [47].

This paper addresses contribution 1 and presents the first definition of PPL and a very simple (non-generalized) timing model.

Paper B

Toward Static Timing Analysis of Parallel Software
Andreas Gustavsson, Jan Gustafsson and Björn Lisper.
Presented at the WCET workshop, 2012 [50].
This paper addresses contributions 2 and 3 and presents a work-in-progress timing analysis that can analyze all aspects of PPL, except synchronization. The presented analysis uses abstract execution to derive safe estimations of the BCET and WCET of the analyzed program.

**Paper C**

*Toward Static Timing Analysis of Parallel Software - Technical Report*
Andreas Gustavsson, Jan Gustafsson and Björn Lisper.
Technical report, 2012 [51].

This paper addresses contributions 2 and 3 and is an extended version of paper B. The paper includes all the mathematical details and a sketch for the correctness/soundness proof.

**Paper D**

*Timing Analysis of Parallel Software Using Abstract Execution*
Andreas Gustavsson, Jan Gustafsson and Björn Lisper.
Presented at the VMCAI conference, 2014 [52].

This paper addresses contributions 1, 2 and 3 and summarizes the theoretical work presented in this thesis. It presents a timing analysis that is based on the analysis defined in Papers B and C. The presented analysis derives safe estimations of the BCET and WCET for any program defined using a slightly modified version of PPL as presented in paper A, given a (safe) timing model of the underlying architecture.

**Paper E**

*Towards WCET analysis of multicore architectures using UPPAAL*
Andreas Gustavsson, Andreas Ermedahl, Björn Lisper and Paul Pettersson.
Presented at the WCET workshop, 2010 [49].

This paper does not address any of the main contributions of this thesis. However, this paper contains the pilot study discussed in Section 1.5.

### 1.9 Thesis Outline

The rest of this thesis is organized as follows.
Chapter 2 presents some research that is closely related to the material presented in this thesis. It also presents a brief introduction to the strategies traditionally used in WCET analysis.

Chapter 3 introduces the reader to the fundamental concepts and theories needed to understand the contents of the following chapters.

Chapter 4 formally defines PPL, a concurrent programming language based on shared memory and primitives for mutually exclusive execution.

Chapter 5 presents an abstraction of the PPL semantics. Note that the abstraction is not safe for arbitrary PPL programs and that special care must be taken if using it (cf. Chapter 6).

Chapter 6 defines a safe timing analysis using abstract execution based on the abstraction made in Chapter 5.

Chapter 7 presents some examples that show how the analysis presented in Chapter 6 handles communication and synchronization in PPL programs.

Chapter 8 presents an implementation of the analysis presented in the previous chapters.

Chapter 9 evaluates the analysis based on the implementation of the analysis as discussed in Chapter 8.

Chapter 10 discusses the research questions and the method presented in this thesis. The chapter also gives pointers to future work.

For the reader’s convenience, the following appendices are provided.

Appendix A summarizes the notation and nomenclature used in this thesis.

Appendices B-H respectively present listings of the assumptions, definitions, figures, tables, algorithms, lemmas and theorems defined in this thesis.
Related Work

WCET-related research started with the introduction of timing schemas by Shaw in 1989 [115]. Shaw presents rules to collapse the CFG (Control Flow Graph) of a program until a final single value represents the WCET. This chapter presents some research related to this thesis and also to the traditional three-phase WCET analysis. Excellent overviews of the WCET research from the years 2000 [103] and 2008 [126] have been presented.

2.1 Static WCET Analysis for Sequential Systems

In this thesis, an approach for static analysis of the timing behavior of arbitrary concurrent programs based on threads, shared memory and synchronization on locks, as given by a small concurrent programming language, is presented. The field of static WCET analysis has, just until recently, mainly been focusing on sequential programs executing on single-processor systems. This is the kind of research referenced in this section and on which the method presented in this thesis is based.

In the field of processor-behavior (low-level) analysis, most research efforts have been dedicated to analyzing the effects of different hardware features, including pipelines [31,55, 77, 117, 123], caches [75, 77, 123, 125], branch predictors [25], and super-scalar CPUs [76, 112].

Within flow (high-level) analysis, most research has been dedicated to loop bound analysis. Flow analysis can also identify infeasible paths, i.e., paths which are executable according to the program control flow graph structure,
Chapter 2

Related Work

WCET-related research started with the introduction of timing schemas by Shaw in 1989 [115]. Shaw presents rules to collapse the CFG (Control Flow Graph) of a program until a final single value represents the WCET. This chapter presents some research related to this thesis and also to the traditional three-phase WCET analysis. Excellent overviews of the WCET research from the years 2000 [103] and 2008 [126] have been presented.

2.1 Static WCET Analysis for Sequential Systems

In this thesis, an approach for static analysis of the timing behavior of arbitrary concurrent programs based on threads, shared memory and synchronization on locks, as given by a small concurrent programming language, is presented. The field of static WCET analysis has, just until recently, mainly been focusing on sequential programs executing on single-processor systems. This is the kind of research referenced in this section and on which the method presented in this thesis is based.

In the field of processor-behavior (low-level) analysis, most research efforts have been dedicated to analyzing the effects of different hardware features, including pipelines [31, 55, 77, 117, 123], caches [75, 77, 123, 125], branch predictors [25], and super-scalar CPUs [76, 112].

Within flow (high-level) analysis, most research has been dedicated to loop bound analysis. Flow analysis can also identify infeasible paths, i.e., paths which are executable according to the program control flow graph structure,
but not feasible when considering the semantics of the program and the possible input data values. There are numerous approaches to flow analysis, such as using abstract interpretation, symbolic execution, abstract execution, Presburger arithmetics, specialized data flow analyses, and syntactical analysis of parse trees [46, 56, 57, 81, 123].

Three main methods exist for the WCET calculation: the tree-based method [24, 25, 77], originating from Park’s timing schemas [98]; the path-based method [55, 118]; and the Implicit Path Enumeration Technique (IPET) [33, 57, 75, 104, 123], where the WCET calculation problem is formulated as an Integer Linear Programming (ILP) problem, and the set of execution paths is restricted by linear constraints.

An alternative way of computing the ILP problem is by using a graph-based approach [104]. A comparison of the graph-based and IPET approaches has also been presented [60]. The graph-based approach is conducted using model checking in UPPAAL [10, 72, 124]. It is shown that IPET outperforms the model checking-based approach, but that model checking allows calculating tight WCET bounds and easy integration of complex hardware models. A combined approach is proposed, where model checking is used to analyze local regions of the code, while IPET is used to solve the global analysis. Other motivations to why model checking could be useful in WCET analysis have also been presented [88].

For analyses based on abstract execution, it is possible to calculate the BCET and WCET estimates of sequential programs during the abstract execution, without first generating flow facts [34, 46]. This thesis uses basically the same approach, but applies it to explicitly concurrent programs.

2.2 Static WCET Analysis for Parallel Systems

In this thesis, focus is put on statically analyzing concurrent programs with synchronizing and communicating threads executing on some arbitrary (i.e., sequential or parallel) architecture without any restrictions to thread migration. All hardware aspects are assumed to be covered by the model of the underlying architecture, which is, however, not the focus of this thesis.

Some other research has been conducted within the field of static WCET analysis for multi-core and other types of multi-processor systems. This research is in most cases very young and, in some cases, the level of evaluation is lacking depth. However, the related methods presented in this section are not proven to produce safe results, as far as the author knows. This is one of the
main differences compared to the method presented in this thesis.

Note that not all of the material referred to here focus on analysis of concurrent programs and are not as general as the method presented in this thesis. In some cases, the system is assumed to consist of sequential programs (i.e., independent processes) executing on individual cores of a shared memory multi-core processor. A clear distinction of whether the material focuses on concurrent or sequential programs will be made. Also note that some of the material mainly focuses on how different hardware aspects, such as caches that are shared between processors, affect the timing behavior of programs. This is not the focus of this thesis.

A static analysis method for analyzing concurrent programs executing on a multi-core processor with a shared L2 instruction cache has been presented [130]. A limitation of this analysis is that the L1 data cache is assumed to be perfect (i.e., all accesses are assumed to be hits, which is generally not the case) and thus does not affect the contents of the L2 cache. Based on this work, the same authors also address the same problem for the case that the shared L2 cache is direct-mapped [131].

There is also an approach for analyzing sequential programs executing on separate cores of a multi-core processor with a shared L2 instruction cache (also assuming a perfect L1 data cache) that takes effects from timing anomaly influenced pipelines into account [20].

Staschulat et al. [119] consider an integrated task- and system-level analysis to estimate memory access times for sequential programs running in parallel with programs executing on other processors. Their approach requires full information about all tasks running in the system, and it makes quite strong assumptions about the task model.

Mittermayr and Blieberger [89] use a graph based approach and Kronecker algebra to calculate an estimation of the WCET of concurrent programs. The graph is referred to as CPG (Concurrent Program Graph) and plays a role similar to the CFG for sequential programs.

Ozaktas et al. [95] focus on analyzing synchronization delays experienced by concurrent POSIX threads executing on time-predictable shared-memory multi-core architectures.

Potop-Butucaru and Puaut [101] target static timing analysis of concurrent programs executing on a parallel processor where “channels” are used to communicate between, and synchronize, the parallel tasks. Additional edges representing such communication and synchronization are then used to connect the CFGs of the individual tasks. The goal is to enable the use of the traditional three-phase WCET analysis when analyzing parallel systems.
There is also some research approaches targeting, for example, deadlock detection [30], data race freeness and other types of security properties [54] and data flow analysis for concurrent programs [39, 69, 73], which do not consider timing properties of the analyzed programs. However, these methods could be incorporated to increase the tightness of WCET analyses. Another example is the Mthread plugin [1], which can be used by the Framework for Modular Analysis of C programs, Frama-C, to perform a safe value analysis of concurrent C programs using abstract interpretation. A survey of analyses for concurrent programs, with guidelines for how to best analyze different types of such programs, was presented in 2001 [109].

2.3 WCET Analysis of Parallel Systems Using Model Checking

Model checking is a very interesting technique that has been investigated as an alternative for performing WCET analysis. The research approaches presented in this section focus on analyzing different aspects of analyzing multi-processor systems and are mostly related to the pilot study presented in Section 1.5. Note that not all approaches tackle concurrent software, though. A clear distinction will be made for these cases.

Lv et al. [83] and Wu and Zhang [129] use model checking of timed automata to perform WCET analysis. In this approach, a timed automata-model of the system to be analyzed is created. Then, specific properties of the model are verified to find a WCET estimate for the analyzed system. The achievable tightness of the WCET estimate depends on the level of details in the timed automata-model. Both papers mainly propose methods for reducing the size of the state space by altering the program model without affecting the true WCET of the model. This is a very important aspect when using model checking in general. If the model is too large and complex, the state space will “explode,” which means that the number of possible states is very large and analyzing the model becomes infeasible.

Lv et al. [84] have also combined abstract interpretation with model checking to avoid the scalability problems found in, for example, the pilot study presented in Section 1.5 of this thesis [49]. This work does not focus on explicitly concurrent software, though.

In an attempt to overcome the inherent and general problem of a huge state space size when considering model checking approaches, symbolic model checking was introduced in 1992 [86]. This approach is similar to abstract ex-
execution in that all derived states are not saved. In symbolic model checking, sets of states are represented using boolean functions. This approach basically corresponds to using a similar abstract domain when performing abstract execution. To further lower the state space complexity, bounded model checking was introduced in 1999 [13, 14]. This approach reduces model checking to a propositional satisfiability problem and has been extensively used to find hardware bugs but similar approaches have also been used for very accurate WCET analysis [15] and C program verification [21].

Extensions of the bounded model checking approach for verifying the absence of bugs, such as deadlocks and data races, in concurrent software have been presented [37, 105, 106]. These approaches either bound the number of allowed context (i.e., thread) switches and focus on single-core hardware or assume a predictable sequential behavior of the program, which is clearly a drawback when analyzing arbitrary parallel systems. A common property of these approaches is to put focus on verifying the absence of data races etc. based on the functional behavior of the analyzed program, not on analyzing the timing behavior of it.

A technique closely related to bounded model checking is symbolic execution. Here, constraints for how the different statements in the program affect the system state are derived by symbolically executing each statement. Then properties for the different possible paths through a program can be derived by solving these constraints. Luckow combined model checking of timed automata and symbolic execution to derive bounds on the WCET for Java real-time systems [80].

## 2.4 Multi-Core Analyzability

Some other research addresses the problem of low predictability in multi-core processors. This work mostly gives multi-core design guidelines and suggestions on how to use additional or modified hardware to increase the predictability, and thus, the analyzability. This research could potentially drastically lower the complexity of the timing model later discussed in this thesis.

In an extension [53] to the previously discussed static WCET analysis method for parallel systems [130], memory bits for each instruction are used to determine whether the instruction should be cached or not. For example, to avoid pollution of the shared cache, “Static Single Usage” instructions (i.e., instructions in the program that are only referenced/ executed once) should not be cached. This generates the possibility to determine a tighter WCET estimate.
Special arbiters (hardware circuits) can be added to a shared memory multicore processor to synchronize the memory accesses from different cores in order to increase the timing predictability of the system [96]. The result is a multi-core architecture that can be analyzed with existing single-core (and single-task) WCET analysis tools.

GAMC [97] is an SDRAM controller which upper bounds the delay a core can suffer from memory-access interferences from other cores. This is an important approach since the largest memory access latency will occur when accessing the main memory. The result is tight WCET approximations which only differ a few percent from the largest measured execution times, for a specific analyzed program suite.

Time Division Multiple Access (TDMA)-based memory bus access policies can also be introduced to make all memory accesses predictable, regarding the WCET [6, 111]. The problem with this approach is that the performance of the processor will be seriously degraded since, in the average case, a memory access from any core will be stalled for half the TDMA period (and the whole period in the worst case).

Kelter et al. [67] suggest to use the Priority Division (PD) protocol instead of the TDMA protocol. They show that PD is a very promising replacement for TDMA that provides predictability while not degrading the performance as severely as TDMA.

The MERASA project [87, 110] strives towards providing a timing analyzable multi-core CPU with a system level software (cf. operating system). A case study [110] has been performed, in which an estimation of the WCET of a parallel 3D multi-grid solver, executing on the MERASA multi-core platform, is derived. The parMERASA project [99] is a continuation of the MERASA project.

The PROARTIS project [19] is basically a continuation of the parMERASA project in which focus is put on timing analyzable hardware, specially adapted for probabilistic WCET analysis. One example of such hardware is a first level cache with random placement and replacement policies which has been implemented and evaluated. Random replacement policies will also be developed for second level caches and translation look-aside buffers [70].
Chapter 3

Preliminaries

In general, basing a timing analysis on the concrete semantics of a program is infeasible due to the enormous, or even infinite, number of states that must be explored. As discussed in Section 1.6, abstract interpretation [27, 41, 91] is a method for safely approximating the concrete program semantics and can be used to obtain a set of possible abstract states for each point in a program. An abstract state collects, and most often over-approximates, the information given by a set of concrete semantic states. This means that an analysis based on abstractly interpreting the semantics of a program can become less complex and more efficient, but perhaps also less precise, compared to an analysis based on the concrete semantics (which cannot even be considered a feasible option). The analysis presented in this thesis is based on abstract interpretation. Therefore, this chapter introduces the foundations used by abstract interpretation techniques.

Some of the presented lemmas and theorems, with their accompanying proofs, are originally defined elsewhere [41, 91]. For these lemmas and theorems, proper references are given in the title of their proofs. However, all proofs are presented here for completeness and for their instructiveness.

NOTE. To increase the readability of this thesis, the maximum operator, max, is from here on assumed to have the same definition as the supremum operator, sup. Likewise, the minimum operator, min, is from here on assumed to have the same definition as the infimum operator, inf.

This chapter can probably be skipped by readers already very familiar with
complete lattices and Galois connections. However, some new and instructive material is introduced so it might be worth skipping through the standard parts and focus on the new and unknown parts.

**Note.** A summary of the notation and nomenclature used in this thesis can be found in Appendix A.

### 3.1 Partially Ordered Sets & Complete Lattices

The *relation*, as described by $\mathcal{R}: A \times B \rightarrow \{\text{true}, \text{false}\}$ where $A \times B$ is the Cartesian product of the two sets $A$ and $B$, between two elements $a \in A$ and $b \in B$ is denoted by $a \mathcal{R} b$. Given that for every $a \in A$, there is at most one element, $b \in B$, such that $a \mathcal{R} b$, then $\mathcal{R}$ is said to be a partial function from $A$ to $B$. Given that for every $a \in A$, there is exactly one element, $b \in B$, such that $a \mathcal{R} b$, then $\mathcal{R}$ is said to be a total function from $A$ to $B$.

A partial ordering is a relation $\sqsubseteq: A \times A \rightarrow \{\text{true}, \text{false}\}$ that is reflexive (i.e., $\forall a \in A : a \sqsubseteq a$), transitive (i.e., $\forall a, a', a'' \in A : ((a \sqsubseteq a' \land a' \sqsubseteq a'') \Rightarrow a \sqsubseteq a''$) and anti-symmetric (i.e., $\forall a, a' \in A : ((a \sqsubseteq a' \land a' \sqsubseteq a) \Rightarrow a = a')$). The pair $(A, \sqsubseteq)$ is a partially ordered set if $\mathcal{R}: A \times A \rightarrow \{\text{true}, \text{false}\}$ is a partial ordering on $A$.

A subset $A'$ of $A$ has $a \in A$ as an upper bound if $\forall a' \in A' : a' \sqsubseteq a$ and as a lower bound if $\forall a' \in A' : a \sqsubseteq a'$. The element $a \in A$ is the least upper bound of $A'$ if $a$ is an upper bound of $A'$ and for all other upper bounds, $a' \in A$, of $A'$, $a \sqsubseteq a'$ (cf. Definition 3.28). The element $a \in A$ is the greatest lower bound of $A'$ if $a$ is a lower bound of $A'$ and for all other lower bounds, $a' \in A$, of $A'$, $a' \sqsubseteq a$ (cf. Definition 3.27). Note that a greatest lower bound and/or a least upper bound might not exist for all subsets of a partially ordered set. When they do exist, they are unique (since $\sqsubseteq$ is anti-symmetric) and will be denoted $\bigcap A'$ and $\bigcup A'$, respectively. The shorthand $a \sqcap a'$ will be used to denote $\bigcap \{a, a'\}$. Likewise, $a \sqcup a'$ will be used to denote $\bigcup \{a, a'\}$.

A complete lattice, $V = \langle V, \sqsubseteq, \sqcup, \sqcap, \bot, \top \rangle$, is a partially ordered set, $(V, \sqsubseteq)$, such that all subsets have greatest lower bounds and least upper bounds. The least element of $V$ is denoted $\bot$ (the *bottom element*) and is defined as $\bot = \bigcup \emptyset = \bigcap V$. The greatest element of $V$ is denoted $\top$ (the *top element*) and is defined as $\top = \bigcup V = \bigcap \emptyset$.

---

1 Extensive introductions to complete lattices can be found in other literature [91].
The properties of monotone, completely additive and completely multiplicative functions are given in Definitions 3.1, 3.2 and 3.3, respectively. Note that when $V_1$ and $V_2$ are complete lattices, all subsets of these sets have least upper bounds and greatest lower bounds. Lemma 3.4 states some specific properties of a completely multiplicative function.

**Definition 3.1 (Monotone function):**
A function, $f : V_1 \rightarrow V_2$, between the partially ordered sets $V_1 = (V_1, \sqsubseteq_1)$ and $V_2 = (V_2, \sqsubseteq_2)$ is monotone if:

$$\forall v_1, v'_1 \in V_1 : v_1 \sqsubseteq_1 v'_1 \Rightarrow f(v_1) \sqsubseteq_2 f(v'_1)$$

**Definition 3.2 (Completely additive function):**
A function, $f : V_1 \rightarrow V_2$, between the partially ordered sets $V_1 = (V_1, \sqsubseteq_1)$ and $V_2 = (V_2, \sqsubseteq_2)$ is completely additive if for all $V'_1 \subseteq V_1$

$$f(\bigcup_1 V'_1) = \bigcup_2 \{f(v) | v \in V'_1\}$$

develop whenever $\bigcup_1 V'_1$ and $\bigcup_2 \{f(v) | v \in V'_1\}$ exist.

**Definition 3.3 (Completely multiplicative function):**
A function, $f : V_1 \rightarrow V_2$, between the partially ordered sets $V_1 = (V_1, \sqsubseteq_1)$ and $V_2 = (V_2, \sqsubseteq_2)$ is completely multiplicative if for all $V'_1 \subseteq V_1$

$$f(\bigcap_1 V'_1) = \bigcap_2 \{f(v) | v \in V'_1\}$$

develop whenever $\bigcap_1 V'_1$ and $\bigcap_2 \{f(v) | v \in V'_1\}$ exist.

**Lemma 3.4 (Completely multiplicative functions):**
If $V = \langle V, \subseteq, \cup, \cap, \bot, \top \rangle$ and $\tilde{V} = \langle \tilde{V}, \subseteq, \tilde{\cup}, \tilde{\cap}, \tilde{\bot}, \tilde{\top} \rangle$ are complete lattices and $\tilde{V}$ is finite, then the three conditions

1. $\gamma : \tilde{V} \rightarrow V$ is monotone,
2. $\gamma(\tilde{\top}) = \top$, and
3. $\gamma(\tilde{v} \cap \tilde{v}') = \gamma(\tilde{v}) \cap \gamma(\tilde{v}')$, whenever $\tilde{v} \tilde{\subseteq} \tilde{v}' \land \tilde{v}' \tilde{\subseteq} \tilde{v}$, where $\tilde{v}, \tilde{v}' \in \tilde{V}$

are jointly equivalent to $\gamma : \tilde{V} \rightarrow V$ being completely multiplicative.

**Proof ([91]):** Assume that $V = \langle V, \subseteq, \cup, \cap, \bot, \top \rangle$ and $\tilde{V} = \langle \tilde{V}, \subseteq, \tilde{\cup}, \tilde{\cap}, \tilde{\bot}, \tilde{\top} \rangle$ are complete lattices and that $\tilde{V}$ is finite.
First note that if \( \gamma : \hat{V} \to V \) is completely multiplicative, then the three conditions trivially hold. Next, assuming that the three conditions are fulfilled, it will be proven that
\[
\gamma(\bigcap \hat{V}') = \bigcap \{ \gamma(\bar{v}) \mid \bar{v} \in \hat{V}' \}
\]
where \( \hat{V}' \subseteq \hat{V} \), using induction on the finite cardinality of \( \hat{V}' \subseteq \hat{V} \).

If the cardinality of \( \hat{V}' \) is 0, then \( \gamma(\bigcap \hat{V}') = \bigcap \{ \gamma(\bar{v}) \mid \bar{v} \in \hat{V}' \} \) follows from condition 2. This proves the base case of the induction.

If the cardinality of \( \hat{V}' \) is larger than 0, then \( \hat{V}' = \hat{V}'' \cup \{ \bar{v}' \} \) where \( \bar{v}'' \not\in \hat{V}'' \); which ensures that the cardinality of \( \hat{V}'' \) is strictly less than that of \( \hat{V}' \). Note that by condition 1, \( \gamma(\bar{v}'' \cup \bar{v}') = \gamma(\bar{v}) \cap \gamma(\bar{v}') \) also when \( \bar{v} \subseteq \bar{v} \lor v' \subseteq \bar{v} \). Hence, by assuming that \( \gamma(\bigcap \hat{V}'') = \bigcap \{ \gamma(\bar{v}) \mid \bar{v} \in \hat{V}'' \} \) (this is the induction assumption),
\[
\gamma(\bigcap \hat{V}') \quad \text{calc.}
\]
\[
\begin{align*}
\text{cond. 1 and 3} & \Rightarrow \\
\text{ind. ass.} & \Rightarrow (\bigcap \{ \gamma(\bar{v}) \mid \bar{v} \in \hat{V}'' \}) \cap \gamma(\bar{v}') \\
\text{calc.} & \Rightarrow \bigcap \{ \gamma(\bar{v}) \mid \bar{v} \in \hat{V}'' \} \cup \{ \gamma(\bar{v}') \} \\
\text{calc.} & \Rightarrow \bigcap \{ \gamma(\bar{v}) \mid \bar{v} \in \hat{V}' \}
\end{align*}
\]
which proves the lemma.

\section{3.2 Constructing Complete Lattices}

There are several different ways to construct complete lattices. Any given set can be \textit{lifed} into a complete lattice (Theorem 3.5).

\textbf{Theorem 3.5 (Complete lattice – Lifting):}
\textit{If }\( S \text{ is a set, then } (\mathcal{P}(S), \subseteq, \cup, \bigcap, \emptyset, S) \text{ is a complete lattice.}\)

\textbf{PROOF.} Assume that \( S \) is a set and let \( S^\mathcal{P} \subseteq \mathcal{P}(S) \). It is then trivially the case that \( \bigcup S^\mathcal{P} = \bigcup S^\mathcal{P}, \bigcap S^\mathcal{P} = \bigcap S^\mathcal{P}, \bot = \emptyset \) and \( T = S \) if \( \subseteq \subseteq \subseteq \) (note that \( \subseteq \) is reflexive, transitive and anti-symmetric by definition).

The \textit{Cartesian product} of two complete lattices is a complete lattice (Theorem 3.6).
Theorem 3.6 (Complete lattice – Cartesian product):
If \( \langle V_1, \subseteq_1, \cup_1, \cap_1, \bot_1, \top_1 \rangle \) and \( \langle V_2, \subseteq_2, \cup_2, \cap_2, \bot_2, \top_2 \rangle \) are complete lattices, then so is \( \langle V, \subseteq, \cup, \cap, \bot, \top \rangle \) where (let \( V' \subseteq V \)):

\[
V = V_1 \times V_2 = \{(v_1, v_2) \mid v_1 \in V_1 \land v_2 \in V_2\}
\]

\[
(\forall v_1, v_2) \subseteq (v'_1, v'_2) \iff v_1 \subseteq_1 v'_1 \land v_2 \subseteq_2 v'_2 \quad \text{where} \quad v_1, v'_1 \in V_1 \land v_2, v'_2 \in V_2
\]

\[
\bigcup V' = \{v_1 \in V_1 \mid \exists v_2 \in V_2 : (v_1, v_2) \in V'\},
\]

\[
\bigcap V' = \{v_2 \in V_2 \mid \exists v_1 \in V_1 : (v_1, v_2) \in V'\}
\]

\[
\bot = (\bot_1, \bot_2)
\]

\[
\top = (\top_1, \top_2)
\]

PROOF. Assume that \( \langle V_1, \subseteq_1, \cup_1, \cap_1, \bot_1, \top_1 \rangle \) and \( \langle V_2, \subseteq_2, \cup_2, \cap_2, \bot_2, \top_2 \rangle \) are complete lattices and let \( V = \{(v_1, v_2) \mid v_1 \in V_1 \land v_2 \in V_2\} \) and \( (v_1, v_2) \subseteq (v'_1, v'_2) \iff v_1 \subseteq_1 v'_1 \land v_2 \subseteq_2 v'_2 \) where \( v_1, v'_1 \in V_1 \) and \( v_2, v'_2 \in V_2 \). (Note that it is straightforward to verify that \( V, \subseteq \) is a partially ordered set since \( \subseteq_1 \) and \( \subseteq_2 \) are partial orders.) Also assume that \( V' \subseteq V \).

Since \( \bigcup_1 \{v_1 \in V_1 \mid \exists v_2 \in V_2 : (v_1, v_2) \in V\} \subseteq_1 V'_1 \) for all upper bounds, \( V'_1 \), of \( \{v_1 \in V_1 \mid \exists v_2 \in V_2 : (v_1, v_2) \in V\} \) and \( \bigcup_2 \{v_2 \in V_2 \mid \exists v_1 \in V_1 : (v_1, v_2) \in V\} \subseteq_2 V'_2 \) for all upper bounds, \( V'_2 \), of \( \{v_2 \in V_2 \mid \exists v_1 \in V_1 : (v_1, v_2) \in V\} \), it is easy to see that \( \bigcup V' = (\bigcup_1 \{v_1 \in V_1 \mid \exists v_2 \in V_2 : (v_1, v_2) \in V\}, \bigcup_2 \{v_2 \in V_2 \mid \exists v_1 \in V_1 : (v_1, v_2) \in V\}) \subseteq (V'_1, V'_2) \) (cf. the definition of \( \subseteq \) above). \( \bigcap V' \) is shown in a similar manner.

Since \( \bot_1 = \bigcup_1 \emptyset \) and \( \bot_2 = \bigcup_2 \emptyset \), it is easy to see that \( \bot = (\bigcup_1 \emptyset, \bigcup_2 \emptyset) = (\bot_1, \bot_2) \). \( \top \) is shown in a similar manner.

A space of total functions where the domain of the functions is a set and the range is a complete lattice is itself a complete lattice (Theorem 3.7).

Theorem 3.7 (Complete lattice – Total function space):
If \( S \) is a set and \( \langle V_1, \subseteq_1, \cup_1, \cap_1, \bot_1, \top_1 \rangle \) is a complete lattice, then \( \langle V, \subseteq, \cup, \cap, \bot, \top \rangle \) where (let \( V' \subseteq V \)):

\[
V = S \rightarrow V_1 = \{f : S \rightarrow V_1 \mid f \text{ is a total function}\}
\]

\[
f \subseteq f' \iff \forall s \in S : f(s) \subseteq_1 f'(s) \quad \text{where} \quad f, f' \in V
\]

\[
\bigcup V' = \bigcup S \bigcup \{f(s) \mid f \in V'\},
\]

\[
\bigcap V' = \bigcap S \bigcap \{f(s) \mid f \in V'\},
\]

\[
\bot = \bigcup \{s \in S \mid \bot_1\}
\]

\[
\top = \bigcup \{s \in S \mid \top_1\}
\]
is also a complete lattice. \hfill \Box

**Proof.** Assume that $S$ is a set and $\langle V_1, \subseteq_1, \cup_1, \cap_1, \perp_1, \top_1 \rangle$ is a complete lattice, $V = S \rightarrow V_1 = \{ f : S \rightarrow V_1 \mid f \text{ is a total function} \}$ and $f \subseteq f' \iff \forall s \in S : f(s) \subseteq f'(s)$ where $f, f' \in V$. (Note that it is straightforward to verify that $(V, \subseteq)$ is a partially ordered set.) Also assume that $V' \subseteq V$. Note that the totality of $f \in V$ will be implicitly used.

It is easy to see that $\forall s \in S: \forall f' \in V': f'(s) \subseteq_1 \bigcup_1 \{ f(s) \mid f \in V' \}$ and that $\forall s \in S: \bigcup_1 \{ f(s) \mid f \in V' \} \subseteq_1 f''(s)$ for any $f'' \in V$ such that $\forall s \in S: \forall f' \in V': f'(s) \subseteq_1 f''(s)$ since $\langle V_1, \subseteq_1, \cup_1, \cap_1, \perp_1, \top_1 \rangle$ is a complete lattice. But, then it must be that $\bigcup V' = \lambda s \in S: \bigcup_1 \{ f(s) \mid f \in V' \}$ (cf. the definition of $\subseteq$ above).

$\bigcap V'$ is shown in a similar manner.

Since $\perp_1 = \bigcup_1 \emptyset$, it is easy to see that $\perp = \lambda s \in S: \bigcup_1 \emptyset = \lambda s \in S. \perp_1. \top$ is shown in a similar manner. \hfill \blacksquare

A space of monotone functions where both the domain and the range of the functions are complete lattices is itself a complete lattice (Theorem 3.8).

**Theorem 3.8 (Complete lattice – Monotone function space):**

If $\langle V_1, \subseteq_1, \cup_1, \cap_1, \perp_1, \top_1 \rangle$ and $\langle V_2, \subseteq_2, \cup_2, \cap_2, \perp_2, \top_2 \rangle$ are complete lattices, then so is $\langle V, \subseteq, \cup, \cap, \perp, \top \rangle$ where (let $V' \subseteq V$):

\[
V = V_1 \rightarrow V_2 = \{ f : V_1 \rightarrow V_2 \mid f \text{ is a monotone function} \}
\]

\[f \subseteq f' \iff \forall v_1 \in V_1 : f(v_1) \subseteq_2 f'(v_1) \text{ where } f, f' \in V\]

\[
\bigcup V' = \lambda v_1 \in V_1. \bigcup_2 \{ f(v_1) \mid f \in V' \},
\]

\[
\bigcap V' = \lambda v_1 \in V_1. \bigcap_2 \{ f(v_1) \mid f \in V' \},
\]

\[
\perp = \lambda v_1 \in V_1. \perp_2
\]

\[
\top = \lambda v_1 \in V_1. \top_2
\]  \hfill \Box

**Proof.** Similar to the proof of Theorem 3.7 with the addition that the monotonicity of $f \in V$ gives that $\forall v_1, v'_1 \in V_1 : v_1 \subseteq_1 v'_1 \Rightarrow f(v_1) \subseteq_2 f(v'_1)$ (cf. Definition 3.1). \hfill \blacksquare

### 3.3 Galois Connections & Galois Insertions

The concrete semantics of a programming language can be abstracted in many different ways. The choice of abstraction is done by defining an abstract domain. A domain is, in general, a complete lattice, and an abstract domain is
essentially the set of all possible abstract states that fit the definition of the domain. It is often shown that the abstract domain is a safe over-approximation of the concrete domain by deriving a Galois connection between the two domains [91]. A Galois connection between two domains (i.e., complete lattices), \( V \) and \( D \), is described by an abstraction function, \( \alpha \), and a concretization function, \( \gamma \), which must fulfill the criterion in Definition 3.9.

**Definition 3.9 (Galois connection):**

\( \langle \alpha : V \to D, \gamma : D \to V \rangle \) is a Galois connection iff \( \alpha \) and \( \gamma \) are monotone functions that fulfill

\[
\begin{align*}
\alpha \circ \gamma & \sqsubseteq_{D \to D} \lambda d. d \\
\gamma \circ \alpha & \sqsupseteq_{V \to V} \lambda v. v
\end{align*}
\]

for all \( v \in V \) and \( d \in D \), where \( V \) is the concrete domain and \( D \) is the abstract domain.

An often useful special case of a Galois connection is called a Galois insertion; cf. Definition 3.10.

**Definition 3.10 (Galois insertion):**

\( \langle \alpha : V \to D, \gamma : D \to V \rangle \) is a Galois insertion iff \( \alpha \) and \( \gamma \) are monotone functions that fulfill

\[
\begin{align*}
\alpha \circ \gamma & = \lambda d. d \\
\gamma \circ \alpha & \sqsupseteq_{V \to V} \lambda v. v
\end{align*}
\]

for all \( v \in V \) and \( d \in D \), where \( V \) is the concrete domain and \( D \) is the abstract domain.

A function in the concrete domain, \( f : V \to V \), can be safely approximated by a function in the abstract domain, \( \tilde{f} : D \to D \), iff \( \forall d \in D : f(\gamma(d)) \sqsubseteq V \gamma(\tilde{f}(d)) \). The best approximation is achieved by inducing \( f \) along \( \alpha \) [91]; cf. Definition 3.11.

**Definition 3.11 (Induced function):**

Assuming that \( \langle \alpha : V \to D, \gamma : D \to V \rangle \) is a Galois connection, the best approximation, \( \tilde{f} \), of \( f : V \to V \) in \( D \) is given by:

\[
\tilde{f} = \alpha \circ f \circ \gamma
\]

Sometimes, it is more convenient to work with adjunctions (cf. Definition 3.12) instead of Galois connections.
Definition 3.12 (Adjunction):\
\langle \alpha : V \rightarrow D, \gamma : D \rightarrow V \rangle is said to be an adjunction between the complete lattices
\( V = \langle V, \sqsubseteq_V, \sqcup_V, \sqcap_V, \perp_V, \top_V \rangle \) and \( D = \langle D, \sqsubseteq_D, \sqcup_D, \sqcap_D, \perp_D, \top_D \rangle \) iff \( \alpha \) and \( \gamma \) are total functions that satisfy
\[
\alpha(v) \sqsubseteq_D d \iff v \sqsubseteq_V \gamma(d)
\]
for all \( v \in V \) and \( d \in D \). \( \square \)

In fact, adjunctions are Galois connections (Theorem 3.13).

Theorem 3.13 (Adjunctions and Galois connections):\
\( \langle \alpha : V \rightarrow D, \gamma : D \rightarrow V \rangle \) is an adjunction iff it is a Galois connection. \( \square \)

**Proof** ([91]). First assume that \( \langle \alpha : V \rightarrow D, \gamma : D \rightarrow V \rangle \) is an adjunction. It will be proven that it also is a Galois connection by showing that \( \gamma \circ \alpha \sqsubseteq_V \lambda \nu. v \) and \( \alpha \circ \gamma \sqsubseteq_D \lambda d. d \). For any \( v \in V \), trivially \( \alpha(v) \sqsubseteq_D \alpha(v) \). Using that \( \alpha(v) \sqsubseteq_D d \Rightarrow v \sqsubseteq_V \gamma(d) \), it can be established that \( \gamma(\alpha(v)) \). Similarly, for any \( d \in D \), trivially \( \gamma(d) \sqsubseteq_V \gamma(d) \). Using that \( v \sqsubseteq_V \gamma(d) \Rightarrow \alpha(v) \sqsubseteq_D d \), it can be established that \( \alpha(\gamma(d)) \sqsubseteq_D d \). Thus, \( \langle \alpha : V \rightarrow D, \gamma : D \rightarrow V \rangle \) is a Galois connection.

Next assume that \( \langle \alpha : V \rightarrow D, \gamma : D \rightarrow V \rangle \) is a Galois connection. It will be proven that it also is an adjunction by showing that \( \alpha(v) \sqsubseteq_D d \Rightarrow v \sqsubseteq_V \gamma(d) \) and \( v \sqsubseteq_V \gamma(d) \Rightarrow \alpha(v) \sqsubseteq_D d \). So, first assume that \( \alpha(v) \sqsubseteq_D d \). Then, since \( \gamma \) is monotone, \( \gamma(\alpha(v)) \sqsubseteq_V \gamma(d) \). Using that \( \gamma \circ \alpha \sqsubseteq_V \lambda \nu. v \), it can be established that \( v \sqsubseteq_V \gamma(\alpha(v)) \sqsubseteq_V \gamma(d) \) as required. For the second part of the proof, assume that \( v \sqsubseteq_V \gamma(d) \). Then, since \( \alpha \) is monotone, \( \alpha(v) \sqsubseteq_D \alpha(\gamma(d)) \). Using that \( \alpha \circ \gamma \sqsubseteq_D \lambda d. d \), it can be established that \( \alpha(v) \sqsubseteq_D \alpha(\gamma(d)) \sqsubseteq_D d \) as required. \( \Box \)

The abstraction and concretization functions are strictly related as described by Lemma 3.14.

Lemma 3.14 (Relation between \( \alpha \) and \( \gamma \)):\
If \( V = \langle V, \sqsubseteq, \sqcup, \sqcap, \perp, \top \rangle \) and \( \bar{V} = \langle \bar{V}, \sqsubseteq, \sqcup, \sqcap, \perp, \top \rangle \) are complete lattices, and \( \langle \alpha : V \rightarrow \bar{V}, \gamma : \bar{V} \rightarrow V \rangle \) is a Galois connection between these lattices, then (let \( \bar{v} \in \bar{V} \) and \( \bar{v} \in \bar{V} \)):

1. \( \alpha \) uniquely determines \( \gamma \) by \( \gamma(\bar{v}) = \bigcup \{ v \mid \alpha(v) \sqsubseteq \bar{v} \} \) and \( \gamma \) uniquely determines \( \alpha \) by \( \alpha(v) = \bigcap \{ v \mid v \sqsubseteq \gamma(\bar{v}) \} \).

2. \( \alpha \) is completely additive and \( \gamma \) is completely multiplicative.

In particular, \( \alpha(\perp) = \perp \) and \( \gamma(\top) = \top \). \( \square \)
PROOF ([91]). Assume that \( V = \langle V, \subseteq, \cup, \cap, \perp, \top \rangle \) and \( \tilde{V} = \langle \tilde{V}, \subseteq, \tilde{\cup}, \tilde{\cap}, \tilde{\perp}, \top \rangle \) are complete lattices, \( \langle \alpha : V \rightarrow \tilde{V}, \gamma : \tilde{V} \rightarrow V \rangle \) is a Galois connection between these lattices, \( v \in V \) and \( \tilde{v} \in \tilde{V} \).

To show 1, it will first be shown that \( \gamma \) is determined by \( \alpha \). Since \( \langle \alpha : V \rightarrow \tilde{V}, \gamma : \tilde{V} \rightarrow V \rangle \) is an adjunction (Theorem 3.13), it must be that \( \gamma(\tilde{v}) = \bigcup \{ v \mid v \subseteq \gamma(\tilde{v}) \} = \bigcup \{ v \mid \alpha(v) \subseteq \tilde{v} \} \). Assume that both \( \langle \alpha, \gamma_1 \rangle \) and \( \langle \alpha, \gamma_2 \rangle \) are Galois connections, then \( \gamma_1(\tilde{v}) = \bigcup \{ v \mid v \subseteq \gamma_1(\tilde{v}) \} = \bigcup \{ v \mid \alpha(v) \subseteq \tilde{v} \} = \bigcup \{ v \mid v \subseteq \gamma_2(\tilde{v}) \} = \gamma_2(\tilde{v}) \), and thus, \( \gamma_1 = \gamma_2 \). This shows that \( \alpha \) uniquely determines \( \gamma \). Similarly, it must be that \( \alpha(v) = \bigcap \{ \tilde{v} \mid \alpha(v) \subseteq \tilde{v} \} = \bigcap \{ \tilde{v} \mid v \subseteq \gamma(\tilde{v}) \} \). This shows that \( \gamma \) uniquely determines \( \alpha \).

To show 2, consider \( V' \subseteq V \), then
\[
\alpha(\bigcup V') \subseteq \tilde{v} \quad \text{Th. 3.13} \quad \bigcup V' \subseteq \gamma(\tilde{v})
\]
\[
\text{calc.} \quad \forall v \in V' : v \subseteq \gamma(\tilde{v})
\]
\[
\text{Th. 3.13} \quad \forall v \in V' : \alpha(v) \subseteq \tilde{v}
\]
\[
\text{calc.} \quad \bigcap \{ \alpha(v) \mid v \in V' \} \subseteq \tilde{v}
\]
and it follows that \( \alpha(\bigcup V') = \bigcup \{ \alpha(v) \mid v \in V' \} \).

The proof that \( \gamma(\bigcap \tilde{V'}) = \bigcap \{ \gamma(\tilde{v}) \mid \tilde{v} \in \tilde{V'} \} \) is analogous.

Thus, by Lemma 3.15, it suffices to specify either a completely additive abstraction function or a completely multiplicative concretization function in order to obtain a Galois connection.

**Lemma 3.15 (Galois connection – Existence):**

If \( V = \langle V, \subseteq, \cup, \cap, \perp, \top \rangle \) and \( \tilde{V} = \langle \tilde{V}, \subseteq, \tilde{\cup}, \tilde{\cap}, \tilde{\perp}, \top \rangle \) are complete lattices, and

1. \( \alpha : V \rightarrow \tilde{V} \) is completely additive, then there exists a \( \gamma : \tilde{V} \rightarrow V \) such that \( \langle \alpha, \gamma \rangle \) is a Galois connection.

2. \( \gamma : \tilde{V} \rightarrow V \) is completely multiplicative, then there exists an \( \alpha : V \rightarrow \tilde{V} \) such that \( \langle \alpha, \gamma \rangle \) is a Galois connection.

**Proof** ([91]). Assume that \( V = \langle V, \subseteq, \cup, \cap, \perp, \top \rangle \) and \( \tilde{V} = \langle \tilde{V}, \subseteq, \tilde{\cup}, \tilde{\cap}, \tilde{\perp}, \top \rangle \) are complete lattices, \( v \in V \) and \( \tilde{v} \in \tilde{V} \).

To show 1, assume that \( \alpha \) is completely additive and define \( \gamma \) by:
\[
\gamma(\tilde{v}) = \bigcup \{ v' \mid \alpha(v') \subseteq \tilde{v} \}
\]
Then it must be that \( \alpha(v) \subseteq \tilde{v} \Rightarrow v \in \{ v' \mid \alpha(v') \subseteq \tilde{v} \} \Rightarrow v \subseteq \gamma(\tilde{v}) \), where the last implication follows from the definition of \( \gamma \). For the other direction, first
observe that \( v \subseteq \gamma(\tilde{v}) \Rightarrow \alpha(v) \sqsubseteq \alpha(\gamma(\tilde{v})) \) since \( \alpha \) is completely additive and thus monotone. Then,
\[
\alpha(\gamma(\tilde{v})) = \alpha(\bigsqcup \{v' \mid \alpha(v') \sqsubseteq \tilde{v}\}) = \bigsqcup \{\alpha(v') \mid \alpha(v') \sqsubseteq \tilde{v}\}
\]
and so \( v \subseteq \gamma(\tilde{v}) \Rightarrow \alpha(v) \sqsubseteq \tilde{v} \). Thus, \( \langle \alpha, \gamma \rangle \) is a Galois connection (Theorem 3.13).

The proof of 2 is similar.

\[ \Box \]

### 3.4 Constructing Galois Connections

A Galois connection can be constructed in several ways. The following theorems (except Theorem 3.21) specify some of them.

The Cartesian product can be used to combine two existing Galois connections (Theorem 3.16).

**Theorem 3.16 (Galois connection – Independent attribute method):**

If \( \langle \alpha_1 : V_1 \rightarrow D_1, \gamma_1 : D_1 \rightarrow V_1 \rangle \) and \( \langle \alpha_2 : V_2 \rightarrow D_2, \gamma_2 : D_2 \rightarrow V_2 \rangle \) are Galois connections, then so is \( \langle \alpha : (V_1 \times V_2) \rightarrow (D_1 \times D_2), \gamma : (D_1 \times D_2) \rightarrow (V_1 \times V_2) \rangle \), where

\[
\begin{align*}
\alpha((v_1, v_2)) &= (\alpha_1(v_1), \alpha_2(v_2)) \\
\gamma((d_1, d_2)) &= (\gamma_1(d_1), \gamma_2(d_2))
\end{align*}
\]

and \( (v_1, v_2) \in V_1 \times V_2 \) and \( (d_1, d_2) \in D_1 \times D_2 \).

**Proof** ([91]). Assume that \( \langle \alpha_1 : V_1 \rightarrow D_1, \gamma_1 : D_1 \rightarrow V_1 \rangle \) and \( \langle \alpha_2 : V_2 \rightarrow D_2, \gamma_2 : D_2 \rightarrow V_2 \rangle \) are Galois connections, \( (v_1, v_2) \in V_1 \times V_2 \) and \( (d_1, d_2) \in D_1 \times D_2 \). Note that \( V_1 \times V_2 \) and \( D_1 \times D_2 \) are complete lattices (Theorem 3.6).

First calculate the following.

\[
\begin{align*}
\alpha((v_1, v_2)) &\sqsubseteq_D (d_1, d_2) \\
\text{Def. } \alpha &\iff \langle \alpha_1(v_1), \alpha_2(v_2) \rangle \sqsubseteq_D (d_1, d_2) \\
\text{calc. } \alpha_1(v_1) &\sqsubseteq_{D_1} d_1 \land \alpha_2(v_2) \sqsubseteq_{D_2} d_2 \\
\text{Th. 3.13 } v_1 &\sqsubseteq_{V_1} \gamma_1(d_1) \land v_2 \sqsubseteq_{V_2} \gamma_2(d_2) \\
\text{calc. } (v_1, v_2) &\sqsubseteq_V (\gamma_1(d_1), \gamma_2(d_2)) \\
\text{Def. } \gamma &\iff (v_1, v_2) \sqsubseteq_V \gamma((d_1, d_2))
\end{align*}
\]

Then, using Theorem 3.13, the result follows.

\[ \Box \]
Then, using Theorem 3.13, the result follows.

**Theorem 3.17 (Galois connection – Lifted independent attribute method):**

If \( \langle \alpha_1 : \mathcal{P}(V_1) \rightarrow D_1, \gamma_1 : D_1 \rightarrow \mathcal{P}(V_1) \rangle \) and \( \langle \alpha_2 : \mathcal{P}(V_2) \rightarrow D_2, \gamma_2 : D_2 \rightarrow \mathcal{P}(V_2) \rangle \) are Galois connections, then so is \( \langle \alpha : \mathcal{P}(V_1 \times V_2) \rightarrow (D_1 \times D_2), \gamma : (D_1 \times D_2) \rightarrow \mathcal{P}(V_1 \times V_2) \rangle \), where

\[
\begin{aligned}
\alpha(V) &= (\alpha_1(\{v_1 \in V_1 \mid \exists v_2 \in V_2 : (v_1, v_2) \in V\}), \\
\alpha_2(\{v_2 \in V_2 \mid \exists v_1 \in V_1 : (v_1, v_2) \in V\})
\end{aligned}
\]

and \( V \subseteq V_1 \times V_2 \) and \( (d_1, d_2) \in D_1 \times D_2 \).

**Proof.** Assume that \( \langle \alpha_1 : \mathcal{P}(V_1) \rightarrow D_1, \gamma_1 : D_1 \rightarrow \mathcal{P}(V_1) \rangle \) and \( \langle \alpha_2 : \mathcal{P}(V_2) \rightarrow D_2, \gamma_2 : D_2 \rightarrow \mathcal{P}(V_2) \rangle \) are Galois connections, \( V \subseteq V_1 \times V_2 \) and \( (d_1, d_2) \in D_1 \times D_2 \). Note that \( \mathcal{P}(V_1 \times V_2) \) and \( D_1 \times D_2 \) are complete lattices (Theorems 3.5 and 3.6).

First, calculate

\[
\alpha(V) \subseteq (d_1, d_2) \quad \text{Def. } \alpha \quad (\alpha_1(V'_1), \alpha_2(V'_2)) \subseteq (d_1, d_2)
\]

\[
\text{calc.} \quad \alpha_1(V'_1) \subseteq \alpha_1(d_1) \land \alpha_2(V'_2) \subseteq \alpha_2(d_2)
\]

\[
\text{Th. 3.13} \quad V'_1 \subseteq \gamma_1(d_1) \land V'_2 \subseteq \gamma_2(d_2)
\]

\[
\text{calc.} \quad V'_1 \times V'_2 \subseteq \gamma_1(d_1) \times \gamma_2(d_2)
\]

\[
\text{Def. } \gamma \quad V'_1 \times V'_2 \subseteq \gamma((d_1, d_2))
\]

\[
\text{Th. 3.13} \quad V \subseteq \gamma((d_1, d_2))
\]

where \( V'_1 = \{v_1 \in V_1 \mid \exists v_2 \in V_2 : (v_1, v_2) \in V\} \) and \( V'_2 = \{v_2 \in V_2 \mid \exists v_1 \in V_1 : (v_1, v_2) \in V\} \). Then, using Theorem 3.13, the result follows.

Both the concrete and abstract domains of an existing Galois connection can be lifted to derive a new Galois connection (Theorem 3.20). Note that Lemmas 3.18 and 3.19 give that the specified abstraction and concretization functions are monotone.

**Lemma 3.18 (Monotonicity of \( \alpha \)):**

The function \( \alpha : \mathcal{P}(V) \rightarrow \mathcal{P}(D) \), defined as

\[
\alpha(V') = \{\alpha(v) \mid v \in V'\}
\]

where \( V' \subseteq V \), \( \alpha \) is monotone and \( \alpha : V \rightarrow D \), is monotone.
The function \( \alpha \) and \( D \) are monotone.

**Proof.** This proof amounts to showing that \( \forall V', V'' \in \mathcal{P}(V) : (V' \subseteq V'' \Rightarrow \alpha_\mathcal{P}(V') \subseteq \alpha_\mathcal{P}(V'')) \).

Assume that \( V', V'' \in \mathcal{P}(V) \) and that \( V' \subseteq V'' \). Then, by definition:

\[
\alpha_\mathcal{P}(V'') \overset{\text{Def.}}{=} \alpha_\mathcal{P}(V') \subseteq \alpha_\mathcal{P}(V''),
\]

where the rewriting of \( \alpha(V'') \) and the set splitting are possible since \( V' \subseteq V'' \) and \( \alpha \) is monotone.

Thus, it has been shown that \( \alpha_\mathcal{P} \) is monotone. \( \blacksquare \)

**Lemma 3.19 (Monotonicity of \( \gamma_\mathcal{P} \)):**

The function \( \gamma_\mathcal{P} : \mathcal{P}(D) \to \mathcal{P}(V) \), defined as

\[
\gamma_\mathcal{P}(D') = \{ v \in V \mid \alpha(v) \in D' \}
\]

where \( D' \subseteq D \), \( \alpha \) is monotone and \( \gamma : D \to V \), is monotone. \( \square \)

**Proof.** This proof amounts to showing that \( \forall D', D'' \in \mathcal{P}(D) : (D' \subseteq D'' \Rightarrow \gamma_\mathcal{P}(D') \subseteq \gamma_\mathcal{P}(D'')) \).

Assume that \( D', D'' \in \mathcal{P}(D) \) and that \( D' \subseteq D'' \). Then, by definition:

\[
\gamma_\mathcal{P}(D'') \overset{\text{Def.}}{=} \gamma_\mathcal{P}(D') \subseteq \gamma_\mathcal{P}(D''),
\]

where the rewriting of \( D'' \) and the set splitting are possible since \( D' \subseteq D'' \) and \( \alpha \) is monotone.

Thus, \( \gamma_\mathcal{P}(D') \subseteq \gamma_\mathcal{P}(D'') \), and hence it has been shown that \( \gamma_\mathcal{P} \) is monotone. \( \blacksquare \)
Theorem 3.20 (Galois connection – Double lifting):
If \( \langle \alpha : V \to D, \gamma : D \to V \rangle \) is a Galois connection, then so is \( \langle \alpha_\mathcal{P} : \mathcal{P}(V) \to \mathcal{P}(D), \gamma_\mathcal{P} : \mathcal{P}(D) \to \mathcal{P}(V) \rangle \), where

\[
\begin{align*}
\alpha_\mathcal{P}(V') &= \{ \alpha(v) \mid v \in V' \} \\
\gamma_\mathcal{P}(D') &= \{ v \in V \mid \alpha(v) \in D' \}
\end{align*}
\]

and \( V' \subseteq V \) and \( D' \subseteq D \).

PROOF. Assume that \( \langle \alpha : V \to D, \gamma : D \to V \rangle \) is a Galois connection. Note that \( \mathcal{P}(V) \) and \( \mathcal{P}(D) \) are complete lattices (Theorem 3.5).

Since \( \alpha_\mathcal{P} \) and \( \gamma_\mathcal{P} \) are monotone (Lemmas 3.18 and 3.19, respectively), this proof amounts to showing that (cf. Definition 3.9)

1. \( \gamma_\mathcal{P}(\alpha_\mathcal{P}(V')) \supseteq V' \)
2. \( \alpha_\mathcal{P}(\gamma_\mathcal{P}(D')) \subseteq D' \)

where \( V' \subseteq V \) and \( D' \subseteq D \). Note that both cases trivially hold if \( V' = \emptyset \) or \( D' = \emptyset \), which corresponds to the bottom elements in the two lattices. Therefore, assume that \( V' \neq \emptyset \) and \( D' \neq \emptyset \).

For case 1, assume that \( V' \subseteq V \). Then, by definition:

\[
\gamma_\mathcal{P}(\alpha_\mathcal{P}(V')) = \{ v \in V \mid \alpha(v) \in \{ \alpha(v') \mid v' \in V' \} \}
\]

Assume that \( v'' \in V' \), then it must be that \( \alpha(v'') \in \{ \alpha(v') \mid v' \in V' \} \). But, then \( v'' \in \gamma_\mathcal{P}(\alpha_\mathcal{P}(V')) \) and thus \( \gamma_\mathcal{P}(\alpha_\mathcal{P}(V')) \supseteq V' \).

For case 2, assume that \( D' \subseteq D \). Then, by definition:

\[
\alpha_\mathcal{P}(\gamma_\mathcal{P}(D')) = \{ \alpha(v) \mid v \in \{ v' \in V \mid \alpha(v') \in D' \} \}
\]

Assume that \( d \in \alpha_\mathcal{P}(\gamma_\mathcal{P}(D')) \). Then it must be that \( \exists v \in \{ v' \in V \mid \alpha(v') \in D' \} : d = \alpha(v) \). Hence, for that \( v \), it must be that \( \alpha(v) \in D' \), and therefore, \( d \in D' \). Thus, \( \alpha_\mathcal{P}(\gamma_\mathcal{P}(D')) \subseteq D' \).

It might be tempting to use the definition of \( \alpha_\mathcal{P} \) and \( \gamma_\mathcal{P} \) as given in Theorem 3.21, but as the theorem shows, this does not result in a Galois connection.

Theorem 3.21 (Not a Galois connection – Double lifting):
If \( \langle \alpha : V \to D, \gamma : D \to V \rangle \) is a Galois connection, then \( \langle \alpha' : \mathcal{P}(V) \to \mathcal{P}(D), \gamma' : \mathcal{P}(D) \to \mathcal{P}(V) \rangle \) is not a Galois connection, where

\[
\begin{align*}
\alpha'(V') &= \{ \alpha(v) \mid v \in V' \} \\
\gamma'(D') &= \{ \gamma(d) \mid d \in D' \}
\end{align*}
\]

and \( V' \subseteq V \) and \( D' \subseteq D \).
PROOF. Assume that \( \langle \alpha : V \to D, \gamma : D \to V \rangle \) is a Galois connection. From the definition of \( \alpha'_{\gamma} \) and \( \gamma'_{\alpha} \), it clearly follows that they are monotone since \( \alpha \) and \( \gamma \) are (cf. Lemma 3.18).

By way of contradiction, assume that \( \langle \alpha'_{\gamma}, \gamma'_{\alpha} \rangle \) is a Galois connection. Then, by Definition 3.9, \( \gamma'_{\alpha}(\alpha'_{\gamma}(V')) \supseteq V' \). A closer look at \( \gamma'_{\alpha}(\alpha'_{\gamma}(V')) \) reveals that:
\[
\gamma'_{\alpha}(\alpha'_{\gamma}(V')) = \{ \gamma(d) \mid d \in \{ \alpha(v) \mid v \in V' \} \}
\]
Assume that \( v' \in V' \), then \( v' \in \gamma'_{\alpha}(\alpha'_{\gamma}(V')) \) since \( \gamma'_{\alpha}(\alpha'_{\gamma}(V')) \supseteq V' \). This means that \( \exists d' \in \{ \alpha(v) \mid v \in V' \} : d' = \alpha(v') \) and hence, for this \( d' \), \( \forall v'' \in \{ \gamma(d) \mid d \in \{ \alpha(v) \mid v \in V' \} \} : v' = v'' = \gamma(d') = \gamma(\alpha(v')) \).

But, since \( \langle \alpha, \gamma \rangle \) is a Galois connection, \( \gamma(\alpha(v')) \supseteq v' \). This means that it could be the case that \( \gamma(\alpha(v')) \not\supseteq v' \), and thus \( v' \neq v'' \), which means that \( \gamma'_{\alpha}(\alpha'_{\gamma}(V')) \not\supseteq V' \) is possible. Thus, \( \langle \alpha'_{\gamma}, \gamma'_{\alpha} \rangle \) is not a Galois connection. \( \blacksquare \)

The domains of a Galois connection can be extended to spaces of (total or monotone) functions (Theorem 3.22).

**Theorem 3.22 (Galois connection – Function space):**
If \( \langle \alpha : V \to D, \gamma : D \to V \rangle \) is a Galois connection, then so is \( \langle \alpha' : (S \to V) \to (S \to D), \gamma' : (S \to D) \to (S \to V) \rangle \) for some set, \( S \), where:
\[
\begin{align*}
\alpha'(f) &= \alpha \circ f \\
\gamma'(g) &= \gamma \circ g
\end{align*}
\]

PROOF ([91]). Assume that \( \langle \alpha : V \to D, \gamma : D \to V \rangle \) is a Galois connection and that \( S \) is a set. Note that \( S \to V \) and \( S \to D \) are complete lattices (Theorems 3.7 and 3.8).

First note that \( \alpha' \) and \( \gamma' \) are monotone since \( \alpha \) and \( \gamma \) are. Furthermore, since \( \langle \alpha, \gamma \rangle \) is a Galois connection,
\[
\gamma'(\alpha'(f)) = \gamma \circ \alpha \circ f \supseteq f
\]
and
\[
\alpha'((\gamma'(g)) = \alpha \circ \gamma \circ g \supseteq g
\]
and, thus, the theorem holds. \( \blacksquare \)

A lifted concrete domain of a Galois connection can be extended to a lifted space of (total or monotone) functions when also extending the abstract domain (Theorem 3.24). Note that Lemma 3.23 gives that the concretization function is monotone (otr w is short for otherwise).
3.4 Constructing Galois Connections

Lemma 3.23 (Monotonicity of \( \gamma_s \)):
The function \( \gamma_s : (S \rightarrow D) \rightarrow \mathcal{P}(S \rightarrow V) \), defined as
\[
\gamma_s(d) = \begin{cases} 
S \rightarrow V & \text{if } d = \top \\
\emptyset & \text{if } d = \bot \\
\{ f \in S \rightarrow V \mid \forall s \in S : (f \ s) \in \gamma(d \ s) \} & \text{otrw}
\end{cases}
\]
for some set \( S \) and complete lattices \( V \) and \( D \), is monotone, given that \( \gamma : D \rightarrow \mathcal{P}(V) \) is a monotone function and \( d \in S \rightarrow D \).

Proof. This proof amounts to showing that \( \forall d', d'' \in S \rightarrow D : (d' \subseteq d'' \Rightarrow \gamma_s(d') \subseteq \gamma_s(d'')) \), which is trivially the case if \( d' = \top \) or \( d'' = \top \).

Assume that \( \gamma : D \rightarrow \mathcal{P}(V) \) is a monotone function, \( d', d'' \in S \rightarrow D \) and that \( d' \subseteq d'' \land d' \neq \top \land d'' \neq \top \). Then, by definition:
\[
\begin{align*}
\gamma_s(d') &= \{ f \in S \rightarrow V \mid \forall s \in S : (f \ s) \in \gamma(d' \ s) \} \\
\gamma_s(d'') &= \{ f \in S \rightarrow V \mid \forall s \in S : (f \ s) \in \gamma(d'' \ s) \}
\end{align*}
\]
Since \( \gamma \) is monotone, it must be that \( \forall s \in S : \gamma(d' \ s) \subseteq \gamma(d'' \ s) \). This means that
\[
\begin{align*}
\gamma_s(d'') &= \{ f \in S \rightarrow V \mid \forall s \in S : (f \ s) \in (\gamma(d'' \ s) \setminus \gamma(d' \ s)) \} \\
&\subseteq \{ f \in S \rightarrow V \mid \forall s \in S : (f \ s) \in (\gamma(d'' \ s) \setminus \gamma(d' \ s)) \} \\
&= \gamma_s(d') \cup \{ f \in S \rightarrow V \mid \forall s \in S : (f \ s) \in (\gamma(d'' \ s) \setminus \gamma(d' \ s)) \}
\end{align*}
\]
and thus, trivially, \( \gamma_s(d') \subseteq \gamma_s(d'') \).

\[\blacksquare\]

Theorem 3.24 (Galois connection – Lifted function space):
If \( \langle \alpha : \mathcal{P}(V) \rightarrow D, \gamma : D \rightarrow \mathcal{P}(V) \rangle \) is a Galois connection, then so is \( \langle \alpha_s : \mathcal{P}(S \rightarrow V) \rightarrow (S \rightarrow D), \gamma_s : (S \rightarrow D) \rightarrow \mathcal{P}(S \rightarrow V) \rangle \), for some set \( S \), where
\[
\begin{align*}
\alpha_s(V') &= \begin{cases} 
\top & \text{if } V' = S \rightarrow V \\
\bot & \text{if } V' = \emptyset \\
\lambda s \in S. \alpha(\{ v' \ s \mid v' \in V' \}) & \text{otrw}
\end{cases} \\
\gamma_s(d) &= \begin{cases} 
S \rightarrow V & \text{if } d = \top \\
\emptyset & \text{if } d = \bot \\
\{ f \in S \rightarrow V \mid \forall s \in S : (f \ s) \in \gamma(d \ s) \} & \text{otrw}
\end{cases}
\end{align*}
\]
and \( V' \subseteq S \rightarrow V \) and \( d \in S \rightarrow D \).

\[\blacksquare\]
Proof. Assume that \( \langle \alpha : \mathcal{P}(V) \to D, \gamma : D \to \mathcal{P}(V) \rangle \) is a Galois connection, \( S \) is a set, \( V' \subseteq S \to V \) and \( d \in S \to D \). Note that \( \mathcal{P}(S \to V) \) and \( S \to D \) are complete lattices (Theorems 3.5, 3.7 and 3.8).

First note that:

\[
\gamma_s(\alpha_s(S \to V)) = \gamma_s(\top) = S \to V \supseteq S \to V
\]

\[
\gamma_s(\alpha_s(\emptyset)) = \gamma_s(\bot) = \emptyset \supseteq \emptyset
\]

\[
\alpha_s(\gamma_s(\top)) = \alpha_s(S \to V) = \top \subseteq \top
\]

\[
\alpha_s(\gamma_s(\bot)) = \alpha_s(\emptyset) = \bot \subseteq \bot
\]

Then note that \( \gamma_s \) is monotone (Lemma 3.23) and calculate the following.

\[
\alpha_s(V') \subseteq d \quad \text{Def. } \alpha_s \quad \lambda s \in S. \alpha(\{v' | v' \in V'\}) \subseteq d
\]

\[
\gamma_s \text{ mon. } \quad \gamma_s(\lambda s \in S. \alpha(\{v' | v' \in V'\})) \subseteq \gamma_s(d)
\]

\[
\text{Def. } \gamma_s \quad \{f \in S \to V | \forall s \in S : (f s) \in \gamma(\alpha(\{v' | v' \in V'\})) \} \subseteq \gamma_s(d)
\]

\[
\gamma_s(\alpha_s(\emptyset)) \quad \text{Def. } \gamma_s \quad \{f \in S \to V | \forall s \in S : (f s) \in \gamma(\alpha(\{v' | v' \in V'\})) \} \subseteq \gamma_s(d)
\]

\[
\text{Def. } \gamma_s \quad \{f \in S \to V : (f s) \in \alpha(\gamma(\{v' | v' \in V'\})) \} \subseteq \gamma_s(d)
\]

\[
\text{Def. } \gamma_s \quad \{f \in S \to V : (f s) \in \alpha(\gamma(\{v' | v' \in V'\})) \} \subseteq \gamma_s(d)
\]

\[
\text{Def. } \gamma_s \quad \{f \in S \to V : (f s) \in \alpha(\gamma(\{v' | v' \in V'\})) \} \subseteq \gamma_s(d)
\]

Then, using Theorem 3.13, the result follows.

The domains of a Galois connection can be indexed with the elements from some set (Theorem 3.25).

**Theorem 3.25 (Galois connection – Indexing):**
If \( \langle \alpha : V \to D, \gamma : D \to V \rangle \) is a Galois connection, then so is \( \langle \alpha' : (S \times V) \to (S \times D), \gamma' : (S \times D) \to (S \times V) \rangle \), for some set \( S \supseteq s \) (with the partial order =), where

\[
\left\{ \alpha'((s,v)) = (s, \alpha(v)) \right\}
\]

\[
\left\{ \gamma'((s',d)) = (s', \gamma(d)) \right\}
\]

and \( (s,v) \in S \times V \) and \( (s',d) \in S \times D \). The top elements, \( \top' \) and \( \top' \), correspond to the elements \( (s,v) \) and \( (s,d) \) for some \( s \in S \), respectively, where
\(\alpha(v) = \top_D\) and \(\gamma(d) = \top_V\). The bottom elements are defined in a corresponding manner.

\(\alpha'\) and \(\gamma'\) for \(\langle \alpha' : (V \times S) \to (D \times S), \gamma' : (D \times S) \to (V \times S) \rangle\) are defined similarly.

\[\]

**Proof.** Assume that \(\langle \alpha : V \to D, \gamma : D \to V \rangle\) is a Galois connection, \(S\) is a set, \((s, v) \in S \times V\) and \((s', d) \in S \times D\).

First note that:

\[
\begin{align*}
\gamma'(\alpha'(\top')) &= \gamma'(\top'_D) = \top'_V \sqcup_V \top'_V \\
\gamma'(\alpha'(\bot')) &= \gamma'(\bot'_D) = \bot'_V \sqcup_V \bot'_V \\
\alpha'(\gamma'(\top'_D)) &= \alpha'(\top'_V) = \top'_D \sqsubseteq_D \top'_D \\
\alpha'(\gamma'(\bot'_D)) &= \alpha'(\bot'_V) = \bot'_D \sqsubseteq_D \bot'_D
\end{align*}
\]

Then, calculate the following.

\[
\alpha'(s, v)) \sqsubseteq_{S \times D} (s', d) \quad \overset{\text{Def. } \alpha'}{=} \quad (s, \alpha(v)) \sqsubseteq_{S \times D} (s', d) \\
\overset{\text{calc.}}{=} \quad s = s' \land \alpha(v) \sqsubseteq_D d \\
\overset{\text{Th. 3.13}}{=} \quad s = s' \land v \sqsubseteq_V \gamma(d) \\
\overset{\text{calc.}}{=} \quad (s, v) \sqsubseteq_{S \times V} (s', \gamma(d)) \\
\overset{\text{Def. } \gamma'}{=} \quad (s, v) \sqsubseteq_{S \times V} \gamma'(s', d))
\]

Now, using Theorem 3.13, the result follows.

The proof for \(\langle \alpha' : (V \times S) \to (D \times S), \gamma' : (D \times S) \to (V \times S) \rangle\) being a Galois connection is conducted analogously.

\[\]

### 3.5 Constructing Galois Insertions

A Galois insertion \(\langle \alpha, \gamma \rangle\) between two domains, \(D\) and \(\tilde{D}\), can be constructed by following steps 1-5 below [41].

1. A domain, \(D\), with a partial order, \(\sqsubseteq\), a least (bottom) element, \(\bot\), a greatest (top) element, \(\top\), a greatest lower bound, \(\sqcap\), and a least upper bound, \(\sqcup\), so that \(\langle D, \sqsubseteq, \sqcup, \sqcap, \bot, \top \rangle\) is a complete lattice must be given.

2. Define a domain \(\tilde{D}\) and a monotone concretization function \(\gamma : \tilde{D} \to D\).

3. Define the partial order \(\preceq\) for \(\tilde{D}\).
4. The greatest lower bound $\bigcap$ and the least upper bound $\bigcup$ must exist for all subsets of $\bar{D}$. Then, by definition, $(\bar{D}, \subseteq, \bigcap, \bigcup, \top, \bot)$ is a complete lattice.

5. Define the abstraction function $\alpha : D \rightarrow \bar{D}$, which must be monotone.

Assuming that the domains $D$ and $\bar{D}$ and the monotone concretization function, $\gamma$, are defined, the partial ordering $\subseteq$ can easily be defined as given by Definition 3.26 [41].

**Definition 3.26 (Partial order):**
The partial order $\subseteq$ for the domain $\bar{D}$ is defined by $\forall d_1, d_2 \in \bar{D} : (d_1 \subseteq d_2 \iff \gamma(d_1) \subseteq \gamma(d_2))$. □

Based on this definition of the partial order, the greatest lower bound and least upper bound can be defined as given by Definitions 3.27 and 3.28, respectively [41].

**Definition 3.27 (Greatest lower bound):**
The element $\bar{d} \in \bar{D}$ is a lower bound of $\bar{D}' \subseteq \bar{D}$ iff $\forall d' \in \bar{D}' : \bar{d} \subseteq d'$. The element $\bar{d} \in \bar{D}$ is the greatest lower bound of $\bar{D}' \subseteq \bar{D}$ ($\bar{d} = \bigcap \bar{D}'$) iff $\bar{d}$ is a lower bound of $\bar{D}'$ and for all other lower bounds $\bar{d}'$ of $\bar{D}'$, $\bar{d}' \subseteq \bar{d}$. □

**Definition 3.28 (Least upper bound):**
The element $\bar{d} \in \bar{D}$ is an upper bound of $\bar{D}' \subseteq \bar{D}$ iff $\forall d' \in \bar{D}' : \bar{d} \subseteq d'$. The element $\bar{d} \in \bar{D}$ is the least upper bound of $\bar{D}' \subseteq \bar{D}$ ($\bar{d} = \bigcup \bar{D}'$) iff $\bar{d}$ is an upper bound of $\bar{D}'$ and for all other upper bounds $\bar{d}'$ of $\bar{D}'$, $\bar{d} \subseteq \bar{d}'$. □

The abstraction function $\alpha$ can be defined based on the definition of the greatest lower bound operator as given by Definition 3.29 [41].

**Definition 3.29 (Abstraction function, $\alpha$):**
Given two domains $D$ and $\bar{D}$ and a monotone concretization function $\gamma : \bar{D} \rightarrow D$, the abstraction function $\alpha : D \rightarrow \bar{D}$ is defined by:

$$\alpha(d) = \bigcap \{ \bar{d} \mid d \subseteq \gamma(d) \}$$

where $d \in D$ and $\bar{d} \in \bar{D}$. □

Alternatively, assuming that two domains and a monotone abstraction function have been defined, the concretization function $\gamma$ can be defined based on the least upper bound operator as given by Definition 3.30 [41].
Definition 3.30 (Alternative definition – Concretization function, \( \gamma \)): Given two domains \( D \) and \( \bar{D} \) and a monotone abstraction function \( \alpha : D \to \bar{D} \), the concretization function \( \gamma : \bar{D} \to D \) is defined by:

\[
\gamma(d) = \bigcup \{d' | \alpha(d) \preceq d'\}
\]

where \( d \in D \) and \( d \in \bar{D} \).

\[\square\]

3.6 The Interval Domain

One example of an abstract domain for values is the interval domain [35, 41, 91]. The definition of an interval is given in Definition 3.31.

**Definition 3.31 (Interval):**

An interval is defined as \([n_1, n_2]\), where \( n_1, n_2 \in \text{Val} = \mathbb{Z} \cup \{-\infty, \infty\} \) are the lower and upper bounds of the interval, respectively, and \( n_1 \leq n_2 \). Formally, the set of all intervals is defined as \( \text{Intv} = \{\perp_{\text{int}}, \top_{\text{int}}\} \cup \{[n_1, n_2] | n_1 \leq n_2 \land n_1, n_2 \in \text{Val}\} \), where \( \perp_{\text{int}} \) denotes an invalid interval (i.e., an empty interval or an interval where \( n_2 < n_1 \)) and \( \top_{\text{int}} = [-\infty, \infty] \) is greater than any other element of \( \text{Intv} \).

A Galois insertion will now be created between \( \mathcal{P}(\text{Val}) \) and \( \text{Intv} \), using the steps of Section 3.5. The concretization function \( \gamma_{\text{int}} : \text{Intv} \to \mathcal{P}(\text{Val}) \) is given by Definition 3.32.

**Definition 3.32 (Concretization of interval):**

\[
\gamma_{\text{int}}(i) = \begin{cases} 
\mathbb{Z} \cup \{-\infty, \infty\} & \text{if } i = \top_{\text{int}} \\
\emptyset & \text{if } i = \perp_{\text{int}} \\
\{n \in \text{Val} | n_1 \leq n \leq n_2\} & \text{otherwise (i.e., } i = [n_1, n_2] \}
\end{cases}
\]

The partial order relation for intervals, \( \subseteq_{\text{int}} \), is given by Definition 3.33 (using Definition 3.26).
Definition 3.33 (Partial order for intervals):

\[
\begin{cases}
i \subseteq \text{int} \land \text{int} \\
\bot \subseteq \text{int} i \\
[n_1, n_2] \subseteq \text{int} [n_1', n_2'] \iff n_1' \leq n_1 \land n_2 \leq n_2'
\end{cases}
\]

The greatest lower bound operator for intervals \(\cap\text{int}\) is defined as given by Definition 3.34 (using Definition 3.27).

Definition 3.34 (Greatest lower bound for intervals):

\[
\begin{cases}
i \cap \text{int} \land \text{int} i = i \\
i \cap \bot \text{int} = \bot \text{int} \land \text{int} i = \bot \text{int} \\
[n_1, n_2] \cap \text{int} [n_1', n_2'] = \\
\begin{cases}
\max\{\{n_1, n_1'\}, \min\{n_2, n_2'\}\} & \text{if } \max\{\{n_1, n_1'\}\} \leq \min\{\{n_2, n_2'\}\} \\
\bot_{\text{int}} & \text{otherwise (i.e., if } \max\{\{n_1, n_1'\}\} > \min\{\{n_2, n_2'\}\}\}
\end{cases}
\]

The least upper bound operator for intervals \(\cup\text{int}\) is defined as given by Definition 3.35 (using Definition 3.28).

Definition 3.35 (Least upper bound for intervals):

\[
\begin{cases}
i \cup \text{int} \land \text{int} i = \land \text{int} \cup \text{int} i = \land \text{int} \\
i \cup \bot \text{int} = \bot \text{int} \cup \text{int} i = i \\
[n_1, n_2] \cup \text{int} [n_1', n_2'] = [\min\{\{n_1, n_1'\}\}, \max\{\{n_2, n_2'\}\}]
\end{cases}
\]

The abstraction function \(\alpha_{\text{int}} : \mathcal{P}(\text{Val}) \rightarrow \text{Intv}\) is defined as given by Definition 3.36 (using Definition 3.29).

Definition 3.36 (Abstraction to interval):

\[
\alpha_{\text{int}}(V) = \begin{cases}
\land \text{int} & \text{if } V = \mathbb{Z} \cup \{-\infty, \infty\} \\
\bot_{\text{int}} & \text{if } V = \emptyset \\
[\min(V), \max(V)] & \text{otherwise (i.e., if } V \neq \mathbb{Z} \cup \{-\infty, \infty\} \text{ and } V \neq \emptyset\}
\end{cases}
\]

To show that \(\langle \alpha_{\text{int}}, \gamma_{\text{int}} \rangle\) is a Galois insertion, it would suffice to show that \(\gamma_{\text{int}}\) is monotone, since the steps of Section 3.5 have been used. However, for clarity, the entire proof is given in the proof of Theorem 3.39. Note that Lemmas 3.37 and 3.38 give that \(\gamma_{\text{int}}\) and \(\alpha_{\text{int}}\), respectively, are monotone.
Lemma 3.37 (Monotonicity of $\gamma_{\text{int}}$):
The function $\gamma_{\text{int}} : \text{Intv} \to \mathcal{P}(\text{Val})$ is monotone.

**Proof.** It should be shown that $\forall i, i' \in \text{Intv} : (i \sqsubseteq_{\text{int}} i' \Rightarrow \gamma_{\text{int}}(i) \subseteq \gamma_{\text{int}}(i'))$.

Note that the proof is trivial for the case that $i = \perp_{\text{int}}$ or $i' = \top_{\text{int}}$.

Assume that $i = [n_1, n_2] \in \text{Intv}$ and $i' = [n'_1, n'_2] \in \text{Intv}$, such that $i \sqsubseteq_{\text{int}} i'$. Further assume that $n \in \gamma_{\text{int}}(i)$. Then it must be the case that $n_1 \leq n \leq n_2$ (Definition 3.32). Since $i \sqsubseteq_{\text{int}} i'$, it must be the case that $n'_1 \leq n_1 \leq n \leq n_2 \leq n'_2$ (Definition 3.33). But, then it must be that $n \in \gamma_{\text{int}}(i')$ (Definition 3.32), and thus, $\gamma_{\text{int}}(i) \subseteq \gamma_{\text{int}}(i')$.

Lemma 3.38 (Monotonicity of $\alpha_{\text{int}}$):
The function $\alpha_{\text{int}} : \mathcal{P}(\text{Val}) \to \text{Intv}$ is monotone.

**Proof.** It should be shown that $\forall V, V' \in \mathcal{P}(\text{Val}) : (V \subseteq V' \Rightarrow \alpha_{\text{int}}(V) \sqsubseteq_{\text{int}} \alpha_{\text{int}}(V'))$.

Note that the proof is trivial for the case that $V = \emptyset$ or $V' = \mathbb{Z} \cup \{-\infty, \infty\}$.

Assume that $V, V' \in \mathcal{P}(\text{Val})$, such that $V \subseteq V'$. Further assume that $\alpha_{\text{int}}(V) = [n_1, n_2]$ and $\alpha_{\text{int}}(V') = [n'_1, n'_2]$. Since $V \subseteq V'$, it must be that $\forall v \in V : \{v\} \subseteq V'$, and hence, $\{n_1, n_2\} \subseteq V'$. But then, it must be that $\min(V') = n'_1 \leq n_1 = \min(V)$ and $\max(V') = n'_2 \leq n_2 = \max(V)$, and thus, $[n_1, n_2] \sqsubseteq_{\text{int}} [n'_1, n'_2]$ (Definition 3.33), which means that $\alpha_{\text{int}}(V) \sqsubseteq_{\text{int}} \alpha_{\text{int}}(V')$.

Theorem 3.39 (Galois insertion – Intervals):
$\langle \alpha_{\text{int}} : \mathcal{P}(\text{Val}) \to \text{Intv}, \gamma_{\text{int}} : \text{Intv} \to \mathcal{P}(\text{Val}) \rangle$ is a Galois insertion.

**Proof.** The proof amounts to showing that the constraints in Definition 3.10 are fulfilled by $\langle \alpha_{\text{int}}, \gamma_{\text{int}} \rangle$. Note that $\mathcal{P}(\text{Val})$ and $\text{Intv}$ are complete lattices [91].

According to Lemmas 3.37 and 3.38, $\gamma_{\text{int}}$ and $\alpha_{\text{int}}$ are monotone. To show that $\alpha_{\text{int}}(\gamma_{\text{int}}(i)) = i$, assume that $i \in \text{Intv}$.

- If $i = \top_{\text{int}}$, then $\gamma_{\text{int}}(i) = \mathbb{Z} \cup \{-\infty, \infty\}$. Thus, $\alpha_{\text{int}}(\gamma_{\text{int}}(i)) = \alpha_{\text{int}}(\mathbb{Z} \cup \{-\infty, \infty\}) = \top_{\text{int}} = i$.

- If $i = \perp_{\text{int}}$, then $\gamma_{\text{int}}(i) = \emptyset$. Thus, $\alpha_{\text{int}}(\gamma_{\text{int}}(i)) = \alpha_{\text{int}}(\emptyset) = \perp_{\text{int}} = i$.

- Otherwise (i.e., if $i = [n_1, n_2]$) then $\gamma_{\text{int}}(i) = \{n \in \text{Val} | n_1 \leq n \leq n_2\}$. Thus, $\alpha_{\text{int}}(\gamma_{\text{int}}(i)) = \alpha_{\text{int}}(\{n \in \text{Val} | n_1 \leq n \leq n_2\}) = [n_1, n_2] = i$.

To show that $\gamma_{\text{int}}(\alpha_{\text{int}}(V)) \supseteq V$, assume that $V \in \mathcal{P}(\text{Val})$. 

\[\top_{\text{int}} \sqcup \{n \in \text{Val} | n_1 \leq n \leq n_2\} = \top_{\text{int}} \sqcup [n_1, n_2] = \top_{\text{int}} \]
• If $V = \mathbb{Z} \cup \{-\infty, \infty\}$, then $\alpha_{\text{int}}(V) = \top_{\text{int}}$. Thus, $\gamma_{\text{int}}(\alpha_{\text{int}}(V)) = \gamma_{\text{int}}(\top_{\text{int}}) = \mathbb{Z} \cup \{-\infty, \infty\} \supseteq \mathbb{Z} \cup \{-\infty, \infty\} = V$.

• If $V = \emptyset$, then $\alpha_{\text{int}}(V) = \bot_{\text{int}}$. Thus, $\gamma_{\text{int}}(\alpha_{\text{int}}(V)) = \gamma_{\text{int}}(\bot_{\text{int}}) = \emptyset \supseteq \emptyset = V$.

• Otherwise, $\alpha_{\text{int}}(V) = [\min(V), \max(V)]$. Thus, $\gamma_{\text{int}}(\alpha_{\text{int}}(V)) = \gamma_{\text{int}}([\min(V), \max(V)]) = \{n \in \text{Val} \mid \min(V) \leq n \leq \max(V)\} \supseteq V$. ■
Chapter 4

PPL: a Concurrent Programming Language

In this chapter, a concurrent programming language, PPL, will be defined. The language basically models a simple processor instruction set. This means that adapting the language to model (and thereby the analysis, presented in Chapter 6, to work on) the instruction set of a real processor could be done reasonably easy.

The concurrent entities of execution are referred to as threads and a PPL program consists of a static set of threads; i.e., dynamic thread creation and thread destruction are not featured. PPL provides both thread-private memory and memory that is globally shared between threads, referred to as registers, \( r \in \text{Reg} \), and variables, \( x \in \text{Var} \), respectively. Arithmetical operations and boolean comparisons can only be performed within a thread, using the values of the thread's registers. A thread can move data between its registers and the variables (in both directions) to for example achieve communication with other threads. PPL also provides shared resources, referred to as locks, \( lck \in \text{Lck} \), that can be acquired in a mutually exclusive manner by the threads and can hence be used for synchronizing threads. Currently, there is no fairness in how threads acquire locks (cf. Table 4.5). In other words, a thread could starve (wait forever on some lock) if there is at least one other thread that tries to acquire the lock at the same points in time as the considered thread. (Cf. the instruction set of a multi-core CPU, which typically provides access to both local and global memory, a shared memory bus and atomic, i.e., mutually exclusive, operations.)
Note that PPL does not provide functions or pointers. This decision was made in order to put focus on the challenges arising from parallelism: communication and synchronization between threads.

The operations (statements) provided by the instruction set may have variable execution times depending on the properties of the underlying architecture, which is further discussed below.

| NOTE. A summary of the notation and nomenclature used in this thesis can be found in Appendix A. |

The syntax of PPL, which is a set of operations using the discussed architectural features, is defined in Table 4.1. \( P \in \text{Prg} \) denotes a program, which simply is a (static) set of threads, i.e., \( P = \text{Thrd} \in \mathcal{P}(\text{ThrdID} \times \text{Stm}) = \text{Prg} \), where each thread, \( T \in \text{Thrd} \), is a pair of a unique identifier, \( d \in \text{ThrdID} \), and a statement, \( s \in \text{Stm} \) (note that a statement can be a sequence of statements; cf. \( s_1 ; s_2 \)). This makes every thread unique and distinguishable from other threads, even if several threads consist of the same statement. To increase the readability of the semantics, it will be assumed that the axiom-statements (all statements except the sequentially composed statement, \( s_1 ; s_2 \)) of each thread, \( T \in \text{Thrd} \), are uniquely labeled with consecutive labels, \( l \in \text{Lbl}_T = \mathbb{N}^+ \) (i.e., the set of labels is the set of positive integers), and stored in an array-like fashion in ascending order of their labels. \( a \in \text{Aexp} \) and \( b \in \text{Bexp} \) denote an arithmetical and a boolean expression, respectively, and \( n \in \text{Val} \) is an integer value, negative infinity or positive infinity; i.e., \( \text{Val} = \mathbb{Z} \cup \{-\infty, \infty\} \).

Locks can be acquired in a mutually exclusive manner using \text{lock} and released using \text{unlock}. Values can be transferred between variables and registers using \text{load} and \text{store}. Conditional branching is performed using \text{if}. A register is assigned a value using \text{:=}. A no-operation is performed using \text{skip}. And, \text{halt} stops the execution of the issuing thread. \text{halt} must be the last statement of each thread in the program, but it could also occur anywhere “within” a thread.

The semantics of PPL is formally defined in Section 4.2. Note that in the following, \( \text{Time} = \mathbb{Z} \cup \{-\infty, \infty\} \). In other words, discrete points in time, including negative and positive infinity, will be considered.
are uniquely labeled with consecutive labels, except the sequentially composed statement, of the semantics, it will be assumed that the axiom-statements (all statements even if several threads consist of the same statement. To increase the readability of the trace, which is further discussed below.

The operations (statements) provided by the instruction set may have various semantics depending on the properties of the underlying architectural features, is defined in Table 4.1.

<table>
<thead>
<tr>
<th>Table 4.1: The syntax of PPL.</th>
</tr>
</thead>
</table>

\[
P := \{T_1, \ldots, T_m\} \\
T := (d, s) \\
y := [halt]^l | [skip]^l | [r := a]^l | [if b goto l']^l | [load r from x]^l | [store r to x]^l | [lock lck]^l | [unlock lck]^l | s_1 ; s_2 \\
a := n | r | a_1 + a_2 | a_1 - a_2 | a_1 \times a_2 | a_1 / a_2 \\
b := \text{true} | \text{false} | !b | a_1 \& \& a_2 | a_1 == a_2 | a_1 <= a_2 \\
P = \text{Thrd} \in \text{Prg} : \text{Program; i.e., a set of threads} \\
\text{T} \in \text{Thrd} : \text{Thread within a program} \\
d \in \text{ThrdID} : \text{Unique thread identifier} \\
s \in \text{Stm} : \text{Statement} \\
l \in \text{Lbl}_T = \mathbb{N}^+ : \text{Unique statement-label within thread T} \\
r \in \text{Reg}_T : \text{Register; i.e., local memory in thread T} \\
a \in \text{Aexp} : \text{Arithmetical expression} \\
b \in \text{Bexp} : \text{Boolean expression} \\
x \in \text{Var} : \text{Variable; i.e., global memory} \\
lck \in \text{Lck} : \text{Lock} \\
n \in \text{Val} : \text{Integer} |
4.1 States & Configurations

A number of sub-states will be used when expressing how a set of given statements affects the state of the entire system when the statements are executed in parallel; i.e., when expressing the semantics of PPL. For each thread, \( T \), of a program, \( \text{Reg}_T \) is the set of register names defined in \( T \), and there is an instance of each of the following states.

\[ pc_T : \text{Lbl}_T \to \text{Val} \] – a program counter that keeps track of which label (i.e., statement) within the thread \( T \) that is active.

\[ \tau_T : \text{Reg}_T \to \text{Val} \] – a mapping from the registers of thread \( T \) to their values.

\[ t_T^\theta : \text{Time} \] – an absolute point in discrete time when the previous statement in \( T \) was executed.

For the program as a whole, common to all threads, there is an instance of each of the following states.

\[ \exists : \text{Var} \to \text{Thrd} \to \mathcal{P}(\text{Val} \times \text{Time}) \] – a nested mapping from variables and threads to a set of writes; i.e., a pair of a value and an absolute point in time.

\[ \exists : \text{Lck} \to (\text{Lck}_{\text{stt}} \times \text{Thrd}_\perp \times \text{Time} \times \text{Thrd}_\perp \times \text{Time}) \] – a mapping from locks to their values; i.e., a state \( (\text{Lck}_{\text{stt}} = \{\text{unlocked, locked}\}) \), a current owner \( (\text{Thrd}_\perp = \text{Thrd} \cup \{\perp_{\text{thrd}}\}) \), an absolute point in time (i.e., a deadline for) when the lock must have been taken by the current owner, a previous owner, and an absolute point in time when the lock was last released. For the case that no thread owns the lock, the owner is \( \perp_{\text{thrd}} \).

**NOTE.** The only information about locks that is needed in the concrete case is the current owner of each lock (cf. Tables 4.2 and 4.5). The rest of the information is only necessary when expressing the abstract semantics (cf. Chapter 5). However, the soundness of the...
abstract semantics is more easily proven if this information is included in the concrete case as well.

The types of the states \( \mathfrak{x} \) and \( \mathfrak{l} \) might look a bit peculiar at first glance; the need for their definitions will become apparent when defining an abstract interpretation of the PPL semantics in Section 5.8.

The above listed states, together with the threads of the program, will be referred to as a program state or configuration, \( c \in \text{Conf} \). \( \text{Conf} \) and \( c \) are defined as follows.

\[
\text{Conf} ::= (\prod_{T \in \text{Thrd}} \{T\} \times \text{Lbl}_T \times (\text{Reg}_T \rightarrow \text{Val}) \times \text{Time})) \times \n (\text{Var} \rightarrow \text{Thrd} \rightarrow \mathcal{P}(\text{Val} \times \text{Time})) \times 
(l\text{ck} \rightarrow (l\text{ck}_{\text{stat}} \times \text{Thrd}_\bot \times \text{Time} \times \text{Thrd}_\bot \times \text{Time}))
\]
\[
c ::= \langle [T, pc_T, v_T, t^d_T]_{T \in \text{Thrd}}, \mathfrak{x}, \mathfrak{l} \rangle
\]

Since it is not possible to beforehand determine the number of threads specified by a program, \( \prod_{T \in \text{Thrd}} \ldots \) expands to \( |\text{Thrd}| \) instances (i.e., one instance for each thread, \( T \), in a given program) of type \( \{T\} \times \text{Lbl}_T \times (\text{Reg}_T \rightarrow \text{Val}) \times \text{Time} \). Likewise, \( \ldots \prod_{T \in \text{Thrd}} \) is defined to expand in the corresponding manner. This way, \( c \in \text{Conf} \) can be regarded as a tuple with a known size when the number of threads in a program is known.

Sub-components of a configuration will also be of interest when considering an axiom statement of a single thread (see Table 4.2). Therefore, the following “smaller” configurations, \( \text{axConf}^{\text{in}} \in \text{axConf}^{\text{in}}_T \) and \( \text{axConf}^{\text{out}} \in \text{axConf}^{\text{out}}_T \), are defined for \( T \in \text{Thrd} \). \( \text{axConf}^{\text{in}} \) is used as input to the semantic rules for axiom statements and \( \text{axConf}^{\text{out}} \) is used as output from the rules (cf. Table 4.2).

\[
\text{axConf}^{\text{in}}_T ::= \{T\} \times \text{Lbl}_T \times (\text{Reg}_T \rightarrow \text{Val}) \times 
(\text{Var} \rightarrow \text{Thrd} \rightarrow \mathcal{P}(\text{Val} \times \text{Time})) \times 
(l\text{ck} \rightarrow (l\text{ck}_{\text{stat}} \times \text{Thrd}_\bot \times \text{Time} \times \text{Thrd}_\bot \times \text{Time})) \times \text{Time}
\]
\[
\text{axConf}^{\text{in}}_T ::= \langle [T, pc, v, \mathfrak{x}, \mathfrak{l}] \rangle
\]

\[
\text{axConf}^{\text{out}}_T ::= \text{Lbl}_T \times (\text{Reg}_T \rightarrow \text{Val}) \times 
(\text{Var} \rightarrow \text{Thrd} \rightarrow \mathcal{P}(\text{Val} \times \text{Time})) \times 
(l\text{ck} \rightarrow (l\text{ck}_{\text{stat}} \times \text{Thrd}_\bot \times \text{Time} \times \text{Thrd}_\bot \times \text{Time}))
\]
\[
\text{axConf}^{\text{out}}_T ::= \langle [pc, v, \mathfrak{x}, \mathfrak{l}] \rangle
\]
4.2 Semantics

The semantic rules for individual statements within a thread, the language axioms, are described by the relation $\xrightarrow{ax} : \text{axConf}^\text{in}_T \times \text{axConf}^\text{out}_T \rightarrow \{\text{true}, \text{false}\}$, which is formally defined in Table 4.2. Note that $\mathcal{A}$ and $\mathcal{B}$ are defined in Tables 4.3 and 4.4, respectively.

$\xrightarrow{ax}$ describes the semantics of a single statement within a thread when considered in isolation from all other threads. It should be noted that when the value for a name in a mapping is updated, a special syntax is used. Consider for example the register update statement, $r := a$. Here, the resulting value of the arithmetical expression $a$ (as given by $\mathcal{A}[a]$ where $\mathcal{A}$ is the register state) is written to register $r$. The updated register state, $\mathcal{A}[r \mapsto \mathcal{A}[a]]$, is such that for the register $r$, it has the value $\mathcal{A}[a]$, and for all other registers, it has the same value as $\mathcal{A}$. It should also be noted that the notation $\mathcal{A}[r]$ is used to denote the value of register $r$ in the register state $\mathcal{A}$.

The semantic rule for a set of threads (i.e., the program) is described by the relation $\xrightarrow{prg} : \text{Conf} \times \text{Conf} \rightarrow \{\text{true}, \text{false}\}$, which is defined based on $\xrightarrow{ax}$ as given in Table 4.5. Note that the functions STM and STT, OWN, DL, POWN and REL are defined in Tables 4.6 and 4.7, respectively, and that STM is a total function. STM($T, pc$) gives the active statement in thread $T$; i.e., the statement pointed to by $pc$. STT, OWN, DL, POWN and REL simply mask out the current state, the current owner, the deadline for the owner assignment, the previous owner and the point in time when the lock was last released, respectively, from a given lock value.

@ is used to introduce two notations for a single entity and $exp ? exp_1 : exp_2$ is $exp_1$ if $exp$, and $exp_2$ otherwise (cf. Appendix A).

To execute a program, or rather, to derive some possible execution trace for a given initial configuration, $c \in \text{Conf}$, a succeeding configuration is given by any $c' \in \text{Conf}$, such that $c \xrightarrow{prg} c'$. Then, a succeeding configuration, $c'' \in \text{Conf}$, to $c'$ is given by $c' \xrightarrow{prg} c''$, and so on. Since not all parts of Tables 4.2 and 4.5 are relevant for all statements, the $\xrightarrow{ax}$ and $\xrightarrow{prg}$ relations (i.e., the semantics of PPL) will be further explained below based on what statement is considered.

Note that $\xrightarrow{ax}$ and $\xrightarrow{prg}$ are relations, not functions, since one single input configuration can have several outputs; e.g., if two or more threads execute lock lck, where STT( lck) = unlocked, then lck is assigned to one of these threads by $\xrightarrow{prg}$ in a non-deterministic fashion.
Table 4.2: \(<T, pc, x, x, l, t> \rightarrow_{ax} <pc', x', x', l'>\), the semantics of concrete axiom transitions.

<table>
<thead>
<tr>
<th>STM(T, pc)</th>
<th>\langle pc', x', x', l' \rangle</th>
<th>If</th>
</tr>
</thead>
<tbody>
<tr>
<td>[halt]_{pc}</td>
<td>\langle pc, x, x, l \rangle</td>
<td></td>
</tr>
<tr>
<td>[skip]_{pc}</td>
<td>\langle pc + 1, r, x, l \rangle</td>
<td></td>
</tr>
<tr>
<td>[r := a]_{pc}</td>
<td>\langle pc + 1, r[r \mapsto a], x, l \rangle</td>
<td></td>
</tr>
<tr>
<td>[if b goto l]_{pc}</td>
<td>\langle pc + 1, r, x, l \rangle</td>
<td>\neg \mathcal{R}[b]x</td>
</tr>
<tr>
<td>[if b goto l]_{pc}</td>
<td>\langle pc + 1, r, x, l \rangle</td>
<td>\mathcal{R}[b]r</td>
</tr>
<tr>
<td>[store r to x]_{pc}</td>
<td>\langle pc + 1, r[x \mapsto x(lk)]</td>
<td></td>
</tr>
<tr>
<td>[load r from x]_{pc}</td>
<td>\langle pc + 1, r, x, l \rangle</td>
<td></td>
</tr>
<tr>
<td>[lock lck]_{pc}</td>
<td>\langle pc + 1, r, x, [lk \mapsto locked, T, DL(lck)], OWN(lck), REL(lck)]</td>
<td>OWN(lck) = T</td>
</tr>
<tr>
<td>[lock lck]_{pc}</td>
<td>\langle pc, x, l \rangle</td>
<td>OWN(lck) \neq T</td>
</tr>
<tr>
<td>[unlock lck]_{pc}</td>
<td>\langle pc + 1, r, x, l[lck \mapsto unlocked, T, T, DL(lck)], T, t] \rangle</td>
<td>OWN(lck) = T</td>
</tr>
<tr>
<td>[unlock lck]_{pc}</td>
<td>\langle pc + 1, r, x, l \rangle</td>
<td>OWN(lck) \neq T</td>
</tr>
</tbody>
</table>

where \( \mathcal{R}(r, x, x) = r[r \mapsto v] \) for some \( v \), such that

\[
\exists v' \in \text{Time} : (v, v') \in \bigcup_{T \in \text{Thrd}} ((x, x) T') \quad \text{if} \quad \bigcup_{T \in \text{Thrd}} ((x, x) T') \neq \emptyset
\]

or

\[
v \in \gamma_{\text{Int}}([-\infty, \infty])
\]
### Table 4.3: Semantics of concrete evaluation of arithmetic expressions.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n$</td>
<td>$\mathcal{A}[n] = n$</td>
</tr>
<tr>
<td>$r$</td>
<td>$\mathcal{A}[r] = r$</td>
</tr>
<tr>
<td>$a_i + a_2$</td>
<td>$\mathcal{A}[a_1 + a_2] = \mathcal{A}[a_1] + \mathcal{A}[a_2]$</td>
</tr>
<tr>
<td>$a_1 - a_2$</td>
<td>$\mathcal{A}[a_1 - a_2] = \mathcal{A}[a_1] - \mathcal{A}[a_2]$</td>
</tr>
<tr>
<td>$a_1 \times a_2$</td>
<td>$\mathcal{A}[a_1 \times a_2] = \mathcal{A}[a_1] \times \mathcal{A}[a_2]$</td>
</tr>
<tr>
<td>$a_1 / a_2$</td>
<td>$\mathcal{A}[a_1 / a_2] = \left[ \frac{\mathcal{A}[a_1]}{\mathcal{A}[a_2]} \right]$</td>
</tr>
</tbody>
</table>

### Table 4.4: Semantics of concrete evaluation of boolean expressions.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{true}$</td>
<td>$\mathcal{B}[\text{true}] \iff \text{true}$</td>
</tr>
<tr>
<td>$\text{false}$</td>
<td>$\mathcal{B}[\text{false}] \iff \text{false}$</td>
</tr>
<tr>
<td>$a_1 \land a_2$</td>
<td>$\mathcal{B}[a_1 \land a_2] \iff \mathcal{B}[a_1] \land \mathcal{B}[a_2]$</td>
</tr>
<tr>
<td>$a_1 \lor a_2$</td>
<td>$\mathcal{B}[a_1 \lor a_2] \iff \mathcal{B}[a_1] \lor \mathcal{B}[a_2]$</td>
</tr>
<tr>
<td>$a_1 = a_2$</td>
<td>$\mathcal{B}[a_1 = a_2] \iff \mathcal{A}[a_1] = \mathcal{A}[a_2]$</td>
</tr>
<tr>
<td>$a_1 &lt; a_2$</td>
<td>$\mathcal{B}[a_1 &lt; a_2] \iff \mathcal{A}[a_1] &lt; \mathcal{A}[a_2]$</td>
</tr>
<tr>
<td>$a_1 &gt; a_2$</td>
<td>$\mathcal{B}[a_1 &gt; a_2] \iff \mathcal{A}[a_1] &gt; \mathcal{A}[a_2]$</td>
</tr>
<tr>
<td>$a_1 \leq a_2$</td>
<td>$\mathcal{B}[a_1 \leq a_2] \iff \mathcal{A}[a_1] \leq \mathcal{A}[a_2]$</td>
</tr>
<tr>
<td>$a_1 \geq a_2$</td>
<td>$\mathcal{B}[a_1 \geq a_2] \iff \mathcal{A}[a_1] \geq \mathcal{A}[a_2]$</td>
</tr>
<tr>
<td>$a_1 =! a_2$</td>
<td>$\mathcal{B}[a_1 =! a_2] \iff \mathcal{A}[a_1] =! \mathcal{A}[a_2]$</td>
</tr>
<tr>
<td>$a_1 \neq a_2$</td>
<td>$\mathcal{B}[a_1 \neq a_2] \iff \mathcal{A}[a_1] \neq \mathcal{A}[a_2]$</td>
</tr>
</tbody>
</table>

---
Table 4.5: $c \xrightarrow{prg} c'$, the semantics of concrete program transitions.

\[
\text{Thrd}_{ex} \neq \emptyset \land \forall T \in \text{Thrd}_{ex} : \langle T, pc_{T}, t_{T}, x_{T}, \|'' \rangle, t''_{T} \xrightarrow{prg} \langle pc'_{T}, t'_{T}, x'_{T}, \|' \rangle,
\]

where
\[
t = \min(\{t''_{T} + \text{TIME}(c, T) | T \in \text{Thrd} \land \text{STM}(T, pc_{T}) \neq [\text{halt}]^{pc_{T}}\})
\]
\[
\text{Thrd}_{exe} = \{T \in \text{Thrd} | t = t''_{T} + \text{TIME}(c, T) \land \text{STM}(T, pc_{T}) \neq [\text{halt}]^{pc_{T}}\}
\]
\[
t''_{T} = \begin{cases} t''_{T} + \text{TIME}(c, T) & \text{if } T \in \text{Thrd}_{exe} \\ t''_{T} & \text{otherwise} \end{cases}
\]
\[
x' = \lambda T \in \text{Thrd}.(T = T' \land (x''_{T}, x) T' : \emptyset) \text{ otrw}
\]
\[
\text{where } T' \text{ is one of the threads in } \text{Thrd}_{x} = \{T \in \text{Thrd}_{exe} | \exists r \in \text{Reg}_{T} : \text{STM}(T, pc_{T}) = [\text{store } r \text{ to } x]^{pc_{T}}\}
\]
\[
\text{if } \{T \in \text{Thrd}_{exe} | \text{STM}(T, pc_{T}) = [\text{lock } lck]^{pc_{T}}\} \neq \emptyset \land
\]
\[
\text{OWN}(\downarrow lck) = \bot_{\text{thrd}} \text{ otrw}
\]
\[
\|'' lck = \begin{cases} \uparrow' lck & \text{if OWN}(\|'' lck) \in \text{Thrd}_{exe} \\ lck & \text{otherwise} \end{cases}
\]
\[
\|' lck = \begin{cases} \uparrow' \text{OWN}(\|'' lck) lck & \text{if OWN}(\|'' lck) \in \text{Thrd}_{exe} \\ lck & \text{otherwise} \end{cases}
\]
Table 4.6: Definition of $\text{STM}$ and $\text{LABELS}$.

$$\text{STM} : (\text{Thrd} \times \text{Lbl}) \rightarrow \text{Stm} = ((\text{ThrdID} \times \text{Stm}) \times \text{Lbl}) \rightarrow \text{Stm}$$

$$\text{STM}((d,s),pc) = \begin{cases} 
s & \text{if } s \text{ is an axiom statement} \\
\text{STM}((d,s'),pc) & \text{if } s = s' ; s'' \land pc \in \text{LABELS}(s') \\
\text{STM}((d,s''),pc) & \text{if } s = s' ; s'' \land pc \in \text{LABELS}(s'') 
\end{cases}$$

$$\text{LABELS} : \text{Stm} \rightarrow \mathcal{P}(\text{Lbl})$$

$$\text{LABELS}(s) = \begin{cases} 
\{l\} & \text{if } s \text{ is an axiom statement, } [\ldots] \\
\text{LABELS}(s') \cup \text{LABELS}(s'') & \text{if } s = s' ; s''
\end{cases}$$

Table 4.7: Definition of $\text{STT}$, $\text{OWN}$, $\text{DL}$, $\text{POWN}$ and $\text{REL}$.

$$\text{STT} : (\text{Lckstt} \times \text{Thrd}_\perp \times \text{Time} \times \text{Thrd}_\perp \times \text{Time}) \rightarrow \text{Lckstt}$$

$$\text{STT}((u,T,t,T',t')) = u$$

$$\text{OWN} : (\text{Lckstt} \times \text{Thrd}_\perp \times \text{Time} \times \text{Thrd}_\perp \times \text{Time}) \rightarrow \text{Thrd}_\perp$$

$$\text{OWN}((u,T,t,T',t')) = T$$

$$\text{DL} : (\text{Lckstt} \times \text{Thrd}_\perp \times \text{Time} \times \text{Thrd}_\perp \times \text{Time}) \rightarrow \text{Time}$$

$$\text{DL}((u,T,t,T',t')) = t$$

$$\text{POWN} : (\text{Lckstt} \times \text{Thrd}_\perp \times \text{Time} \times \text{Thrd}_\perp \times \text{Time}) \rightarrow \text{Thrd}_\perp$$

$$\text{POWN}((u,T,t,T',t')) = T'$$

$$\text{REL} : (\text{Lckstt} \times \text{Thrd}_\perp \times \text{Time} \times \text{Thrd}_\perp \times \text{Time}) \rightarrow \text{Time}$$

$$\text{REL}((u,T,t,T',t')) = t'$$
The function \( \text{TIME} \) and the set \( \text{Thrd}_{\text{exe}} \)

\[ \text{TIME} : (\text{Conf} \times \text{Thrd}) \rightarrow \text{Time} \] is assumed to be provided by a timing model of the underlying architecture. \( \text{TIME}(c, T) \) should return a relative, discrete execution time for the active statement of thread \( T \), i.e., \( \text{STM}(T, pc_T) \), based on the current system state as given by \( c \). Note that as of now, \( \text{TIME} \) is defined completely without regard to the state of the underlying architecture (e.g., the hardware). If such information is desirable or even necessary in the definition of \( \text{TIME} \), it could easily be added to the current definition of configurations.

Given Assumption 4.1, time is guaranteed to move forward when using \( prg \) for a given configuration (Lemma 4.2). Assumption 4.3 gives that a thread that is waiting to acquire some lock cannot spin an infinite number of times in zero amount of time. Note that the definition of \( \text{TIME} \) does not lie within the scope of this thesis.

Assumption 4.1 (\( \text{TIME} \) is non-negative):
It is assumed that \( \forall c \in \text{Conf} : \forall T \in \text{Thrd} : 0 \leq \text{TIME}(c, T). \)

Lemma 4.2 (Time only moves forward):
Given that the two configurations \( c@\langle[T, pc_T, z_T, t^a_T]|_{T \in \text{Thrd}}, \mathbb{x}, \emptyset \rangle \in \text{Conf} \) and \( c'@\langle[T, pc'_T, z'_T, t'^a_T]|_{T \in \text{Thrd}}, \mathbb{x}', \emptyset' \rangle \in \text{Conf} \) are such that \( c \xrightarrow{prg} c' \), \( \forall T \in \text{Thrd} : t^a_T \leq t'^a_T. \)

PROOF. Assume that \( c@\langle[T, pc_T, z_T, t^a_T]|_{T \in \text{Thrd}}, \mathbb{x}, \emptyset \rangle \in \text{Conf} \) and \( c'@\langle[T, pc'_T, z'_T, t'^a_T]|_{T \in \text{Thrd}}, \mathbb{x}', \emptyset' \rangle \in \text{Conf} \) are such that \( c \xrightarrow{prg} c' \). From Table 4.5, it is apparent that there are two possibilities for the value of \( t'^a_T \).

If \( t^a_T + \text{TIME}(c, T) = \min(\{t^a_T + \text{TIME}(c, T') | T' \in \text{Thrd}\}) \land \text{STM}(T, pc_T) \neq \lceil \text{halt} \rceil | pc_T \), then \( t'^a_T = t^a_T + \text{TIME}(c, T) \). Thus, \( t'^a_T \geq t^a_T \) (Assumption 4.1).

If \( t^a_T + \text{TIME}(c, T) \neq \min(\{t^a_T + \text{TIME}(c, T') | T' \in \text{Thrd}\}) \lor \text{STM}(T, pc_T) = \lceil \text{halt} \rceil | pc_T \), then \( t'^a_T = t^a_T \).

Thus, it must be that \( \forall T \in \text{Thrd} : t^a_T \leq t'^a_T. \)

Assumption 4.3 (\( \text{TIME} \) is non-zero when spin-locking):
It is assumed that \( \forall c@\langle[T, pc_T, z_T, t^a_T]|_{T \in \text{Thrd}}, \mathbb{x}, \emptyset \rangle \in \text{Conf} : \forall T \in \text{Thrd} : ((\exists \text{lck} \in \text{Lck} : \text{STM}(T, pc_T) = \lceil \text{lock lck} \rceil | pc_T \land \text{OWN}(\text{lck}\#T) \not\in \{\text{thrd}\}_{\text{exe}}, T) \Rightarrow 0 < \text{TIME}(c, T) ). \)

The set of threads to execute, \( \text{Thrd}_{\text{exe}} \) (i.e., the threads whose active statements will take effect) on a transition from a given configuration, \( c@\langle[T, pc_T, z_T, t^a_T]|_{T \in \text{Thrd}}, \mathbb{x}, \emptyset \rangle \in \text{Conf} \), is determined based on \( t^a_T \) and \( \text{TIME}(c, T) \).
for each thread $T \in \text{Thrd}$. It simply consists of the threads that will execute their active statements at the nearest point in time, denoted by $t$ in Table 4.5. Only threads in $\text{Thrd}_{exe}$ affect the system state upon a transition between configurations.

An illustration of how $\text{Thrd}_{exe}$ is determined is given in Figure 4.8. For $c_1$ in Figure 4.8a, $t = t_{T_1}^{a} + \text{TIME}(c_1, T_2) = t_{T_2}^{a} + \text{TIME}(c_1, T_3) = 6$ and $t_{T_3}^{a} + \text{TIME}(c_1, T_1) = 10$. Thus, $\text{Thrd}_{exe} = \{T_2, T_3\}$ and $t_{T_2}^{a} = t_{T_1}^{a} = 6$, while $t_{T_1}^{a} = t_{T_3}^{a} = 7$. For $c_2$ in Figure 4.8b, $\text{Thrd}_{exe}$ is determined in a similar manner (note that $c_1 \xrightarrow{prg} c_2$).

The statements halt and skip

As previously discussed, halt stops the execution of a thread and skip is a no-operation. This is implemented by letting the semantic rule for halt re-
turn the thread’s input state without modifying it, which means that the issuing thread will still be executing the same halt-statement in the next iterative step; thus the thread halts. Note that threads issuing a halt-statement are not included in \texttt{Thrd}_{\text{exe}}, however. The rule for the skip-statement only increments the thread’s program counter, \( pc \), and thus advances the thread to execute its subsequent statement in the next iterative step.

**Assignment of register values**

The statement \( r := a \) returns a register state in which the register \( r \) has the value of the arithmetic expression \( a \). The value of \( a \) is, in the general case, dependent on the register values in the input register state and is determined using the function \( \mathcal{A} : \mathsf{Aexp} \to (\mathsf{Reg} \to \mathsf{Val}) \to \mathsf{Val} \). \( \mathcal{A} \) evaluates arithmetic expressions based on a given register state as defined in Table 4.3. It should be noted that the calculations \( \infty - \infty, \infty / \infty, 0 \times \infty \) and \( 0/0 \) are undefined and hence not supported by \( \mathcal{A} \). The operation \( x/0 \), where \( x \in \mathsf{Val} \), results in \( \infty \) if \( 0 < x \) and \( -\infty \) if \( x < 0 \).

**The conditional statement if**

The statement \( \text{if } b \text{ goto } l \) performs conditional branching. If the boolean expression \( b \) evaluates to \texttt{true}, the issuing thread’s \( pc \) is set to \( l \). If \( b \) evaluates to \texttt{false}, then if acts like the skip-statement. The value of \( b \) is, in the general case, dependent on the register values in the input register state and is determined using the function \( \mathcal{B} : \mathsf{Bexp} \to (\mathsf{Reg} \to \mathsf{Val}) \to \mathsf{Bool} \). \( \mathcal{B} \) evaluates boolean expressions based on a given register state as defined in Table 4.4.

**The statements store and load**

To achieve a high precision in the analysis (see Chapters 5 and 6), the abstraction of the state for variables will need to save write history; i.e., what abstract writes (each write being a pair of value and time) have been performed by each thread on each variable (see Chapter 5). Therefore, to derive a Galois connection between the concrete and abstract domains for variable states, the concrete state, \( x \), has to be defined accordingly. This is why the definition of \( x \) might look a bit peculiar at first glance. In the concrete semantics, only one single write is saved for each variable, though, since this is all the information that is needed in the concrete case. If several threads write to a variable (using the \texttt{store}-statement) at the same time, there is a race on that variable and the resulting state will contain one of the writes; i.e., one of the threads will win
the race. The winning thread is non-deterministically chosen from one of the threads writing the variable at the given point in time (see the definition of $\mathcal{R}$ in Table 4.5).

load is defined to put the value of the saved write (or rather, the value of one of the saved writes in the general case) in the given register (see the definition of $\mathcal{A}$ in Table 4.2).

**The statements lock and unlock**

As the observant reader might have noticed already, the only information that should be needed in order to successfully express the semantic behavior of locks is what thread is currently assigned (i.e., is currently the owner of) a lock. This is truly the case. However, the extra information given in the concrete state for locks, $\mathcal{A}$, will ease the deriving of an approximation of the concrete semantics (see Chapter 5) and achieve a high precision in the analysis (see Chapter 6). This is why the definition of $\mathcal{A}$ might look a bit peculiar at first glance. Here, the state of locks, i.e., locked or unlocked, is only used to increase the readability of the rules in Tables 4.2 and 4.5. Note that a consequence of this is that, in a state of locks, i.e., $\mathcal{A}$, the given valid configuration, $\mathcal{C}$, will ease the deriving of an approximation of the concrete semantics (see Chapter 5) and achieve a high precision in the analysis (see Chapter 6). This is why the definition of $\mathcal{A}$ might look a bit peculiar at first glance. Here, the state of locks, i.e., locked or unlocked, is only used to increase the readability of the rules in Tables 4.2 and 4.5. Note that a consequence of this is that, in a state of locks, i.e., $\mathcal{A}$, the given valid configuration, $\mathcal{C}$, is only applied to axiom input configuration $\mathcal{C}$ and the thread’s pc.

Only one single thread can be assigned a given lock at any point in time.

Note that $\mathcal{A}''$ is only an intermediate lock state, whose sole purpose is to assign the ownership of a free lock to one of the threads executing lock on it, if any. $\mathcal{A}''$ is only used as an input to the $\rightarrow$-relation. The resulting lock state, $\mathcal{A}'$, is in turn based on the output lock states from $\rightarrow$, $\mathcal{A}'$, given that T is the owner of the considered lock in $\mathcal{A}''$.

The unlock-statement has the same behavior as the skip-statement if the given lock is not assigned to the issuing thread. If the issuing thread is assigned (and thus, has acquired; Definition 4.4 and Lemma 4.5) the given lock, unlock is defined to release the lock so that it can be re-assigned in the next iterative
step to some thread, if any, issuing `lock` on it. Note that a thread can repeatedly acquire a lock that is assigned to (and thus, taken by) the thread, without first releasing it.

**Definition 4.4 (Valid concrete configuration):**
A concrete configuration, \( c @ \langle [T, pc_T, t_T^2]_{T \in \text{Thrd}}, \emptyset \rangle \in \text{Conf} \), is valid with respect to the lock state, \( lck \), iff

\[
\forall lck \in \text{Lck} : ((\text{STT}(lck) = \text{locked} \iff \text{OWN}(lck) \neq \bot_{\text{thrd}}) \land \\
(\text{STT}(lck) = \text{unlocked} \iff \text{OWN}(lck) = \bot_{\text{thrd}}) \land \\
\forall T \in \text{Thrd} : \text{REL}(lck) \leq t_T^q + \text{TIME}(c, T)) \]

\( \square \)

**Lemma 4.5 (\( \xrightarrow{\text{pro}} \) preserves lock state validity):**
Given that the configuration \( c @ \langle [T, pc_T, t_T^2]_{T \in \text{Thrd}}, \emptyset \rangle \in \text{Conf} \) is valid (cf. Definition 4.4), then so is \( c' @ \langle [T, pc_T', t_T'^q]_{T \in \text{Thrd}}, \emptyset \rangle \in \text{Conf} \), whenever \( c \xrightarrow{\text{pro}} c' \).

**PROOF.** From Table 4.2, it is apparent that the possible axiom output lock states, called \( \emptyset_T \) in Table 4.5, given an input lock state, called \( \emptyset'' \) in Table 4.5, are

1. \( \emptyset''|lck \mapsto (\text{locked}, T, \text{DL}(\emptyset'' lck), \text{POWN}(\emptyset'' lck), \text{REL}(\emptyset'' lck)) \), whenever \( \text{STM}(T, pc_T) = [\text{lock } lck]^{pc_T} \land \text{OWN}(\emptyset'' lck) = T \),

2. \( \emptyset''|lck \mapsto (\text{unlocked}, \bot_{\text{thrd}}, \text{DL}(\emptyset'' lck), T, t_T'^q) \), whenever \( \text{STM}(T, pc_T) = [\text{unlock } lck]^{pc_T} \land \text{OWN}(\emptyset'' lck) = T \), and

3. \( \emptyset'' \), otherwise.

Assume that the configurations \( c @ \langle [T, pc_T, t_T^2]_{T \in \text{Thrd}}, \emptyset \rangle \in \text{Conf} \) and \( c' @ \langle [T, pc_T', t_T'^q]_{T \in \text{Thrd}}, \emptyset \rangle \in \text{Conf} \) are such that \( c \) is valid and \( c \xrightarrow{\text{pro}} c' \).

From Table 4.5, it is apparent that \( \xrightarrow{\text{ax}} \) is only applied to axiom input configurations in which \( \emptyset'' \) is such that

1. \( \emptyset'' lck = \emptyset lck \), or

2. \( \emptyset'' lck = (\text{unlocked}, T, t_T'^q, \text{POWN}(\emptyset lck), \text{REL}(\emptyset lck)) \).

For the first case, it is easy to see that all the three possible output lock states result in a valid configuration since \( c \) is valid.
The second case only occurs when $\exists T' \in \text{Thrd}_{\text{exe}} : (\text{STM}(T', pc_T) = [\text{lock}\ lck]^{pc_{T'}} \land \text{OWN}(\top lck) = \bot_{\text{thrd}})$. Note that the assigned owner, $T \in \text{Thrd}_{\text{exe}}$, is one of the threads executing $\text{lock}\ lck$. For thread $T$, the output lock state is $\top lck \mapsto (\text{locked}, T, DL(\top lck), \text{POWN}(\top lck), \text{REL}(\top lck))$, since $\text{STM}(T, pc_T) = [\text{lock}\ lck]^{pc_T} \land \text{OWN}(\top lck) = T$. Hence, $\top lck = \top lck \mapsto (\text{locked}, T, t_{T'}^{pl}, \text{POWN}(\top lck), \text{REL}(\top lck))$.

Since time moves forward for each thread (Lemma 4.2), it is easy to see that $c'$ is valid.

Lemma 4.6 gives some important properties of the “intermediate” lock state, $\top lck$, defined in Table 4.5, which is used as a means of assigning a lock to a specific thread. These properties will be used when proving the correctness of the abstract semantics in Tables 5.12 and 5.13, presented on pages 111 and 115, respectively. To increase the readability of the upcoming proofs in Chapter 5, $\prec$ is used here instead of $\geq$. This is to better correlate with the $\preceq_t$ operator, defined in Definition 5.14.

Lemma 4.6 (Properties of $\top lck$):

If for some valid configuration, $c @ ([T, pc_T, x_T, t_{T'}^p]_{T \in \text{Thrd}_{\text{exe}}, x, \top lck}) \in \text{Conf}$, and lock, $\top lck \in \text{Lck}$, $\text{OWN}(\top lck) = \bot_{\text{thrd}} \land \exists T' \in \{T \in \text{Thrd}_{\text{exe}} \mid \text{STM}(T, pc_T) = [\text{lock}\ lck]^{pc_T} \} : \text{OWN}(\top lck) = T'$, where $\top lck$ and $\text{Thrd}_{\text{exe}}$ are as defined in Table 4.5, then

1. $\text{STT}(\top lck) = \text{unlocked},$
2. $\text{DL}(\top lck) \not\prec_{\text{prg}} t_{T'}^{pl},$ and
3. $t_{T'}^{pl} \not\prec_{\text{prg}} \text{REL}(\top lck).$

Proof. For this proof, each of the properties above will be shown based on the definition of $\rightarrow_{ax}$ and $\rightarrow_{\text{prg}}$, defined in Tables 4.2 and 4.5, respectively.

Assume that for the valid configuration $c @ ([T, pc_T, x_T, t_{T'}^p]_{T \in \text{Thrd}_{\text{exe}}, x, \top lck}) \in \text{Conf}$ (cf. Definition 4.4) and some lock, $\top lck \in \text{Lck}$, $\text{OWN}(\top lck) = \bot_{\text{thrd}} \land \exists T' \in \{T \in \text{Thrd}_{\text{exe}} \mid \text{STM}(T, pc_T) = [\text{lock}\ lck]^{pc_T} \} : \text{OWN}(\top lck) = T'$.

1 follows directly from the definition of $\top lck$ in Table 4.5.

Table 4.5 also gives that $\text{DL}(\top lck) = t$ and that $t_{T'}^{pl} = t$, since $T' \in \text{Thrd}_{\text{exe}} \land \text{STM}(T, pc_T) = [\text{lock}\ lck]^{pc_T}$. Thus, $\text{DL}(\top lck) = t_{T'}^{pl}$, and hence, 2 has been shown.

For 3, Assumption 4.1 gives that time moves forward when using $\rightarrow_{\text{prg}}$ (Lemma 4.2). Thus, it must be that $t_{T'}^{pl} \geq \text{REL}(\top lck) = \text{REL}(\top lck)$ (cf. Table 4.5), which concludes the proof.
Initial configurations

For the above (and the following) reasoning to hold, any initial configuration must be valid (cf. Definition 4.4). If this is not the case, Lemmas 4.5 and 4.6 do not hold.

4.3 Collecting Semantics

This section defines the collecting semantics, $\mathcal{C}(C)$, of a program [91]; i.e., the set of all possible semantic configurations given an initial set of valid configurations, $C$, (cf. Definition 4.7).

**Definition 4.7 (Collecting semantics):**

The collecting semantics, $\mathcal{C}(C)$, of an initial set of valid configurations, $C$, is defined as [41]:

$$
\mathcal{C}(C) = \bigcup_{i \geq 0} C^i \quad \text{where} \quad \begin{cases} 
C^0 = C \\
C^{i+1} = \{ c' \in \text{Conf} \mid \exists c \in C^i : c \xrightarrow{\text{prog}} c' \} 
\end{cases}
$$

As can be seen, the collecting semantics will include all possible configurations that a given initial configuration can ever reach. Note that the collecting semantics might be of infinite size, for example in the case of a nonterminating program; i.e., the accumulated time, $t^i_T$, for some thread, $T \in \text{Thrd}$, could increase indefinitely.
Abstractly Interpreting PPL

In this chapter, the semantics of PPL, defined in Chapter 4, will be abstracted. First it must be decided what parts of the system state to interpret in an abstract way. The names of the abstract instances will be the name of the concrete instances crowned with ‘˜’ (tilde).

To allow for the timing model of the underlying architecture to be approximated as well, Time will be abstracted using the interval domain, i.e., $\text{Time} \sim \text{Intv}$. This approach is also taken by Chattopadhyay et al. [20] to approximate the execution time of pipeline stages in order to deal with timing anomalies in multi-core platforms.

Val will also be abstracted using intervals, i.e., $\text{Val} \sim \text{Intv}$, to allow for an efficient handling of data flow (note that many other domains could be used as well). Since Thrd, Lbl, Var, Reg, Lck, $\text{Aexp}$ and $\text{Bexp}$ are defined by the software, where the elements of Thrd, Lbl, Var, Reg and Lck are identifiers and the elements of $\text{Aexp}$ and $\text{Bexp}$ are the defined arithmetical and boolean expressions, respectively, it does not make much sense to abstract them for the defined analysis (see Chapter 6). And, since Lckstt is comparable to Bool, an abstraction of it would most probably not be very beneficial. The states affected by the abstractions of Time and Val are $t$, $a$, and $c$. The abstraction of these will be referred to as $\tilde{t}$, $\tilde{a}$, and $\tilde{c}$, respectively.

Note that since $\text{Time} \sim \text{Intv}$ and $\text{Val} \sim \text{Intv}$, the abstraction and concretization functions, the partial order, least upper and greatest lower bound operators, and the top and bottom elements for these domains are inherited from the Intv domain; i.e., $\alpha_t = \alpha_{val} = \alpha_{int}$ etc.
Chapter 5

Abstractly Interpreting PPL

In this chapter, the semantics of PPL, defined in Chapter 4, will be abstracted. First it must be decided what parts of the system state to interpret in an abstract way. The names of the abstract instances will be the name of the concrete instances crowned with ‘˜’ (tilde).

To allow for the timing model of the underlying architecture to be approximated as well, Time will be abstracted using the interval domain, i.e., Tiţne = Intv. This approach is also taken by Chattopadhyay et al. [20] to approximate the execution time of pipeline stages in order to deal with timing anomalies in multi-core platforms. Val will also be abstracted using intervals, i.e., Văl = Intv, to allow for an efficient handling of data flow (note that many other domains could be used as well). Since Thrd, Lbl, Var, Reg, Lck, Aexp and Bexp are defined by the software, where the elements of Thrd, Lbl, Var, Reg and Lck are identifiers and the elements of Aexp and Bexp are the defined arithmetical and boolean expressions, respectively, it does not make much sense to abstract them for the defined analysis (see Chapter 6). And, since Lcksat is comparable to Bool, an abstraction of it would most probably not be very beneficial. The states affected by the abstractions of Time and Val are r, x, tδ, I and c. The abstraction of these will be referred to as ˜r, ˜x, ˜tδ, ˜I and ˜c, respectively.

Note that since Tiţne = Intv and Văl = Intv, the abstraction and concretization functions, the partial order, least upper and greatest lower bound operators, and the top and bottom elements for these domains are inherited from the Intv domain; i.e., αt = αval = αint etc.

The beginning of this chapter (Sections 5.1–5.6) handles all states and op-
operators etc. that must be abstracted in order to define abstract configurations (cf. Section 5.7) and the abstract semantics (cf. Section 5.8).

**Note.** A summary of the notation and nomenclature used in this thesis can be found in Appendix A.

### 5.1 Arithmetical Operators for Intervals

Since values and time are abstracted using the interval domain, the operators of PPL must be extended to act on intervals. This is done in Table 5.1; note that the undefined calculations $\infty - \infty$, $\infty/\infty$, $0 \times \infty$ and $0/0$ are never performed in the definitions, and the resulting interval where such a calculation would have been performed is always $[-\infty, \infty]$, i.e., the top element in the interval domain.

**Note.** In the following, $\downarrow_t$ and $\downarrow_{val}$ both refer to $+_{int}$, and similarly for the rest of the operators.

### 5.2 Abstract Register States

Using Theorems 3.24 and 3.39, it is easy to see that there is indeed a Galois connection, $\langle \alpha_{reg}, \gamma_{reg} \rangle$, between the concrete domain $\mathcal{P}(\text{Reg} \rightarrow \text{Val})$ and the abstract domain $(\text{Reg} \rightarrow \mathcal{V}al) \cup \{\mathcal{I}_{reg}, \mathcal{T}_{reg}\}$ (Theorem 5.6). The concretization function, $\gamma_{reg}$, partial order, $\sqsubseteq_{reg}$, greatest lower bound, $\sqcap_{reg}$, least upper bound, $\sqcup_{reg}$, and abstraction function, $\alpha_{reg}$, are given by Definitions 5.1, 5.2, 5.3, 5.4 and 5.5, respectively. Note that these are standard definitions for abstract states with intervals as abstract values [41]. $\mathcal{I}$ is the bottom element, $\mathcal{I}_{reg}$, if $\forall r \in \text{Reg}: \mathcal{F}_r = \mathcal{I}_{val}$; i.e., if $\mathcal{F}$ maps all registers to $\mathcal{I}_{val}$. The top element, $\mathcal{T}_{reg}$, corresponds to an abstract mapping for which all registers map to $\mathcal{T}_{val}$.

**Definition 5.1 (Concretization of an abstract register state):**

$$
\gamma_{reg}(\mathcal{F}) = \begin{cases} 
\text{Reg} \rightarrow \text{Val} & \text{if } \mathcal{F} = \mathcal{T}_{reg} \\
\emptyset & \text{if } \mathcal{F} = \mathcal{I}_{reg} \\
\{r \in \text{Reg} \rightarrow \text{Val} \mid \forall r \in \text{Reg} : (\mathcal{F}_r) \in \gamma_{val}(\mathcal{F}_r)\} & \text{otherwise}
\end{cases}
$$
Using Theorems 3.24 and 3.39, it is easy to see that there is indeed a Galois connection, using Theorems 3.24 and 3.39, it is easy to see that there is indeed a Galois connection.

Since values and time are abstracted using the interval domain, the operators of the undefined calculations PPL must be extended to act on intervals. This is done in Table 5.1; note that 5.1 Arithmetical Operators for Intervals

\[\begin{align*}
\neg \& \neg = \begin{cases}
[l_1 + l_2, u_1 + u_2] & \text{if } -\infty < l_1, l_2, u_1, u_2 < \infty \\
[l_1 + l_2, \infty] & \text{if } (u_1 = \infty \lor l_2 = \infty) \land \\
[\neg((l_1 = -\infty \land l_2 = \infty) \lor (l_1 = \infty \land l_2 = -\infty))] & \text{if } (l_1 = -\infty \lor l_2 = -\infty) \land \\
(-\infty, u_1 + u_2] & \text{if } (u_1 = -\infty \land u_2 = \infty) \lor \\
(-\infty, \infty) & \text{otherwise}
\end{cases}
\end{align*}\]

\[\begin{align*}
[l_1 - u_2, u_1 - l_2] & \text{if } -\infty < l_1, l_2, u_1, u_2 < \infty \\
[l_1 - u_2, \infty] & \text{if } (u_1 = \infty \lor l_2 = -\infty) \land \\
[\neg((l_1 = \infty \land u_2 = \infty) \lor (l_1 = -\infty \land l_2 = -\infty))] & \text{if } (l_1 = -\infty \lor u_2 = \infty) \land \\
(-\infty, u_1 - l_2] & \text{if } (u_1 = \infty \land l_2 = \infty) \lor \\
(-\infty, \infty) & \text{otherwise}
\end{align*}\]

\[\begin{align*}
[l_1 \cdot u_1] & \text{if } (u_1 < 0 \lor u_2 < 0) \land \\
[\min(V), \max(V)] & \text{if } (u_1 < 0 \lor u_2 < 0) \land \\
[l_1 > 0 \lor -\infty < l_1)] \lor \\
((u_1 < 0 \lor -\infty < l_2) \land \\
(u_2 < 0 \lor -\infty < l_1) \lor \\
((l_1 > 0 \lor u_2 < 0) \land \\
(l_2 > 0 \lor u_1 < \infty) \lor \\
((l_1 > 0 \lor -\infty < l_2) \land \\
(u_2 < 0 \lor u_1 < \infty) \lor \\
(-\infty < l_1, l_2, u_1, u_2 < \infty)
\end{cases}\]

where \( V = \{l_1l_2, l_1u_2, u_1l_2, u_1u_2\} \)

\[\begin{align*}
[l_1 + l_2, \infty] & \text{if } -\infty < l_1, l_2, \infty < \infty \\
[l_1 + \infty, \infty] & \text{if } (l_1 = \infty) \land \\
[l_1 + \neg((l_1 = -\infty \land l_2 = \infty) \lor (l_1 = \infty \land l_2 = -\infty))] & \text{if } (l_1 = -\infty \lor l_2 = -\infty) \land \\
(-\infty, \infty) & \text{otherwise}
\end{cases}\]

\[\begin{align*}
[l_1 - u_2, \infty] & \text{if } -\infty < l_1, l_2, \infty < \infty \\
[l_1 - \infty, \infty] & \text{if } (l_1 = -\infty) \land \\
[l_1 - \neg((l_1 = \infty \land u_2 = \infty) \lor (l_1 = -\infty \land l_2 = -\infty))] & \text{if } (l_1 = -\infty \lor u_2 = \infty) \land \\
(-\infty, \infty) & \text{otherwise}
\end{cases}\]

\[\begin{align*}
[l_1 \cdot \neg((l_1 = -\infty \land l_2 = \infty) \lor (l_1 = \infty \land l_2 = -\infty))] & \text{if } (l_1 = -\infty \lor l_2 = -\infty) \land \\
(-\infty, \infty) & \text{otherwise}
\end{cases}\]

Table 5.1: PPL operators defined for interval arguments.
Table 5.1: *Cont.* PPL operators defined for interval arguments.

<table>
<thead>
<tr>
<th>Interval Expression</th>
<th>Description</th>
</tr>
</thead>
</table>
| \([l_1, u_1] \setminus [l_2, u_2]\) | \[\{[\min(V)], [\max(V)]\} \quad \text{if } (-\infty < l_2 \land u_2 < 0) \lor (0 < l_2 \land u_2 < \infty)\]  
|                     | \([-\infty, -l_1]\) \quad \text{if } l_2 \leq 0 \land u_1 < 0 \]  
|                     | \([-u_1, \infty]\) \quad \text{if } l_2 \leq 0 \land u_2 < 0 \]  
|                     | \([-\infty, \infty]\) \quad \text{otherwise} \]  

Definition 5.2 (Partial order for abstract register states):

\[
\{ \hat{x} \sqsubseteq_{\text{reg}} \hat{r}, \hat{r} \} \quad \text{if } r \sqsubseteq \hat{r} \]

Definition 5.3 (Greatest lower bound of abstract register states):

\[
\{ \hat{\top}_{\text{reg}} \cap_{\text{reg}} \hat{r} = \hat{r}, \quad \hat{\bot}_{\text{reg}} \cap_{\text{reg}} \hat{r} = \hat{r}, \quad \hat{\bot}_{\text{reg}} \cap_{\text{reg}} \hat{r} = \hat{r} \}
\]

Definition 5.4 (Least upper bound of abstract register states):

\[
\{ \hat{\top}_{\text{reg}} \cup_{\text{reg}} \hat{r} = \hat{r}, \quad \hat{\bot}_{\text{reg}} \cup_{\text{reg}} \hat{r} = \hat{r}, \quad \hat{\bot}_{\text{reg}} \cup_{\text{reg}} \hat{r} = \hat{r} \}
\]

Definition 5.5 (Abstraction of a set of register states):

\[
\alpha_{\text{reg}}(\mathbb{R}) = \begin{cases} 
\hat{\top}_{\text{reg}} & \text{if } \mathbb{R} = \text{Reg} \rightarrow \text{Val} \\
\hat{\bot}_{\text{reg}} & \text{if } \mathbb{R} = \emptyset \\
\lambda r \in \text{Reg}. \alpha_{\text{val}}(\{\hat{r} \mid \hat{r} \in \mathbb{R}\}) & \text{otherwise}
\end{cases}
\]
Theorem 5.6 (Galois connection – Register states):
\[ \langle \alpha_{\text{val}}, \gamma_{\text{reg}} \rangle, \] where \( \gamma_{\text{reg}} \) and \( \alpha_{\text{reg}} \) are defined as in Definitions 5.1 and 5.5, respectively, is a Galois connection.

PROOF. Since \( \alpha_{\text{val}} = \alpha_{\text{int}} \) and \( \gamma_{\text{val}} = \gamma_{\text{int}} \), \( \langle \alpha_{\text{val}}, \gamma_{\text{val}} \rangle \) is a Galois insertion between \( \mathcal{P}(\text{Val}) \) and \( \tilde{\text{Val}} \) (Theorem 3.39).

By Theorem 3.24, \( \langle \alpha_{\text{reg}} : \mathcal{P}(\text{Reg} \rightarrow \text{Val}) \rightarrow (\text{Reg} \rightarrow \tilde{\text{Val}}) \cup \{ \tilde{\text{reg}}, \tilde{\text{reg}} \}, \gamma_{\text{reg}} : (\text{Reg} \rightarrow \tilde{\text{Val}}) \cup \{ \tilde{\text{reg}}, \tilde{\text{reg}} \} \rightarrow \mathcal{P}(\text{Reg} \rightarrow \text{Val}) \)\), where \( \gamma_{\text{reg}} \) and \( \alpha_{\text{reg}} \) are as presented in Definitions 5.1 and 5.5, respectively, is a Galois connection. \( \square \)

5.3 Abstract Evaluation of Arithmetic Expressions

The function evaluating arithmetic expressions, \( \mathcal{A} \), must be abstracted since values and register states are abstracted. The abstraction will be \( \mathcal{A} : \text{Aexp} \rightarrow (\text{Reg} \rightarrow \tilde{\text{Val}}) \rightarrow \tilde{\text{Val}} \), which is equivalent to \( \mathcal{A} : \text{Aexp} \rightarrow (\text{Reg} \rightarrow \text{Intv}) \rightarrow \text{Intv} \), and can be derived using Definition 3.11 to induce \( \mathcal{A} \). To do this, \( \mathcal{A} \) must first be lifted to sets of concrete register mappings:
\[
\mathcal{A}_{\mathcal{P}}[a] = \{ \mathcal{A}[[a]] \mid a \in \mathbb{R} \}
\]

The abstract evaluation function is then given by Definition 5.7.

Definition 5.7 (Abstract evaluation of arithmetic expressions):
\[
\mathcal{A}[a] = \alpha_{\text{val}} \circ \mathcal{A}_{\mathcal{P}}[a] \circ \gamma_{\text{reg}} = \alpha_{\text{val}} \circ \lambda \mathbb{R} \{ \mathcal{A}[[a]] \mid a \in \mathbb{R} \} \circ \gamma_{\text{reg}}
\]

The details of this function can be found in Table 5.2. Note that this is a standard definition for abstract evaluation of arithmetic expressions with intervals as abstract values [41, 91].

5.4 Boolean Restriction for Intervals

The function \( \mathcal{B} : \text{Bexp} \rightarrow (\text{Reg} \rightarrow \tilde{\text{Val}}) \rightarrow (\text{Reg} \rightarrow \tilde{\text{Val}}) \) is ideally defined as \( \mathcal{B}_{\text{ind}} \), given in Definition 5.8, and is applied when the if-statement is evaluated to restrict the register states for the subsequent analysis [27, 41].

Definition 5.8 (Boolean restriction):
\[
\mathcal{B}_{\text{ind}}[b] = \alpha_{\text{reg}}(\{ r \in \gamma_{\text{reg}}(b) \mid \mathcal{B}[b](r) \})
\]
Table 5.2: The abstract function evaluating arithmetic expressions.

\[
\begin{align*}
\mathcal{A}[n] & \tilde{r} = \alpha_{val}(\{n\}) \\
\mathcal{A}[r] & \tilde{r} = \tilde{r} \\
\mathcal{A}[a_1 + a_2] & \tilde{r} = \mathcal{A}[a_1] \tilde{r} +_{\text{val}} \mathcal{A}[a_2] \tilde{r} \\
\mathcal{A}[a_1 - a_2] & \tilde{r} = \mathcal{A}[a_1] \tilde{r} -_{\text{val}} \mathcal{A}[a_2] \tilde{r} \\
\mathcal{A}[a_1 \times a_2] & \tilde{r} = \mathcal{A}[a_1] \tilde{r} \times_{\text{val}} \mathcal{A}[a_2] \tilde{r} \\
\mathcal{A}[a_1 / a_2] & \tilde{r} = \mathcal{A}[a_1] \tilde{r} /_{\text{val}} \mathcal{A}[a_2] \tilde{r}
\end{align*}
\]

\(\mathcal{B}_{\tilde{r}}^{\text{ind}}\) is safely induced from \(\mathcal{B}\), using Definition 3.11, so that the concretization of \(\mathcal{B}_{\tilde{r}}^{\text{ind}}[b] \tilde{r}\), where \(b \in \text{Bexp}\), always contains (at least) the concrete register states, derived from \(\tilde{r} \in (\text{Reg} \rightarrow \text{Văl})\), for which \(b\) evaluates to \text{true} [41]. For example, evaluating the statement \(\text{if } b \text{ goto } l\) in some register state, \(\tilde{r}_T\), where \(l \in \text{Lbl}_T\) and \(T \in \text{Thrd}\), can find \(b\) to be \text{true} and/or \text{false} depending on the values of the registers included in the expression, \(b\), for the given register state; the first case occurs when \(\mathcal{B}_{\tilde{r}}^{\text{ind}}[b] \tilde{r}_T \neq \bot_{\text{reg}}\) and the latter when \(\mathcal{B}_{\tilde{r}}^{\text{ind}}[b] \tilde{r}_T = \bot_{\text{reg}}\). Note that both cases can occur for a given pair of \(b\) and \(\tilde{r}_T\). Also note that Definition 5.8 is generic and does not necessarily target the interval domain specifically. The over-approximation \(\mathcal{B}_{\tilde{r}}\) of \(\mathcal{B}_{\tilde{r}}^{\text{ind}}\) for the interval domain will be used in the abstract axiom transition rules (see Table 5.12 on page 111).

Boolean restriction of a register state based on \(b \in \text{Bexp}\) is practically performed by recursively restricting the register state for each subexpression of \(b\). The restricted register state for a subexpression is then further restricted when considering the parent expression for that subexpression. This process is continued until a restricted register state for \(b\) itself is obtained. The details of this process, over-approximately instantiated for the interval domain, can be found in Table 5.3. Note that it will not be proven that \(\mathcal{B}_{\tilde{r}}\) as defined in Table 5.3 is strictly induced from \(\mathcal{B}\) (i.e., is as tight as \(\mathcal{B}_{\tilde{r}}^{\text{ind}}\)), a safe approximation will rather be mathematically derived.

The function \(\mathcal{A}_{\tilde{r}} : \text{Aexp} \rightarrow \text{Intv} \rightarrow (\text{Reg} \rightarrow \text{Văl}) \rightarrow (\text{Reg} \rightarrow \text{Văl})\), i.e., the notation \(\mathcal{A}_{\tilde{r}}[a](i_{\tilde{r}}) \tilde{r}\), where \(a \in \text{Aexp}, i_{\tilde{r}} \in \text{Intv} \text{ and } \tilde{r} \in (\text{Reg} \rightarrow \text{Văl})\), is used when restricting a register state based on arithmetic expressions. The intuition
Table 5.3: Boolean restriction for intervals.

\[
\begin{align*}
\mathcal{B}_\text{reg}[b] &\cdot \bar{\text{reg}} = \bar{\text{reg}} \\
\mathcal{B}_\text{reg}[^\text{true}] &\cdot \bar{\text{reg}} = \bar{\text{reg}} \\
\mathcal{B}_\text{reg}[^\text{false}] &\cdot \bar{\text{reg}} = \bar{\text{reg}} \\
\mathcal{B}_\text{reg}[b_1 \& b_2] &\cdot \bar{\text{reg}} = (\mathcal{B}_\text{reg}[b_1] \cdot \bar{\text{reg}}) \cap (\mathcal{B}_\text{reg}[b_2] \cdot \bar{\text{reg}}) \\
\mathcal{B}_\text{reg}[a_1 == a_2] &\cdot \bar{\text{reg}} = (\mathcal{A}_\text{reg}[a_1] \cap (\mathcal{A}_\text{reg}[a_2] \cdot \bar{\text{reg}})) \cap (\mathcal{A}_\text{reg}[a_1] \cap (\mathcal{A}_\text{reg}[a_2] \cdot \bar{\text{reg}})) \\
\mathcal{B}_\text{reg}[a_1 <= a_2] &\cdot \bar{\text{reg}} = (\mathcal{A}_\text{reg}[a_1] \cap (\mathcal{A}_\text{reg}[a_2] \cdot \bar{\text{reg}})) \cap (\mathcal{A}_\text{reg}[a_1] \cap (\mathcal{A}_\text{reg}[a_2] \cdot \bar{\text{reg}})) \\
\mathcal{B}_\text{reg}[!\text{true}] &\cdot \bar{\text{reg}} = \bar{\text{reg}} \\
\mathcal{B}_\text{reg}[!\text{false}] &\cdot \bar{\text{reg}} = \bar{\text{reg}} \\
\mathcal{B}_\text{reg}[^\text{!}\ b_1] &\cdot \bar{\text{reg}} = (\mathcal{B}_\text{reg}[^\text{!}\ b_1] \cdot \bar{\text{reg}}) \cup (\mathcal{B}_\text{reg}[^\text{!}\ b_2] \cdot \bar{\text{reg}}) \\
\mathcal{B}_\text{reg}[^\text{!}\ a_1 == a_2] &\cdot \bar{\text{reg}} = (\mathcal{A}_\text{reg}[a_1] \cup (\mathcal{A}_\text{reg}[a_2] \cdot \bar{\text{reg}})) \cup (\mathcal{A}_\text{reg}[a_1] \cup (\mathcal{A}_\text{reg}[a_2] \cdot \bar{\text{reg}})) \\
\mathcal{B}_\text{reg}[^\text{!}\ a_1 <= a_2] &\cdot \bar{\text{reg}} = (\mathcal{A}_\text{reg}[a_1] \cup (\mathcal{A}_\text{reg}[a_2] \cdot \bar{\text{reg}})) \cup (\mathcal{A}_\text{reg}[a_1] \cup (\mathcal{A}_\text{reg}[a_2] \cdot \bar{\text{reg}}))
\end{align*}
\]
behind this notation is that all the values for $a$ that must be taken into account are found in the restricting interval $i_{\mathcal{G}}$. This basically renders the equation $a = i_{\mathcal{G}}$ which is recursively solved by considering each subexpression of $a$, and deriving new restricting intervals for these based on what type of subexpression is considered. The axiom cases for the recursion occur when $a = n$ or $a = r$, where $n \in \text{Val}$ and $r \in \text{Reg}$. If the case $a = r$ is encountered, then the interval value of $r$ is trimmed from ranges that lie outside of $i_{\mathcal{G}}$. If the resulting value for $r$ is $\bot_{\text{int}}$, then the equation $a = i_{\mathcal{G}}$ has no solution and hence the restriction process results in $\bot_{\text{reg}}$. The details of this process can be found in Table 5.4. Note that the operators $\otimes_{\text{int}}$, $\ominus_{\text{int}}$ and $\otimes'_{\text{int}}$, as defined in Tables 5.5, 5.6 and 5.7, respectively, are further discussed below.

Solving the equation $a = i_{\mathcal{G}}$ is straightforward for the cases $a = a_1 + a_2$ and $a = a_1 - a_2$ because $+_{\text{int}}$ and $-_{\text{int}}$ can be used when deriving the restricting intervals for the subexpressions, $a_1$ and $a_2$. However, when considering the expressions $a = a_1 \times a_2$ and $a = a_1 / a_2$, special care must be taken as the operators $\times_{\text{int}}$ and $/_{\text{int}}$ cannot be used to solve the equation. Using these operators would for some cases render a restricted register state that does not include all possible concrete cases and thus is not safe. To see this, consider calculating the restricting interval for $r_1$ in $r_1 / r_2 <= 1$, where $r_1 = [-100, 100]$ and $r_2 = [4, 6]$, using the $\times_{\text{int}}$ operator. The restricting interval for the expression $r_1 / r_2$ is obviously $[-\infty, 1]$. This means that the value of $r_1$ would be restricted by $[-\infty, 1] \times_{\text{int}} [4, 6] = [-\infty, 6]$. Thus, the resulting value of $r_1$ would be $[-\infty, 6] \cap_{\text{int}} [-100, 100] = [-100, 6]$. However, also the interval value $[-100, 11]$ for $r_1$ will fulfill the relation $r_1 / r_2 <= 1$ since $[11/6] = 1$ which means that $\times_{\text{int}}$ gives an erroneous result in this case.

For $/_{\text{int}}$, the case is twofold. It can both be unsafe when considering the expression $a = a_1 / a_2$ and unnecessarily un-tight when considering the expression $a = a_1 + a_2$. To see this, first consider calculating the restricting interval for $r_2$ in $r_1 / r_2 == 0$, where $r_1 = [-100, -10]$ and $r_2 = [10, \infty]$, using the $/_{\text{int}}$ operator. The restricting interval for the expression $r_1 / r_2$ is obviously $[0, 0]$. This means that the value of $r_2$ would be restricted by $[-100, -10] /_{\text{int}} [0, 0] = [-\infty, 100]$. Thus, the resulting value of $r_2$ would be $[-\infty, 100] \cap_{\text{int}} [10, \infty] = [10, 100]$. However, also the interval $[10, \infty]$ for $r_2$ will fulfill the relation $r_1 / r_2 == 0$ since $[-100/\infty] = 0$ which means that $/_{\text{int}}$ gives an erroneous result in this case.

Next consider calculating the restricting interval for $r_1$ in $r_1 * r_2 <= -7$, where $r_1 = [-100, 100]$ and $r_2 = [8, 40]$, using the $/_{\text{int}}$ operator. The restricting interval for the expression $r_1 * r_2$ is obviously $[-\infty, \infty]$. This means that the value of $r_1$ would be restricted by $[-\infty, \infty] /_{\text{int}} [8, 40] = [-1, \infty]$. Thus, the re-
Table 5.4: Arithmetic restriction for intervals.

\[
\begin{align*}
\tilde{\mathcal{A}}_R[n](i_R)\tilde{x} &= \tilde{x} \\
\tilde{\mathcal{A}}_R[r](i_R)\tilde{x} &= \begin{cases} 
\tilde{x}[r \mapsto i] & \text{if } i \neq \bot_{\text{int}} \text{ where } i = \tilde{x} \cap_{\text{int}} i_R \\
\bot_{\text{reg}} & \text{otherwise}
\end{cases}
\end{align*}
\]

\[
\begin{align*}
\tilde{\mathcal{A}}_R[a_1 + a_2](i_R)\tilde{x} &= (\tilde{\mathcal{A}}_R[a_1](i_R - \text{int } \tilde{\mathcal{A}}_R[a_2] \tilde{x}) \cap_{\text{reg}} \\
&\quad (\tilde{\mathcal{A}}_R[a_2](i_R - \text{int } \tilde{\mathcal{A}}_R[a_1] \tilde{x}) \tilde{x})
\end{align*}
\]

\[
\begin{align*}
\tilde{\mathcal{A}}_R[a_1 - a_2](i_R)\tilde{x} &= (\tilde{\mathcal{A}}_R[a_1](i_R + \text{int } \tilde{\mathcal{A}}_R[a_2] \tilde{x}) \cap_{\text{reg}} \\
&\quad (\tilde{\mathcal{A}}_R[a_2](\tilde{\mathcal{A}}_R[a_1] \tilde{x} - \text{int } i_R) \tilde{x})
\end{align*}
\]

\[
\begin{align*}
\tilde{\mathcal{A}}_R[a_1 \times a_2](i_R)\tilde{x} &= \begin{align*}
&(\tilde{\mathcal{A}}_R[a_1](i_R \odot_{\text{int}} (\tilde{\mathcal{A}}_R[a_2] \tilde{x} \cap_{\text{int}} [-\infty, -1]) \cup_{\text{int}} \\
&\quad (i_R \odot_{\text{int}} (\tilde{\mathcal{A}}_R[a_2] \tilde{x} \cap_{\text{int}} [0, 0]) \cup_{\text{int}} \\
&\quad (i_R \odot_{\text{int}} (\tilde{\mathcal{A}}_R[a_2] \tilde{x} \cap_{\text{int}} [1, \infty]) \tilde{x}) \cap_{\text{reg}} \\
&\quad ((i_R \odot_{\text{int}} (\tilde{\mathcal{A}}_R[a_1] \tilde{x} \cap_{\text{int}} [-\infty, -1]) \cup_{\text{int}} \\
&\quad (i_R \odot_{\text{int}} (\tilde{\mathcal{A}}_R[a_1] \tilde{x} \cap_{\text{int}} [0, 0]) \cup_{\text{int}} \\
&\quad (i_R \odot_{\text{int}} (\tilde{\mathcal{A}}_R[a_1] \tilde{x} \cap_{\text{int}} [1, \infty]) \tilde{x}) \cap_{\text{reg}} \\
&\quad (((\tilde{\mathcal{A}}_R[a_1] \tilde{x} \cap_{\text{int}} [-\infty, -1]) \odot_{\text{int}} i_R) \cup_{\text{int}} \\
&\quad (((\tilde{\mathcal{A}}_R[a_1] \tilde{x} \cap_{\text{int}} [0, 0]) \odot_{\text{int}} i_R) \cup_{\text{int}} \\
&\quad (((\tilde{\mathcal{A}}_R[a_1] \tilde{x} \cap_{\text{int}} [1, \infty]) \odot_{\text{int}} i_R) \tilde{x})
\end{align*}
\]
\]

\[
\begin{align*}
\tilde{\mathcal{A}}_R[a_1 / a_2](i_R)\tilde{x} &= \begin{align*}
&\begin{cases} 
\tilde{x}[r \mapsto i] & \text{if } i \neq \bot_{\text{int}} \text{ where } i = \tilde{x} \cap_{\text{int}} i_R \\
\bot_{\text{reg}} & \text{otherwise}
\end{cases}
\end{align*}
\end{align*}
\]

\[
\begin{align*}
&\begin{cases} 
\tilde{x}[r \mapsto i] & \text{if } i \neq \bot_{\text{int}} \text{ where } i = \tilde{x} \cap_{\text{int}} i_R \\
\bot_{\text{reg}} & \text{otherwise}
\end{cases}
\end{align*}
\]
sulting value of \( r_1 \) would be \([-1, 100]\). However, the interval \([0, 100]\) is also a safe restricted value for \( r_1 \) since \(-1 \times 8 \leq -7\) and \(-1 \times 40 \leq -7\) (as well as for any value in-between 8 and 40) which means that \( /_\text{int} \) gives an unnecessarily un-tight result in this case.

Table 5.5 defines the operator \( \otimes'_{\text{int}} \) which is used in the definition of \( \mathcal{A}_{\mathcal{G}} \), found in Table 5.4. This operator is derived based on the following equation solution in the infinite integer domain, \( \mathbb{Z} \cup \{-\infty, \infty\} \), and it safely calculates the restricting interval, \( i^{a_1}_{\mathcal{G}} \in \text{Intv} \), for \( a_1 \) in the expression \( a_1 / a_2 \) based on a safe restricting interval, \( i^{a_2}_{\mathcal{G}} \in \text{Intv} \), for that expression and a known value, \( i^{a_2}_{\mathcal{G}} \in \text{Intv} \), for \( a_2 \) (i.e., \( i^{a_1}_{\mathcal{G}} = i^{a_2}_{\mathcal{G}} \otimes'_{\text{int}} i^{a_2}_{\mathcal{G}} \)). Note that in Table 5.4, the operator is used on a strictly negative interval (or \( \bot_{\text{int}} \)), a strictly positive interval (or \( \top_{\text{int}} \)) and the \([0,0]\) interval (or \( \bot_{\text{int}} \)) for \( i^{a_2}_{\mathcal{G}} \). This is to make the definition of the operator more concise. Also note that any operator fulfilling (i.e., over-approximating) the below result qualifies as a substitute for \( \otimes'_{\text{int}} \).

\( \otimes'_{\text{int}} \): Solving \( a_1 / a_2 = v \iff [a_1 / a_2] = v \iff v \leq a_1 / a_2 < v + 1 \) for \( a_1 \) gives:

\[
\begin{align*}
(a_2 = -\infty \vee a_2 = \infty) \wedge v = 0 & \Rightarrow -\infty < a_1 < \infty \\
(a_2 = -\infty \vee a_2 = \infty) \wedge v \neq 0 & \Rightarrow \text{no solution} \\
a_2 = 0 \wedge -\infty < v < \infty & \Rightarrow 0 < a_1 \leq \infty \\
a_2 = 0 \wedge v = -\infty & \Rightarrow -\infty \leq a_1 < 0 \\
0 < a_2 < \infty \wedge v \in \{-\infty, \infty\} & \Rightarrow a_1 = v \\
0 < a_2 < \infty \wedge 0 < v < \infty & \Rightarrow 0 < a_2 v \leq a_1 < a_2 (v + 1) \\
0 < a_2 < \infty \wedge v = 0 & \Rightarrow 0 \leq a_1 < a_2 \\
0 < a_2 < \infty \wedge v = -1 & \Rightarrow -a_2 \leq a_1 < 0 \\
0 < a_2 < \infty \wedge -\infty < v < -1 & \Rightarrow a_2 v \leq a_1 < a_2 (v + 1) < 0 \\
-\infty < a_2 < 0 \wedge v \in \{-\infty, \infty\} & \Rightarrow a_1 = -v \\
-\infty < a_2 < 0 \wedge 0 < v < \infty & \Rightarrow a_2 (v + 1) < a_1 \leq a_2 v < 0 \\
-\infty < a_2 < 0 \wedge v = 0 & \Rightarrow a_1 = 0 \\
-\infty < a_2 < 0 \wedge v = -1 & \Rightarrow 0 < a_1 \leq -a_2 \\
-\infty < a_2 < 0 \wedge -\infty < v < -1 & \Rightarrow 0 < a_2 (v + 1) < a_1 \leq a_2 v
\end{align*}
\]

Table 5.6 defines the operator \( \otimes^*_\text{int} \) which is used in the definition of \( \mathcal{A}_{\mathcal{G}} \), found in Table 5.4. This operator is derived based on the following equation solution in the infinite integer domain, \( \mathbb{Z} \cup \{-\infty, \infty\} \), and it safely calculates the restricting interval, \( i^{a_1}_{\mathcal{G}} \in \text{Intv} \), for \( a_1 \) in the expression \( a_1 * a_2 \) based on
Table 5.5: Multiplication operator for determining the restricting interval for the numerator in an interval division expression.

<table>
<thead>
<tr>
<th>$\bot_{\text{int}} \otimes'_{\text{int}} i$</th>
<th>$i \otimes'<em>{\text{int}} \bot</em>{\text{int}} = \bot_{\text{int}}$ where $i \in \text{Intv}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$[l_1, u_1] \otimes'_{\text{int}} [l_2, u_2]$</td>
<td></td>
</tr>
</tbody>
</table>

\[
\bot_{\text{int}} \text{ if } [l_2, u_2] = [0, 0] \land \{ -\infty, \infty \} \cap \gamma_{\text{int}}([l_1, u_1]) = \emptyset \\
\bot_{\text{int}} \text{ if } [l_2, u_2] \in \{[-\infty, -\infty], [\infty, \infty]\} \land 0 \notin \gamma_{\text{int}}([l_1, u_1]) \\
\min(V_1 \cup V_2 \cup V_3^{\min} \cup V_4^{\min} \cup V_5^{\min} \cup V_6^{\min} \cup V_7 \cup V_8 \cup V_9 \cup V_{10}), \\
\max(V_1 \cup V_2 \cup V_3^{\max} \cup V_4^{\max} \cup V_5^{\max} \cup V_6^{\max} \cup V_7 \cup V_8 \cup V_9 \cup V_{10}) \text{ otrw}
\]

where

\[
V_1 = \{ -\infty, \infty \} \text{ if } \{ -\infty, \infty \} \cap \gamma_{\text{int}}([l_2, u_2]) \neq \emptyset \land 0 \notin \gamma_{\text{int}}([l_1, u_1]) \\
V_2 = \{ 0 \} \text{ if } 0 \notin \gamma_{\text{int}}([l_1, u_1]) \\
V_3^{\min} = \{ l_1 l_2 \} \text{ if } 0 < l_1 \land 0 < l_2 \\
V_3^{\max} = \{ u_1 u_2 + u_2 - 1 \} \text{ if } 0 \leq u_1 \land 0 < u_2 < \infty \\
V_4^{\min} = \{ l_1 u_2 \} \text{ if } l_1 < 0 \land 0 < u_2 \\
V_4^{\max} = \{ u_1 l_2 + l_2 - 1 \} \text{ if } u_1 < 0 \land 0 < l_2 \\
V_5^{\min} = \{ u_1 l_2 + l_2 + 1 \} \text{ if } 0 \leq u_1 \land -\infty < l_2 < 0 \\
V_5^{\max} = \{ l_1 u_2 \} \text{ if } 0 < l_1 \land u_2 < 0 \\
V_6^{\min} = \{ u_1 u_2 + u_2 + 1 \} \text{ if } u_1 < 0 \land u_2 < 0 \\
V_6^{\max} = \{ l_1 l_2 \} \text{ if } l_1 < 0 \land l_2 < 0 \\
V_7 = \{ \infty \} \text{ if } (\infty \in \gamma_{\text{int}}([l_1, u_1]) \land 0 < u_2) \lor \\
(\infty \in \gamma_{\text{int}}([l_1, u_1]) \land l_2 < 0) \\
V_8 = \{ -\infty \} \text{ if } (-\infty \in \gamma_{\text{int}}([l_1, u_1]) \land 0 < u_2) \lor \\
(-\infty \in \gamma_{\text{int}}([l_1, u_1]) \land l_2 < 0) \\
V_9 = \{ 1, \infty \} \text{ if } \infty \in \gamma_{\text{int}}([l_1, u_1]) \land 0 \in \gamma_{\text{int}}([l_2, u_2]) \\
V_{10} = \{ -\infty, -1 \} \text{ if } -\infty \in \gamma_{\text{int}}([l_1, u_1]) \land 0 \in \gamma_{\text{int}}([l_2, u_2])
\]

The sets $V_i$, where $i \in \{ 1, 2, 7, 8, 9, 10 \}$, and $V_i^m$, where $i \in \{ 3, 4, 5, 6 \}$ and $m \in \{ \text{min, max} \}$, have the value $\emptyset$ whenever their condition is not met.
a safe restricting interval, $i_{\mathcal{R}} \in \text{Intv}$, for that expression and a known value, $i_{\mathcal{R}}^a \in \text{Intv}$, for $a_2$ (i.e., $i_{\mathcal{R}}^a = i_{\mathcal{R}}^* \triangledown_{\text{int}} i_{\mathcal{R}}^a$). Remember that if $i_{\mathcal{R}}^a \in [l, u]$ is such that $u < l$, then $i_{\mathcal{R}}^a = \bot_{\text{int}}$ (Definition 3.31). Note that in Table 5.4, the operator is used on a strictly negative interval (or $\bot_{\text{int}}$), a strictly positive interval (or $\bot_{\text{int}}$) and the $[0,0]$ interval (or $\bot_{\text{int}}$) for $i_{\mathcal{R}}^a$. This is to make the definition of the operator more concise. Also note that the restricting interval for $a_2$ is calculated in an equivalent manner: $i_{\mathcal{R}}^a = i_{\mathcal{R}}^* \triangledown_{\text{int}} i_{\mathcal{R}}^a$. Further note that any operator fulfilling (i.e., over-approximating) the below result qualifies as a substitute for $\triangledown_{\text{int}}^*$.

$\triangledown_{\text{int}}^*$: Solving $a_1 \ast a_2 = v \iff a_1 a_2 = v$ for $a_1$ gives (note that if $0 < |v| < |a_2|$ then the equation has no solution since $a_1$ must be an integer; also note that solving the equation for $a_2$ is done in an equivalent manner):

\[
\begin{align*}
a_2 &= 0 \land v \neq 0 \quad \Rightarrow \quad \text{no solution} \\
a_2 &= 0 \land v = 0 \quad \Rightarrow \quad -\infty < a_1 < \infty \\
a_2 \in \{-\infty, \infty\} \land v = 0 &\quad \Rightarrow \quad \text{no solution} \\
a_2 \in \{-\infty, \infty\} \land v = a_2 &\quad \Rightarrow \quad 0 < a_1 \leq \infty \\
a_2 \in \{-\infty, \infty\} \land v = -a_2 &\quad \Rightarrow \quad -\infty \leq a_1 < 0 \\
a_2 \in \{-\infty, \infty\} \land -\infty < v < 0 &\quad \Rightarrow \quad \text{no solution} \\
a_2 \in \{-\infty, \infty\} \land 0 < v < \infty &\quad \Rightarrow \quad \text{no solution} \\
0 < a_2 < \infty \land v \in \{-\infty, \infty\} &\quad \Rightarrow \quad a_1 = v \\
0 < a_2 < \infty \land v = 0 &\quad \Rightarrow \quad a_1 = 0 \\
0 < a_2 < \infty \land a_2 \leq v < \infty &\quad \Rightarrow \quad 0 < \lfloor v/a_2 \rfloor \leq a_1 < \lfloor v/a_2 \rfloor + 1 \\
0 < a_2 < \infty \land 0 < v < a_2 &\quad \Rightarrow \quad \text{no solution} \\
0 < a_2 < \infty \land -\infty < v \leq -a_2 &\quad \Rightarrow \quad \lceil v/a_2 \rceil - 1 < a_1 \leq \lceil v/a_2 \rceil < 0 \\
0 < a_2 < \infty \land -a_2 < v < 0 &\quad \Rightarrow \quad \text{no solution} \\
-\infty < a_2 < 0 \land v \in \{-\infty, \infty\} &\quad \Rightarrow \quad a_1 = -v \\
-\infty < a_2 < 0 \land v = 0 &\quad \Rightarrow \quad a_1 = 0 \\
-\infty < a_2 < 0 \land -a_2 \leq v < \infty &\quad \Rightarrow \quad \lceil v/a_2 \rceil - 1 < a_1 \leq \lceil v/a_2 \rceil < 0 \\
-\infty < a_2 < 0 \land 0 < v < -a_2 &\quad \Rightarrow \quad \text{no solution} \\
-\infty < a_2 < 0 \land -\infty < v \leq a_2 &\quad \Rightarrow \quad 0 < \lfloor v/a_2 \rfloor \leq a_1 < \lfloor v/a_2 \rfloor + 1 \\
-\infty < a_2 < 0 \land a_2 < v < 0 &\quad \Rightarrow \quad \text{no solution}
\end{align*}
\]

Table 5.7 defines the operator $\triangledown_{\text{int}}^*$ which is used in the definition of $\mathcal{R}$, found in Table 5.4. This operator is derived based on the following equation.
### 5.4 Boolean Restriction for Intervals

Table 5.6: Division operator for determining the restricting interval for the factors of an interval multiplication expression.

<table>
<thead>
<tr>
<th>$\perp_{\text{int}} \circ_i^* [l_1, u_1] \circ_i^* [l_2, u_2]$</th>
<th>$\perp_{\text{int}} = \perp_{\text{int}} \quad \text{where} \ i \in \text{Intv}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\perp_{\text{int}}$</td>
<td>$\text{if } 0 \not\in \gamma_{\text{int}}([l_1, u_1]) \land [l_2, u_2] = [0, 0]$</td>
</tr>
<tr>
<td>$\perp_{\text{int}}$</td>
<td>$\text{if } {\infty, \infty} \cap \gamma_{\text{int}}([l_1, u_1]) = \emptyset \land [l_2, u_2] \in {[-\infty, -\infty], [\infty, \infty]}$</td>
</tr>
<tr>
<td>$\perp_{\text{int}}$</td>
<td>$\text{if } 0 \not\in \gamma_{\text{int}}([l_1, u_1]) \land \max{</td>
</tr>
</tbody>
</table>

$[-\infty, \infty]$ if $0 \in \gamma_{\text{int}}([l_1, u_1]) \land 0 \in \gamma_{\text{int}}([l_2, u_2])$

$\min(V_0 \cup V_1^\min \cup V_2^\min \cup V_3 \cup V_4 \cup V_5^\min \cup V_6^\min \cup V_7^\min \cup V_8^\min), \max(V_0 \cup V_1^\max \cup V_2^\max \cup V_3 \cup V_4 \cup V_5^\max \cup V_6^\max \cup V_7^\max \cup V_8^\max)$

**otrw**

where

- $V_0 = \{0\}$ if $0 \not\in \gamma_{\text{int}}([l_1, u_1]) \land u_2 \neq -\infty \land l_2 \neq -\infty$
- $V_1^\min = \{1\}$ if $l_1 = l_2 = -\infty \lor u_1 = u_2 = \infty$
- $V_1^\max = \{\infty\}$ if $l_1 = l_2 = -\infty \lor u_1 = u_2 = \infty$
- $V_2^\min = \{-\infty\}$ if $l_1 = u_2 = -\infty \lor u_1 = -l_2 = \infty$
- $V_2^\max = \{-1\}$ if $l_1 = -u_2 = -\infty \lor u_1 = -l_2 = \infty$
- $V_3 = \{-\infty\}$ if $(u_1 = \infty \land l_2 < 0) \lor (l_1 = -\infty \land 0 < u_2)$
- $V_4 = \{\infty\}$ if $(u_1 = \infty \land 0 < u_2) \lor (l_1 = -\infty \land l_2 < 0)$
- $V_5^\min = \{[l_1/l_2] + (u_2 = \infty ? 1 : 0)\}$ if $0 < l_1 < \infty \land 0 < u_2$
- $V_5^\max = \{[u_1/l_2] - (l_2 = \infty ? 1 : 0)\}$ if $0 < u_1 < \infty \land 0 < l_2$
- $V_6^\min = \{[l_1/l_2] + (l_2 = \infty ? 1 : 0)\}$ if $-\infty < l_1 < 0 \land 0 < l_2$
- $V_6^\max = \{[u_1/u_2] - (u_2 = \infty ? 1 : 0)\}$ if $-\infty < u_1 < 0 \land 0 < u_2$
- $V_7^\min = \{[u_1/u_2] + (u_2 = -\infty ? 1 : 0)\}$ if $0 < u_1 < \infty \land u_2 < 0$
- $V_7^\max = \{[l_1/l_2] - (l_2 = -\infty ? 1 : 0)\}$ if $0 < l_1 < \infty \land l_2 < 0$
- $V_8^\min = \{[l_1/l_2] + (l_2 = -\infty ? 1 : 0)\}$ if $-\infty < l_1 < 0 \land l_2 < 0$
- $V_8^\max = \{[l_1/l_2] - (u_2 = -\infty ? 1 : 0)\}$ if $-\infty < l_1 < 0 \land u_2 < 0$

The sets $V_i$, where $i \in \{0, 3, 4\}$, and $V_i^m$, where $i \in \{1, 2, 5, 6, 7, 8\}$ and $m \in \{\text{min}, \text{max}\}$, have the value $\emptyset$ whenever their condition is not met.
solution in the infinite integer domain, \( \mathbb{Z} \cup \{-\infty, \infty\} \), and it safely calculates the restricting interval, \( i_{\mathbb{R}}^{a_2} \in \text{Intv} \), for \( a_2 \) in the expression \( a_1 / a_2 \) based on a safe restricting interval, \( i_{\mathbb{R}} \in \text{Intv} \), for that expression and a known value, \( i^{a_1} \in \text{Intv} \), for \( a_1 \) (i.e., \( i_{\mathbb{R}}^{a_2} = i^{a_1} \circ \text{int} i_{\mathbb{R}} \)). Remember that if \( i_{\mathbb{R}}^{a_2} \in [l, u] \) is such that \( u < l \), then \( i_{\mathbb{R}}^{a_2} = \bot_{\text{int}} \) (Definition 3.31). Note that in Table 5.4, the operator is used on a strictly negative interval (or \( \bot_{\text{int}} \)), a strictly positive interval (or \( \bot_{\text{int}} \)) and the \([0, 0]\) interval (or \( \bot_{\text{int}} \)) for \( i^{a_1} \). This is to make the definition of the operator more concise. Also note that any operator fulfilling (i.e., over-approximating) the below result qualifies as a substitute for \( \circ \).}

\( \circ_{\text{int}} \): Solving \( a_1 / a_2 = v \iff |a_1/a_2| = v \iff v \leq a_1/a_2 < v + 1 \) for \( a_2 \) gives:

\[
\begin{align*}
a_1 &= 0 \land v = 0 & \Rightarrow & -\infty \leq a_2 < 0 \lor 0 < a_2 \leq \infty \\
a_1 &= 0 \land v \neq 0 & \Rightarrow & \text{no solution} \\
a_1 \in \{-\infty, \infty\} \land v = 0 & \Rightarrow & \text{no solution} \\
a_1 \in \{-\infty, \infty\} \land v = a_1 & \Rightarrow & 0 \leq a_2 < \infty \\
a_1 \in \{-\infty, \infty\} \land v = -a_1 & \Rightarrow & -\infty < a_2 < 0 \\
a_1 \notin \{-\infty, 0, \infty\} \land v \notin \{-\infty, 0, \infty\} & \Rightarrow & \text{no solution} \\
a_1 \notin \{-\infty, 0, \infty\} \land v/a_1 = -\infty & \Rightarrow & \text{no solution} \\
a_1 \notin \{-\infty, 0, \infty\} \land v/a_1 = \infty & \Rightarrow & a_2 = 0 \\
0 < a_1 < \infty \land 0 < v \leq a_1 & \Rightarrow & 0 < a_1/(v + 1) < a_2 \leq a_1/v \\
0 < a_1 < \infty \land a_1 < v < \infty & \Rightarrow & \text{no solution} \\
0 < a_1 < \infty \land v = 0 & \Rightarrow & a_2 \in \{-\infty, \infty\} \\
0 < a_1 < \infty \land v = -1 & \Rightarrow & -\infty < a_2 \leq -a_1 < 0 \\
0 < a_1 < \infty \land -a_1 \leq v < -1 & \Rightarrow & a_1/(v + 1) < a_2 \leq a_1/v < 0 \\
0 < a_1 < \infty \land -\infty < v < -a_1 & \Rightarrow & \text{no solution} \\
-\infty < a_1 < 0 \land 0 < v \leq -a_1 & \Rightarrow & a_1/v \leq a_2 < a_1/(v + 1) < 0 \\
-\infty < a_1 < 0 \land -a_1 < v < \infty & \Rightarrow & \text{no solution} \\
-\infty < a_1 < 0 \land v = 0 & \Rightarrow & a_2 \in \{-\infty, \infty\} \\
-\infty < a_1 < 0 \land v = -1 & \Rightarrow & 0 < -a_1 \leq a_2 < \infty \\
-\infty < a_1 < 0 \land a_1 \leq v < -1 & \Rightarrow & 0 < a_1/v \leq a_2 < a_1/(v + 1) \\
-\infty < a_1 < 0 \land -\infty < v < a_1 & \Rightarrow & \text{no solution}
\end{align*}
\]
Table 5.7: Division operator for determining the restricting interval for the denominator in an interval division expression.

\[
\begin{align*}
\bot_{\text{int}} \odot_{\text{int}} i &= i \odot_{\text{int}} \bot_{\text{int}} \quad \text{where } i \in \text{Intv} \\
[l_1, u_1] \odot_{\text{int}} [l_2, u_2] &= \\
&\begin{cases}
(-\infty, \infty) & \text{if } l_1 \neq -\infty \land u_1 \neq -\infty \land 0 \in \gamma_{\text{int}}([l_2, u_2]) \\
\bot_{\text{int}} & \text{if } [l_1, u_1] = [0, 0] \land 0 \notin \gamma_{\text{int}}([l_2, u_2]) \\
\bot_{\text{int}} & \text{if } (l_1 = -\infty \lor u_1 = -\infty) \land l_2 \neq -\infty \land u_2 \neq -\infty \\
\bot_{\text{int}} & \text{if } 0 \leq l_1 \land u_1 < -\infty \land u_2 = -\infty \\
\bot_{\text{int}} & \text{if } -\infty < l_1 \land u_1 \leq 0 \land l_2 = -\infty \\
\bot_{\text{int}} & \text{if } 0 \notin \gamma_{\text{int}}([l_2, u_2]) \land \max([\{l_1, u_1\}] < \min([\{l_2, u_2\}]) \land (l_2 = -\infty \Rightarrow 0 \leq l_1) \land (u_2 = -\infty \Rightarrow u_1 \leq 0) \\
&\text{min}(V_0 \cup V_1 \cup V_2 \cup V_3^{\min} \cup V_4^{\min} \cup V_5^{\min} \cup V_6^{\min} \cup V_7^{\min} \cup V_8^{\min}), \\
&\text{max}(V_0 \cup V_1 \cup V_2 \cup V_3^{\max} \cup V_4^{\max} \cup V_5^{\max} \cup V_6^{\max} \cup V_7^{\max} \cup V_8^{\max}) \quad \text{otr}
\end{cases}
\end{align*}
\]

where

\[
\begin{align*}
V_0 &= \{0\} & \text{if } (0 < u_1 \land u_2 = \infty) \lor (l_1 < 0 \land l_2 = -\infty) \\
V_1 &= \{0, \infty\} & \text{if } l_1 = l_2 = -\infty \lor u_1 = u_2 = \infty \\
V_2 &= \{-\infty, -1\} & \text{if } l_1 = -u_2 = -\infty \land u_1 = -l_2 = -\infty \\
V_3^{\min} &= \{\lceil l_1/(u_2 + 1) \rceil + 1\} & \text{if } 0 < l_1 \land 0 < u_2 \\
V_3^{\max} &= \{\lfloor u_1/l_2 \rfloor\} & \text{if } 0 < u_1 \land 0 < l_2 \\
V_4^{\min} &= \{\lceil u_1/(u_2 + 1) \rceil + 1\} & \text{if } 0 < u_1 \land u_2 < -1 \\
V_4^{\max} &= \{\lfloor l_1/l_2 \rfloor\} & \text{if } 0 < l_1 \land u_2 < -1 \\
V_5^{\min} &= \{-\infty\} & \text{if } 0 < u_1 \land -1 \in \gamma_{\text{int}}([l_2, u_2]) \\
V_5^{\max} &= \{-l_1\} & \text{if } 0 < l_1 \land -1 \in \gamma_{\text{int}}([l_2, u_2]) \\
V_6^{\min} &= \{\lfloor l_1/l_2 \rfloor\} & \text{if } l_1 < 0 \land 0 < l_2 \\
V_6^{\max} &= \{\lceil u_1/(u_2 + 1) \rceil - 1\} & \text{if } u_1 < 0 \land 0 < u_2 \\
V_7^{\min} &= \{\lfloor u_1/l_2 \rfloor\} & \text{if } u_1 < 0 \land l_2 < -1 \\
V_7^{\max} &= \{\lceil l_1/(u_2 + 1) \rceil - 1\} & \text{if } l_1 < 0 \land u_2 < -1 \\
V_8^{\min} &= \{-u_1\} & \text{if } u_1 < 0 \land -1 \in \gamma_{\text{int}}([l_2, u_2]) \\
V_8^{\max} &= \{-\infty\} & \text{if } l_1 < 0 \land -1 \in \gamma_{\text{int}}([l_2, u_2]) \\
\end{align*}
\]

The sets \(V_i\), where \(i \in \{0, 1, 2\}\), and \(V_i^{m}\), where \(i \in \{3, 4, 5, 6, 7, 8\}\) and \(m \in \{\text{min, max}\}\), have the value \(\emptyset\) whenever their condition is not met.
5.5 Abstract Variable States

Using Theorems 3.17, 3.20, 3.24 and 3.39, a Galois connection, \(\langle \alpha_{var}, \gamma_{var} \rangle\), between the concrete domain \(\mathcal{P}(\text{Var} \rightarrow \text{Thrd} \rightarrow \mathcal{P}(\text{Val} \times \text{Time}))\) and the abstract domain \((\text{Var} \rightarrow \text{Thrd} \rightarrow \mathcal{P}(\tilde{\text{Val}} \times \tilde{\text{Time}})) \cup \{\tilde{1}_{\text{var}}, \tilde{\top}_{\text{var}}\} \ni \tilde{x}\) can be established.

The concretization function, \(\gamma_{var}\), and abstraction function, \(\alpha_{var}\), are given by Definitions 5.9 and 5.10, respectively. \(\tilde{x}\) is the bottom element, \(\tilde{1}_{\text{var}}\), if \(\exists x \in \text{Var} : \exists T \in \text{Thrd} : ((\tilde{x}, x, T)) = \emptyset\); i.e., some variable and thread combination maps to the empty set (there is no write-history available for that combination). Note that such an abstract variable state has no concrete counterparts \((\gamma_{var}(\tilde{1}_{\text{var}})) = \emptyset\). Therefore, an abstract variable state, \(\tilde{x}\), that actually contains no history for thread T on variable x, should have \(((\tilde{x}, x, T)) = \{((\tilde{1}_{\text{val}}, \tilde{1}_{t}))\}\) to make \(\tilde{x} \neq \tilde{1}_{\text{var}}\). Note that \(\gamma_{var}(\tilde{x})\), where \(((\tilde{x}, x, T)) = \{((\tilde{1}_{\text{val}}, \tilde{1}_{t}))\}\) for some variable, \(x\), and thread, \(T\), is a set of concrete states, \(X\), for which all \(\tilde{x} \in X\) are such that \(((\tilde{x}, x, T)) = \emptyset\). The top element, \(\tilde{\top}_{\text{var}}\), corresponds to a state where all variable and thread combinations are mapped to \(\tilde{\text{Val}} \times \tilde{\text{Time}}\).

**Definition 5.9 (Concretization of an abstract variable state):**

\[
\begin{align*}
\gamma_{var}(\tilde{1}_{\text{var}}) &= \text{Var} \rightarrow \text{Thrd} \rightarrow \mathcal{P}(\text{Val} \times \text{Time}) \\
\gamma_{var}(\tilde{1}_{\text{var}}) &= \emptyset \\
\gamma_{var}(\tilde{x}) &= \{f_{var} \in \text{Var} \rightarrow \text{Thrd} \rightarrow \mathcal{P}(\text{Val} \times \text{Time}) | \\
&\quad \forall x \in \text{Var} : (f_{var}(x)) \in \\
&\quad \{f_{\text{Thrd}} \in \text{Thrd} \rightarrow \mathcal{P}(\text{Val} \times \text{Time}) | \\
&\quad \forall T \in \text{Thrd} : (f_{\text{Thrd}}(T)) \in \{W'\} | \\
&\quad (\alpha_{\text{val}}(\{v \in \text{Val} | \exists t \in \text{Time} : (v, t) \in W'\}), \\
&\quad \alpha_{t}(\{t \in \text{Time} | \exists v \in \text{Val} : (v, t) \in W'\})) \in \\
&\quad ((\tilde{x}, x, T)))\}
\end{align*}
\]
5.5 Abstract Variable States

Definition 5.10 (Abstraction of a set of variable states):

\[
\begin{align*}
\alpha_{\text{var}}(\text{Var} \to \text{Thrd} \to \mathcal{P}(\text{Val} \times \text{Time})) &= \top_{\text{var}} \\
\alpha_{\text{var}}(\emptyset) &= \bot_{\text{var}} \\
\alpha_{\text{var}}(\mathbb{X}) &= \lambda x \in \text{Var}. \lambda T \in \text{Thrd}. \\
&\quad \{ (\alpha_{\text{val}}(\{v \in \text{Val} \mid \exists t \in \text{Time} : (v, t) \in W\}), \\
&\quad \alpha_t(\{t \in \text{Time} \mid \exists v \in \text{Val} : (v, t) \in W\})) \mid \ldots \}
\end{align*}
\]

Theorem 5.11 gives that \(\langle \alpha_{\text{var}}, \gamma_{\text{var}} \rangle\) is a Galois connection.

Theorem 5.11 (Galois connection – Variable states):
\(\langle \alpha_{\text{var}}, \gamma_{\text{var}} \rangle\), where \(\gamma_{\text{var}}\) and \(\alpha_{\text{var}}\) are defined as in Definitions 5.9 and 5.10, respectively, defines a Galois connection. \(\square\)

Proof. Since \(\langle \alpha_{\text{int}}, \gamma_{\text{int}} \rangle\) is a Galois insertion (Theorem 3.39) and thus a Galois connection, so are \(\langle \alpha_{\text{val}}, \gamma_{\text{val}} \rangle\) and \(\langle \alpha_t, \gamma_t \rangle\) (since \(\alpha_{\text{val}} = \alpha_t = \alpha_{\text{int}}\) and \(\gamma_{\text{val}} = \gamma_t = \gamma_{\text{int}}\)). Using Theorems 3.17, 3.20 and 3.24 to derive \(\alpha_{\text{var}}\) and \(\gamma_{\text{var}}\), the result follows (note that the cases \(\gamma_{\text{var}}(\top_{\text{var}}), \gamma_{\text{var}}(\bot_{\text{var}}), \alpha_{\text{var}}(\text{Var} \to \text{Thrd} \to \mathcal{P}(\text{Val} \times \text{Time}))\) and \(\alpha_{\text{var}}(\emptyset)\) follow trivially). This will be outlined in the following.

Since \(\langle \alpha_{\text{val}}, \gamma_{\text{val}} \rangle\) and \(\langle \alpha_t, \gamma_t \rangle\) are Galois connections, so is \(\langle \alpha_w, \gamma_w \rangle\) (Theorem 3.17), where:

\[
\begin{align*}
\alpha_w(W) &= (\alpha_{\text{val}}(\{v \in \text{Val} \mid \exists t \in \text{Time} : (v, t) \in W\}), \\
\alpha_t(\{t \in \text{Time} \mid \exists v \in \text{Val} : (v, t) \in W\})) \\
\gamma_w((\check{v}, \check{t})) &= \gamma_{\text{val}}(\check{v}) \times \gamma_t(\check{t})
\end{align*}
\]

and \(W \in \mathcal{P}(\text{Val} \times \text{Time})\) and \((\check{v}, \check{t}) \in \text{Vāl} \times \text{Time}.\)

Since \(\langle \alpha_w, \gamma_w \rangle\) is a Galois connection, so is \(\langle \alpha_\mathcal{P}, \gamma_\mathcal{P} \rangle\) (Theorem 3.20), where

\[
\begin{align*}
\alpha_\mathcal{P}(W') &= \{ \alpha_w(W) \mid W \in W' \} \\
\gamma_\mathcal{P}(D') &= \{ W \in \mathcal{P}(\text{Val} \times \text{Time}) \mid \alpha_w(W) \in D' \}
\end{align*}
\]

and \(W' \in \mathcal{P}(\text{Val} \times \text{Time})\) and \(D' \in \mathcal{P}(\text{Vāl} \times \text{Time}).\)

Since \(\langle \alpha_\mathcal{P}, \gamma_\mathcal{P} \rangle\) is a Galois connection, so is \(\langle \alpha_T, \gamma_T \rangle\) (Theorem 3.24), where

\[
\begin{align*}
\alpha_T(V') &= \lambda T \in \text{Thrd}. \alpha_\mathcal{P}(\{v' T \mid v' \in V'\}) \\
\gamma_T(d) &= \{ f_{\text{Thrd}} \in \text{Thrd} \to \mathcal{P}(\text{Val} \times \text{Time}) \mid \forall T \in \text{Thrd} : (f_{\text{Thrd}} T) \in \gamma_\mathcal{P}(d T) \}
\end{align*}
\]
and $V' \in \mathcal{P}(\text{Thr}d \to \mathcal{P}(\text{Val} \times \text{Time}))$ and $d \in \text{Thr}d \to \mathcal{P}(\text{Val} \times \text{Time})$.

Since $\langle \alpha_T, \gamma_T \rangle$ is a Galois connection, so is $\langle \alpha_{var}, \gamma_{var} \rangle$ (Theorem 3.24), where

$$
\begin{aligned}
\alpha_{var}(X) &= \lambda x \in \text{Var}. \alpha_T(\{x \mid x \in X\}) \\
\gamma_{var}(\tilde{x}) &= \{f_{var} \in \text{Var} \to \text{Thr}d \to \mathcal{P}(\text{Val} \times \text{Time}) \mid \\
& \quad \forall x \in \text{Var} : (f_{var} x) \in \gamma_T(\tilde{x})\}
\end{aligned}
$$

and $X \in \mathcal{P}(\text{Var} \to \text{Thr}d \to \mathcal{P}(\text{Val} \times \text{Time}))$ and $d \in \text{Var} \to \text{Thr}d \to \mathcal{P}(\text{Val} \times \text{Time})$.

Thus, by composing the different parts, the result follows:

$$
\gamma_{var}(\tilde{x}) \overset{\text{Th. 3.24}}{=} \{f_{var} \in \text{Var} \to \text{Thr}d \to \mathcal{P}(\text{Val} \times \text{Time}) \mid \\
& \quad \forall x \in \text{Var} : (f_{var} x) \in \gamma_T(\tilde{x})\}
$$

$$
\overset{\text{Th. 3.24}}{=} \{f_{var} \in \text{Var} \to \text{Thr}d \to \mathcal{P}(\text{Val} \times \text{Time}) \mid \\
& \quad \forall x \in \text{Var} : (f_{var} x) \in \\
& \quad \{f_{\text{Thr}d} \in \text{Thr}d \to \mathcal{P}(\text{Val} \times \text{Time}) \mid \\
& \quad \quad \forall T \in \text{Thr}d : (f_{\text{Thr}d} T) \in \gamma_{\mathcal{P}}((\tilde{x} x) T)\}\}
$$

$$
\overset{\text{Th. 3.20}}{=} \{f_{\text{var}} \in \text{Var} \to \text{Thr}d \to \mathcal{P}(\text{Val} \times \text{Time}) \mid \\
& \quad \forall x \in \text{Var} : (f_{\text{var}} x) \in \\
& \quad \{f_{\text{Thr}d} \in \text{Thr}d \to \mathcal{P}(\text{Val} \times \text{Time}) \mid \\
& \quad \quad \forall T \in \text{Thr}d : (f_{\text{Thr}d} T) \in \{W' \mid \alpha_w(W') \in ((\tilde{x} x) T)\}\}\}
$$

$$
\overset{\text{Th. 3.17}}{=} \{f_{\text{var}} \in \text{Var} \to \text{Thr}d \to \mathcal{P}(\text{Val} \times \text{Time}) \mid \\
& \quad \forall x \in \text{Var} : (f_{\text{var}} x) \in \\
& \quad \{f_{\text{Thr}d} \in \text{Thr}d \to \mathcal{P}(\text{Val} \times \text{Time}) \mid \\
& \quad \quad \forall T \in \text{Thr}d : (f_{\text{Thr}d} T) \in \{W' \mid \\
& \quad \quad \quad (\alpha_{val}(\{v \in \text{Val} \mid \exists t \in \text{Time} : (v, t) \in W\}), \\
& \quad \quad \quad \alpha_t(\{t \in \text{Time} \mid \exists v \in \text{Val} : (v, t) \in W\})) \in \\
& \quad \quad \quad ((\tilde{x} x) T)\}\}\}\}
$$

$$
\alpha_{var}(X) \overset{\text{Th. 3.24}}{=} \lambda x \in \text{Var}. \alpha_T(\{x \mid x \in X\})
$$

$$
\overset{\text{Th. 3.24}}{=} \lambda x \in \text{Var}. \lambda T \in \text{Thr}d. \alpha(T(\{f T \mid f \in \{x \mid x \in X\}\}))
$$

$$
\overset{\text{calc.}}{=} \lambda x \in \text{Var}. \lambda T \in \text{Thr}d. \alpha(T(\{x T \mid x \in X\}))
$$

$$
\overset{\text{Th. 3.20}}{=} \lambda x \in \text{Var}. \lambda T \in \text{Thr}d. \{\alpha_w(W) \mid W \in \{x T \mid x \in X\}\}
$$

$$
\overset{\text{Th. 3.17}}{=} \lambda x \in \text{Var}. \lambda T \in \text{Thr}d. \\
\{((\alpha_{val}(\{v \in \text{Val} \mid \exists t \in \text{Time} : (v, t) \in W\}), \\
\alpha_t(\{t \in \text{Time} \mid \exists v \in \text{Val} : (v, t) \in W\})) \mid \\
W \in \{x T \mid x \in X\}\}\} \blacksquare
$$
The state $\tilde{x} \in (\text{Var} \rightarrow \text{Thrd} \rightarrow \mathcal{P}(\tilde{\text{V}}, \tilde{\text{I}})) \cup \{\tilde{\text{V}}_{\text{var}}, \tilde{\text{I}}_{\text{var}}\}$ can save any number (i.e., history) of abstract writes, $\tilde{w} \in \tilde{\text{V}} \times \tilde{\text{I}}$, for each thread that occur on some variable. This is done to counteract the precision loss due to approximating points in time with intervals. The information available in such a history (i.e., a set of timestamped values) makes it possible to use sequence information (within each thread) and timing information (between threads) to get a reasonably tight value when reading a variable.

For convenience in expressing, and increased readability of, the upcoming algorithms, some relations for abstract writes, $\tilde{w} := (\tilde{v}, \tilde{t})$, will be defined. The partial order, $\subseteq_w$, and least upper bound operator, $\sqcup_w$, for abstract writes follow naturally (cf. Definitions 3.26 and 3.28) from the partial orders and least upper bound operators for abstract values, $\subseteq_{\text{val}}$ and $\sqcup_{\text{val}}$, and abstract time, $\subseteq_t$ and $\sqcup_t$. $\subseteq_w$ and $\sqcup_w$ are given by Definitions 5.12 and 5.13, respectively.

**Definition 5.12 (Partial order of writes):**

\[
\begin{align*}
\tilde{w} &\subseteq_w \tilde{t} \quad \tilde{w} \subseteq_w \tilde{t} \\
\tilde{w} &\subseteq_w \tilde{v} \\
(\tilde{v}_1, \tilde{t}_1) &\subseteq_w (\tilde{v}_2, \tilde{t}_2) \iff \tilde{v}_1 \subseteq_{\text{val}} \tilde{v}_2 \land \tilde{t}_1 \subseteq_t \tilde{t}_2
\end{align*}
\]

**Definition 5.13 (Least upper bound of writes):**

\[
\begin{align*}
\tilde{w} \sqcup_w \tilde{t} &\subseteq_w \tilde{t} \quad \sqcup_w \tilde{w} = \tilde{t} \\
\tilde{w} \sqcup_w \tilde{v} &= \tilde{w} \sqcup_w \tilde{v} = \tilde{w} \\
(\tilde{v}_1, \tilde{t}_1) \sqcup_w (\tilde{v}_2, \tilde{t}_2) &= (\tilde{v}_1 \sqcup_{\text{val}} \tilde{v}_2, \tilde{t}_1 \sqcup_t \tilde{t}_2)
\end{align*}
\]

The precedence relation, $\preceq_t$, on abstract times given by Definition 5.14 will be useful to determine whether two writes are performed at disjoint times (or the order of two arbitrary events).

**Definition 5.14 (Abstract time precedence):**

\[
\begin{align*}
\tilde{t} &\preceq_t \tilde{t} \quad \text{if } \tilde{t} \neq \tilde{t} \\
\tilde{t} &\preceq_t \tilde{t} \quad \text{if } \tilde{t} \neq \tilde{t} \\
\tilde{t}_1 &\preceq_t \tilde{t}_2 \iff \max(\gamma_t(\tilde{t}_1)) < \min(\gamma_t(\tilde{t}_2)) \quad \text{if } \tilde{t}_1, \tilde{t}_2 \not\in \{\tilde{t}_1, \tilde{t}_2\}
\end{align*}
\]
The definitions of the partial order relation, $\mathcal{E}_{\text{var}}$, the greatest lower bound operator, $\sqcap_{\text{var}}$, and the least upper bound operator, $\sqcup_{\text{var}}$, follow naturally from the definition of the domain (cf. Definitions 3.26, 3.27 and 3.28) and are presented in Definitions 5.15, 5.16 and 5.17, respectively.

**Definition 5.15 (Partial order for abstract variable states):**

$$
\begin{aligned}
\forall x \in \text{Var} : \forall T \in \text{Thrd} : (\tilde{x} T) \subseteq (\tilde{x'} T) &
\end{aligned}
$$

**Definition 5.16 (Greatest lower bound of abstract variable states):**

$$
\begin{aligned}
\forall x \in \text{Var} : \forall T \in \text{Thrd} : (\tilde{x} T) \cap (\tilde{x'} T)
\end{aligned}
$$

**Definition 5.17 (Least upper bound of abstract variable states):**

$$
\begin{aligned}
\forall x \in \text{Var} : \forall T \in \text{Thrd} : (\tilde{x} T) \cup (\tilde{x'} T)
\end{aligned}
$$

However, these relations and operators cannot be used directly within the analysis to, for example, join (merge) the histories of writes in several variable states. This is due to the fact that the history in the states might have different sequence information (i.e., traces), that would be lost if merging the two states. Reading a safe and tight value for a variable requires the sequence information to be available. Therefore, the operations to be used within the analysis should instead be defined based on Definition 5.19 to ensure that all threads see safe values at all abstract times. Note that Definition 5.18 determines the unique abstract time defining the most recent write(s) in a set of writes as illustrated for three different sets of writes in Figure 5.8. This definition determines the time of the most recent write both among several threads (i.e., globally) and for single threads (i.e., locally).

**Definition 5.18 (Time of most recent abstract write):**

The most recent write(s), $(\tilde{v}, \tilde{t})$, in a set of abstract writes is defined such that $\min(\gamma_t(\tilde{t})) \geq \min(\gamma_t(\tilde{t}'))$, for all other writes, $(\tilde{v}', \tilde{t}')$. If several writes, $(\tilde{v}', \tilde{t}')$,
The definitions of the partial order relation, \( \preccurlyeq \)\text{var}, the greatest lower bound operator, \( \sqcap \)\text{var}, and the least upper bound operator, \( \sqcup \)\text{var}, follow naturally from the definition of the domain (cf. Definitions 3.26, 3.27 and 3.28) and are presented in Definitions 5.15, 5.16 and 5.17, respectively.

Definition 5.15 (Partial order for abstract variable states):

\[
\begin{align*}
\preccurlyeq &\Rightarrow \forall x \in \text{Var}, \forall T \in \text{Thrd} : \\
&\left( \preceq x T \subseteq \left( \preceq' x T \right) \right)^2
\end{align*}
\]

Definition 5.16 (Greatest lower bound of abstract variable states):

\[
\begin{align*}
\sqcap = &\Rightarrow \sqcap \preceq \flat \preceq \top \text{var} = \Rightarrow ^{\flat \preceq \top \text{var}} \\
\sqcap &\Rightarrow \top \preceq \underline{\sqcap} \text{var} = \Rightarrow ^{\underline{\sqcap} \text{var}} \\
\sqcap &\Rightarrow \underline{\sqcap} \preceq \top \preceq \bot \text{var} = \Rightarrow ^{\preceq \bot \text{var}} \\
\sqcap &\Rightarrow \bot \preceq
\end{align*}
\]

Definition 5.17 (Least upper bound of abstract variable states):

\[
\begin{align*}
\sqcup = &\Rightarrow \sqcup \preceq \flat \preceq \top \text{var} = \Rightarrow ^{\top \text{var}} \\
\sqcup &\Rightarrow \top \preceq \underline{\sqcup} \text{var} = \Rightarrow ^{\underline{\sqcup} \text{var}} \\
\sqcup &\Rightarrow \underline{\sqcup} \preceq \top \preceq \bot \text{var} = \Rightarrow ^{\preceq \bot \text{var}} \\
\sqcup &\Rightarrow \bot \preceq
\end{align*}
\]

However, these relations and operators cannot be used directly within the analysis to, for example, join (merge) the histories of writes in several variable states. This is due to the fact that the history in the states might have different sequence information (i.e., traces), that would be lost if merging the two states. Reading a safe and tight value for a variable requires the sequence information to be available. Therefore, the operations to be used within the analysis should instead be defined based on Definition 5.19 to ensure that all threads see safe values at all abstract times. Note that Definition 5.18 determines the unique abstract time defining the most recent write(s) in a set of writes as illustrated for three different sets of writes in Figure 5.8. This definition determines the time of the most recent write both among several threads (i.e., globally) and for single threads (i.e., locally).

Definition 5.18 (Time of most recent abstract write):

The most recent write(s), \((\preceq v, \preceq t)\), in a set of abstract writes is defined such that

\[
\min (\gamma t (\preceq t)) \geq \min (\gamma t (\preceq t')) , \text{ for all other writes, } (\preceq v', \preceq t').
\]

If several writes, \((\preceq v', \preceq t')\), [255x578]5.5 Abstract Variable States 85

Figure 5.8: Three cases illustrating the timestamps (i.e., time intervals) of the abstract writes in some set and how the time of the most recent write among the writes in the set is defined.
are such that \( \min(\gamma_t(\bar{i})) = \min(\gamma_t(\bar{i}')) \), the abstract time of the most recent abstract write, \( \bar{i} \), is uniquely determined from the write(s) with \( \max(\gamma_t(\bar{i})) = \max(\{\max(\gamma_t(\bar{i})) \mid \bar{i}' \text{ ranges over the timestamps of the writes such that} \min(\gamma_t(\bar{i})) = \min(\gamma_t(\bar{i}'))\}) \).

\[ \square \]

**Definition 5.19 (Safe write history):**

An abstract variable state, \( \hat{x} \), is safe at abstract time \( \bar{i} \) if \( \gamma_{\text{var}}(\hat{x}) \) represents at least all the possible concrete variable states that can be valid at time \( t \in \gamma_t(\bar{i}) \) for the given thread trace(s).

Thus, to be safe at abstract time \( \bar{i} \), \( \hat{x} \) must, for each variable, \( x \in \text{Var} \), and each thread, \( T \in \text{Thrd} \), be such that \(((\hat{x}, x) T)\) contains at least

1. all writes, \((\bar{v}, \bar{i}')\), by \( T \) on \( x \), such that \( \bar{i}' \not\leq \bar{i} \wedge \bar{i} \not\geq \bar{i}' \), and

2. the latest (most recent) write(s), \((\bar{v}, \bar{i}'), \) by \( T \) on \( x \), such that \( \bar{i}' \not\leq \bar{i} \), if \( \bar{i}' \cap_t \bar{\tau}_{\text{mrw}} \neq \top_t \), where \( \bar{\tau}_{\text{mrw}} \) is the abstract time of the globally most recent write, such that \( \bar{\tau}_{\text{mrw}} \not\leq \bar{i} \),

or,

3. \((\bot_{\text{val}}, \bot_1)\), otherwise (i.e., if there are no writes that fit 1 or 2 above), or if no writes have occurred by \( T \) on \( x \).

From how the concrete and abstract domains (cf. Section 4.1 and this section) and transition rules (cf. Section 4.2) are defined, it is apparent that \( \hat{x} \) is a safe approximation of \( x \) (i.e., \( \hat{x} \) contains a safe write history) iff \( \exists x' \in \gamma_{\text{var}}(\hat{x}) : \forall x \in \text{Var} : \forall T \in \text{Thrd} : ((\hat{x}, x) T) \subseteq ((x', x) T) \).

\[ \square \]

Definition 5.20 states that the safe value of a variable as seen by a thread, \( T \), at an abstract time (i.e., interval in time), \( \bar{i} \), is the least upper bound of at least the values of the writes in the following two categories, whenever they exist.

1. The writes by threads other than \( T \) whose timestamps overlap with \( \bar{i} \).

2. The most recent write by threads other than \( T \) that definitely occurred before \( \bar{i} \), provided that the timestamp of that write overlaps with \( \bar{\tau}_{\text{mrw}} \), and the most recent write by \( T \) such that it overlaps with \( \bar{\tau}_{\text{mrw}} \), where \( \bar{\tau}_{\text{mrw}} \) is the timestamp of the globally single most recent write by threads other than \( T \) such that the timestamp precedes \( \bar{i} \), or the most recent timestamp for the writes by \( T \) such that the lower limit of that timestamp precedes the lower limit of \( \bar{i} \).
Figure 5.9: Three cases illustrating what category the abstract timestamps of the writes on some variable, $x$, which must be included for the value of $x$ as seen by thread $T_1$ at abstract time $\tilde{t}$ to be safe, belong to.
Figure 5.9 illustrates three different write histories and how the writes occurring in them are divided into the two categories as outlined by Definition 5.20.

**Definition 5.20 (Safe value of x as seen by thread T):**
Assuming that \( \tilde{x} \) contains a safe write history for all threads on variable \( x \), according to Definition 5.19, then a safe value of \( x \), as seen by thread \( T \), at abstract time \( \tilde{i} \) is the least upper bound, \( \bigcup_{val} \), of the values of at least the following writes on \( x \) (note that for \( T \), sequence information is used to minimize the number of writes that are taken into account).

1. All writes, \( \tilde{w}_{T'} = (\tilde{v}_{T'}, \tilde{i}_{T'}) \), for thread \( T' \in \text{Thrd} \setminus \{T\} \) on \( x \) such that \( \tilde{i}_{T'} < \tilde{i} \) and \( \tilde{i} \notin \tilde{i}_{T'} \).

2. The most recent write(s) in \( \{(\tilde{v}_{T'}, \tilde{i}_{T'}) \in (\tilde{x} \times T') \mid \tilde{i}_{T'} \leq \tilde{i} \wedge \tilde{i}_{T'} \cap \tilde{p}_{mrw} \neq \bot \} \) for each thread \( T' \in \text{Thrd} \setminus \{T\} \), and the single most recent write(s), \( (\tilde{v}_{T}, \tilde{i}_{T}) \in (\tilde{x} \times T) \), such that \( \min(\gamma_t(\tilde{i}_{T})) \leq \min(\gamma_t(\tilde{i})) \), if \( \tilde{i}_{T} \cap \tilde{p}_{mrw} \neq \bot \), where \( \tilde{p}_{mrw} \) is the abstract time of the (globally) most recent abstract write in \( \{(\tilde{v}_{T'}, \tilde{i}_{T'}) \in (\tilde{x} \times T') \mid \min(\gamma_t(\tilde{i}_{T'})) \leq \min(\gamma_t(\tilde{i}))\} \cup \bigcup_{T' \in \text{Thrd} \setminus \{T\}} \{(\tilde{v}_{T'}, \tilde{i}_{T'}) \in (\tilde{x} \times T') \mid \tilde{i}_{T'} \leq \tilde{i}\} \).

Note that Definitions 5.18, 5.19 and 5.20 depend on that points in time are approximated using intervals and that time cannot decrease between subsequent events (cf. Assumption 4.1, and Assumption 5.51 which will be made in Section 5.8 on page 114).

The partial order for abstract variable states to be used within the analysis, \( \tilde{\preceq}_{\text{var}} \), is given by Definition 5.21 based on \( \text{PARTIALORDERVAR} \), defined in Algorithm 5.1, taking the safety of write history (Definition 5.19) into account. Note that \( \text{EARLIESTWRITETHREAD} \), as defined in Algorithm 5.2, returns a deterministically defined write from the given set of writes. The idea is that the history (trace) for each thread and variable should be the same in both states for the relation to be true. However, the histories are allowed to differ somewhat. The greater state could also contain newer writes than those in the history of the lesser state. It could also be the case that the oldest write in the greater state that is not present in both states is an upper bound to all of the most recent writes in the lesser state that are not part of both histories.

**Definition 5.21 (Safe partial order of abstract variable states):**

\[
\begin{align*}
\tilde{x} \preceq_{\text{var}}' \tilde{v}_{\text{var}} \\
\bot_{\text{var}} \preceq_{\text{var}}' \tilde{x} \\
\tilde{x} \preceq_{\text{var}}' \tilde{x} \iff \text{PARTIALORDERVAR}(\tilde{x}, \tilde{x}')
\end{align*}
\]
Definition 5.21 (Safe partial order of abstract variable states): The partial order for abstract variable states to be used within the analysis, which is given by Definition 5.21 based on PARTIAL ORDER VAR, defined in Algorithm 5.1, taking the safety of write history (Definition 5.19) into account.

\[ \text{PARTIAL ORDER VAR}(\bar{x}, \bar{x}') \]

Algorithm 5.1 Partial order of abstract variable states

1: function PARTIALORDERVAR(\bar{x}, \bar{x}')
2: for all \( x \in \text{Var} \) do
3:     for all \( T \in \text{Thrd} \) do
4:         \( W \leftarrow ((\bar{x}, x) T) \)
5:         \( W' \leftarrow ((\bar{x}', x) T) \)
6:     while \( W \neq \emptyset \land W' \neq \emptyset \) do
7:         \( w \leftarrow \text{EARLIESTWRITETHREAD}(W) \)
8:         \( w' \leftarrow \text{EARLIESTWRITETHREAD}(W') \)
9:         \( W \leftarrow W \setminus \{w\} \)
10:        \( W' \leftarrow W' \setminus \{w'\} \)
11:     if \( w \neq w' \) then
12:         for all \( w'' \in W \cup \{w\} \) do
13:             if \( w'' \not\sqsubseteq_{w} w' \) then
14:                 return false
15:         end if
16:     end for
17:     \( W \leftarrow \emptyset \)
18: end if
19: end while
20: if \( W \neq \emptyset \) then
21:     return false
22: end if
23: end for
24: end for
25: return true
26: end function
Algorithm 5.2 Earliest write for a thread

1: function EARLIEST_WRITE_THREAD(\(\tilde{W}\))
2: if \(W = \emptyset\) then
3:   return \(\tilde{1}_W\)
4: end if
5: \(\tilde{t}_{\text{min}} \leftarrow [\infty, \infty]\)
6: for all \((\tilde{v}, \tilde{t}) \in \tilde{W}\) do
7:   if \(\min(\gamma_t(\tilde{t})) < \min(\gamma_{\tilde{t}_{\text{min}}}(\tilde{t}_{\text{min}}))\) then
8:     \(\tilde{t}_{\text{min}} \leftarrow \tilde{t}\)
9:   else if \(\min(\gamma_t(\tilde{t})) = \min(\gamma_{\tilde{t}_{\text{min}}}(\tilde{t}_{\text{min}}))\) then
10:    \(\tilde{t}_{\text{min}} \leftarrow \tilde{t} \cap \tilde{t}_{\text{min}}\)
11: end if
12: end for
13: \(\tilde{W}' \leftarrow \{(\tilde{v}, \tilde{t}) \mid (\tilde{v}, \tilde{t}) \in \tilde{W} \land \tilde{t} = \tilde{t}_{\text{min}}\}\)
14: \(\tilde{v}_{\text{min}} \leftarrow \alpha_{\text{val}}(\{\infty\})\)
15: for all \((\tilde{v}, \tilde{t}) \in \tilde{W}'\) do
16:   if \(\min(\gamma_{\text{val}}(\tilde{v})) < \min(\gamma_{\text{val}}(\tilde{v}_{\text{min}}))\) then
17:     \(\tilde{v}_{\text{min}} \leftarrow \tilde{v}\)
18:   else if \(\min(\gamma_{\text{val}}(\tilde{v})) = \min(\gamma_{\text{val}}(\tilde{v}_{\text{min}}))\) then
19:     \(\tilde{v}_{\text{min}} \leftarrow \tilde{v} \cap \gamma_{\text{val}}(\tilde{v}_{\text{min}})\)
20: end if
21: end for
22: return \((\tilde{v}_{\text{min}}, \tilde{t}_{\text{min}})\)
23: end function
Algorithm 5.3 Lower bounding two abstract variable states

1: function LOWERBOUNDVAR(\(\tilde{x}, \tilde{x}'\))
2: \(\tilde{x}'' \leftarrow \overline{\perp}_{\text{var}}\)
3: for all \(x \in \text{Var}\) do
4: for all \(T \in \text{Thrbd}\) do
5: \(\tilde{W} \leftarrow (\tilde{x} \times x) \cdot T\)
6: \(\tilde{W}' \leftarrow (\tilde{x}' \times x) \cdot T\)
7: \(C \leftarrow \emptyset\)
8: while \(\tilde{W} \neq \emptyset \land \tilde{W}' \neq \emptyset\) do
9: \((\tilde{v}, \tilde{t}) \leftarrow \text{EARLIESTWRITETHREAD(\(\tilde{W}\))}\)
10: \((\tilde{v}', \tilde{t}') \leftarrow \text{EARLIESTWRITETHREAD(\(\tilde{W}'\))}\)
11: \(\tilde{W} \leftarrow \tilde{W} \setminus (\tilde{v}, \tilde{t})\)
12: \(\tilde{W}' \leftarrow \tilde{W}' \setminus (\tilde{v}', \tilde{t}')\)
13: if \((\tilde{v}, \tilde{t}) = (\tilde{v}', \tilde{t}')\) then
14: \(C \leftarrow C \cup \{(\tilde{v}, \tilde{t})\}\)
15: else if \(\tilde{v} \cap_{\text{val}} v' \neq \overline{\perp}_{\text{var}} \land \tilde{t} \cap_{\text{t}} t' \neq \overline{\perp}_{\text{t}}\) then
16: \(C \leftarrow C \cup \{(\tilde{v} \cap_{\text{val}} v', \tilde{t} \cap_{\text{t}} t')\}\)
17: \(\tilde{W} \leftarrow \emptyset\)
18: \(\tilde{W}' \leftarrow \emptyset\)
19: else
20: \(\tilde{W} \leftarrow \emptyset\)
21: \(\tilde{W}' \leftarrow \emptyset\)
22: end if
23: end while
24: if \(C = \emptyset\) then
25: \((\tilde{x}'' \times x) \cdot T) \leftarrow \{(\overline{\perp}_{\text{val}}, \overline{\perp}_{\text{t}})\}\)
26: else
27: \((\tilde{x}'' \times x) \cdot T) \leftarrow C\)
28: end if
29: end for
30: end for
31: return \(\tilde{x}''\)
32: end function
Algorithm 5.4 Upper bounding two abstract variable states

1: function UPPERBOUNDVAR(\(\tilde{x}, \tilde{x}'\))
2: \(\tilde{x}'' \leftarrow \tilde{x}_\text{var}\)
3: for all \(x \in \text{Var}\) do
4:   for all \(T \in \text{Thrd}\) do
5:     \(W \leftarrow (\tilde{x}\ x)\ T\)
6:     \(\tilde{W} \leftarrow (\tilde{x}'\ x)\ T\)
7:     \(C \leftarrow \emptyset\)
8:     \(M \leftarrow (\tilde{\upsilon}\text{val}, \tilde{\upsilon}\text{r})\)
9:     while \(\tilde{W} \neq \emptyset \lor \tilde{W}' \neq \emptyset\) do
10:        \(\tilde{\upsilon} \leftarrow \text{EARLIESTWRITETHREAD}(\tilde{W})\)
11:        \(\tilde{\upsilon}' \leftarrow \text{EARLIESTWRITETHREAD}(\tilde{W}')\)
12:        if \(\tilde{\upsilon} = \tilde{\upsilon}'\) then
13:           \(C \leftarrow C \cup \{\tilde{\upsilon}\}\)
14:           \(W \leftarrow W \setminus \{\tilde{\upsilon}\}\)
15:           \(\tilde{W}' \leftarrow \tilde{W}' \setminus \{\tilde{\upsilon}'\}\)
16:        else if \(\tilde{W} = \emptyset\) then
17:           \(C \leftarrow C \cup \tilde{W}'\)
18:           \(\tilde{W}' \leftarrow \emptyset\)
19:        else if \(\tilde{W}' = \emptyset\) then
20:           \(C \leftarrow C \cup \tilde{W}\)
21:           \(\tilde{W} \leftarrow \emptyset\)
22:        else
23:           \(M \leftarrow (\bigsqcup_{\tilde{\upsilon}w} \tilde{W}) \sqcup_{\tilde{\upsilon}w} (\bigsqcup_{\tilde{\upsilon}w} \tilde{W}')\)
24:           \(W \leftarrow \emptyset\)
25:           \(\tilde{W}' \leftarrow \emptyset\)
26:        end if
27:     end while
28:     \(((\tilde{x}''\ x)\ T) \leftarrow C\)
29:     if \(M \neq (\tilde{\upsilon}\text{val}, \tilde{\upsilon}\text{r})\) then
30:        \(((\tilde{x}''\ x)\ T) \leftarrow ((\tilde{x}''\ x)\ T) \cup \{M\}\)
31:     end if
32: end for
33: end for
34: return \(\tilde{x}''\)
35: end function
Based on this partial order relation, the lower bound and upper bound operators to be used within the analysis, $\cap_{\text{var}}'$ and $\cup_{\text{var}}'$, are given by Definitions 5.22 and 5.23, respectively. Note that $\text{LOWERBOUNDVAR}$ is defined in Algorithm 5.3 and $\text{UPPERBOUNDVAR}$ is defined in Algorithm 5.4. Also note that the notation $((\bar{x}, x) T) \hookrightarrow \ldots$ will be used as a shorthand for $\bar{x} \leftarrow \bar{x}[x \mapsto (\bar{x}, x)[T \mapsto \ldots]]$ to increase readability. Intuitively, this could be compared to updating an element within a 2-dimensional array with a new value.

**Definition 5.22 (Safe lower bound of abstract variable states):**

$$\begin{align*}
\bar{x} \cap_{\text{var}}' \bar{x} &= \bar{x} \\
\perp_{\text{var}} \cap_{\text{var}}' \bar{x} &= \bar{x} \\
\bar{x} \cap_{\text{var}}' \perp &= \perp
\end{align*}$$

$$\bar{x} \cap_{\text{var}}' \bar{x}' = \text{LOWERBOUNDVAR}(\bar{x}, \bar{x}') \square$$

**Definition 5.23 (Safe upper bound of abstract variable states):**

$$\begin{align*}
\bar{x} \cup_{\text{var}}' \bar{x} &= \bar{x} \\
\perp_{\text{var}} \cup_{\text{var}}' \bar{x} &= \bar{x} \\
\bar{x} \cup_{\text{var}}' \perp &= \perp
\end{align*}$$

$$\bar{x} \cup_{\text{var}}' \bar{x}' = \text{UPPERBOUNDVAR}(\bar{x}, \bar{x}') \square$$

**NOTE.** Neither $\subseteq_{\text{var}}, \cap_{\text{var}}'$ nor $\cup_{\text{var}}'$ is currently used by the analysis (cf. Chapter 6) but are just presented for completeness of the abstraction since the operators cannot be directly based on the lattice. However, if for example merging of configurations [43] is introduced to lower the complexity of the analysis, at least $\cup_{\text{var}}'$ will be needed.

$\begin{aligned}
\text{WRITE}(T, \bar{x}, x, \bar{w}) \end{aligned}$, as defined in Algorithm 5.5, safely (Lemma 5.24) adds the write, $\bar{w}$, to the set of write-history for thread $T$ on $x$ in $\bar{x}$; i.e., to $((\bar{x}, x) T)$.

**Lemma 5.24 (Soundness of WRITE):**

Assuming that $\bar{x}$ contains a safe write history for variable $x$ and thread $T$ (cf. Definition 5.19) before the write by thread $T$ is performed at abstract time $\bar{i}$, then so will $\text{WRITE}(T, \bar{x}, x, (\bar{v}, \bar{i}))$.

**PROOF.** Since $\text{WRITE}(T, \bar{x}, x, (\bar{v}, \bar{i}))$ simply adds the write $(\bar{v}, \bar{i})$ to the history of thread $T$’s writes on variable $x$ in the state $\bar{x}$, and $\bar{x}$ is assumed to contain a safe write history for $T$ on $x$, $\text{WRITE}(T, \bar{x}, x, (\bar{v}, \bar{i}))$ trivially fulfills the safety condition in Definition 5.19 with regards to $T$ and $x$. $\square$
Algorithm 5.5 Write to variable

1: function WRITE(T, ˜x, x, w)
2: for all x' ∈ Var do
3: for all T' ∈ Thrd do
4: ( ˜x x') T' ← \{( ˜x x) T \} ∪ \{w\}  \text{ if } x' = x \land T' = T
5: end for
6: end for
7: return ˜x
8: end function

Using the sequence and timing information provided by Definition 5.20, READ( ˜x, x, T, ˜i), as defined in Algorithm 5.6, only takes the writes that might be valid at ˜i (the abstract point, i.e., interval, in time when T issues the READ on x given the variable state ˜x) into consideration for its returned value, ˜v ∈ Vãl, which is safe (Lemma 5.27). These writes, ˜w = (v', ˜i'), come from two categories, as specified in Definition 5.20: the first category covers the writes on x for threads T' ∈ Thrd \ {T} whose timestamps overlap in abstract time with ˜i, i.e., ˜i ⊓ ˜i' ̸= ⊥; the second category covers the most recent write(s) on x for all threads (including T) such that its timestamp overlaps with the overall most recent write of any write, not belonging to the first category. Note that any write for thread T with a timestamp that begins after the beginning of ˜i is discarded. So is any write for T' ∈ Thrd \ {T} such that its timestamp completely succeeds ˜i. This is because such writes can simply not have occurred at the time of issuing the READ (and will thus usually not be included in ˜x at all). Note that mostRecentWriteTime and mostRecentWriteTimeThread are defined based on Definition 5.18 in Algorithms 5.7 and 5.8, respectively, and that these functions give the abstract time of the most recent write among the writes in a set of writes (Lemmas 5.25 and 5.26).

Lemma 5.25 (Soundness of mostRecentWriteTimeThread): mostRecentWriteTimeThread(W), defined in Algorithm 5.8, gives the abstract time of the most recent abstract write in W.

Proof. This proof will be conducted based on the structure of Algorithm 5.8. If W = ∅, then ⊥ is returned. Otherwise, t\_min is the greatest lower limit of the timestamp of any write in W (max(\{min(γ(\tilde{i}) | ∃\tilde{v} ∈ Vãl : (\tilde{v}, \tilde{i}) ∈ \tilde{W}\})) and t\_max is the greatest upper limit of the timestamps of the writes in W such that the lower limit of their timestamps are equal to t\_min (max(∪\{γ(\tilde{i}) | ∃\tilde{v} ∈ Vãl :
Algorithm 5.6 Read from variable

1: function READ(\(\tilde{x}, x, T, \tilde{r}\))
2: \(\tilde{x}' \leftarrow \bot\)  
3: for all \(T' \in \text{Thrd} \setminus \{T\}\) do
4: \((\tilde{x}' x) T') \leftarrow \{(\tilde{v}', \tilde{r}') \in ((\tilde{x} x) T') \mid \tilde{r} \not\subseteq \tilde{r}'\}\)  
5: end for
6: \((\tilde{x}' x) T) \leftarrow \{(\tilde{v}', \tilde{r}') \in ((\tilde{x} x) T) \mid \min(\gamma(\tilde{r})) \geq \min(\gamma(\tilde{r}'))\}\)  
7: \(\tilde{W} \leftarrow \emptyset\)  
8: for all \(T' \in \text{Thrd} \setminus \{T\}\) do
9: \(\tilde{W}_{T'} \leftarrow \{(\tilde{v}', \tilde{r}') \in ((\tilde{x} x) T') \mid \tilde{r} \not\subseteq \tilde{r}'\}\)  
10: \((\tilde{x}' x) T') \leftarrow ((\tilde{x} x) T') \setminus \tilde{W}_{T'}\)  
11: \(\tilde{W} \leftarrow \tilde{W} \cup \tilde{W}_{T'}\)  
12: end for
13: \(\tilde{\rho}_{\text{mrw}} \leftarrow \text{MOSTRECENTWRITE}(\tilde{x}', x)\)
14: if \(\tilde{\rho}_{\text{mrw}} \not= \bot\) then
15: for all \(T' \in \text{Thrd}\) do
16: \(\tilde{\rho}_{\text{mrw}} \leftarrow \text{MOSTRECENTWRITETHREAD}(\tilde{x}', x) T')\)
17: \(\tilde{W} \leftarrow \tilde{W} \cup \{(\tilde{v}', \tilde{r}') \in ((\tilde{x} x) T') \mid \tilde{r}' \cap \tilde{\rho}_{\text{mrw}} \not= \bot \land \tilde{r}' \cap \tilde{\rho}_{\text{mrw}} \not= \bot\}\)  
18: end for
19: end if
20: \(\tilde{\nu} \leftarrow \bigcup_{\text{var}} \{v' \mid \exists \tilde{r}' \in \text{Time} : (\tilde{v}', \tilde{r}') \in \tilde{W}\} \quad \text{if} \; \tilde{W} \not= \emptyset\)  
21: \(\alpha_{\text{var}}(\{-\infty, \infty\})\) otrw
22: return \(\tilde{\nu}\)
23: end function

Algorithm 5.7 Time of most recent write

1: function MOSTRECENTWRITE(\(\tilde{x}, x\))
2: return MOSTRECENTWRITETHREAD(\(\bigcup_{T \in \text{Thrd}} ((\tilde{x} x) T)\))
3: end function

Algorithm 5.8 Time of most recent write in thread

1: function MOSTRECENTWRITETHREAD(\(\tilde{W}\))
2: if \(\tilde{W} = \emptyset\) then
3: return \(\bot\)
4: end if
5: \(t_{\text{min}} \leftarrow \max\{\min(\gamma(\tilde{r})) \mid \exists \tilde{v} \in \text{V\textbar } (\tilde{v}, \tilde{r}) \in \tilde{W}\}\}\)
6: \(t_{\text{max}} \leftarrow \max\{\{\gamma(\tilde{r}) \mid \exists \tilde{v} \in \text{V\textbar } (\tilde{v}, \tilde{r}) \in \tilde{W} \land \min(\gamma(\tilde{r})) = t_{\text{min}}\}\}\)
7: return \(\alpha_{\text{t}}(\{t_{\text{min}}, t_{\text{max}}\})\)
8: end function
Lemma 5.26 (Soundness of \textsc{MostRecentWriteTime}):

\textsc{MostRecentWriteTime}(\bar{x}, x), defined in Algorithm 5.7, gives the abstract time of the most recent abstract write on \( x \) in \( \bar{x} \).

\textbf{Proof.} This proof is trivial since \( \textsc{MostRecentWriteTimeThread}(\bar{W}) \) is the abstract time of the most recent abstract write in \( \bar{W} \) (Lemma 5.25) and the set of writes, \( \bar{W} \), is \( \bigcup_{T \in \text{\textsc{Thrd}}} ((\bar{x} x) T) \); i.e., \( \bar{W} \) is a set containing the abstract writes by all threads in \( \text{\textsc{Thrd}} \) on \( x \). Thus the abstract time of the globally most recent abstract write, as given by Definition 5.18, is returned.

Lemma 5.27 (Soundness of \textsc{Read}):

Assuming that \( \bar{x} \) contains a safe write history at \( t \) (Definition 5.19), a safe value for \( x \) at \( t \) as seen by thread \( T \) (Definition 5.20) is given by \( \textsc{read}(\bar{x}, x, T, t) \).

\textbf{Proof.} The proof amounts to showing that \( \text{\textsc{read}}(\bar{x}, x, T, t) \) is an upper bound to the values of the writes given by Definition 5.20; i.e., to show that all writes given by Definition 5.20 are included in \( \bar{W} \).

On line 4, the new variable state, \( \bar{x}' \), is defined to contain all writes, \( (\bar{v}', \bar{t}') \in ((\bar{x} x) T) \), such that \( \bar{t}' \not\subset \bar{t} ', \) for each \( T' \in \text{\textsc{Thrd}} \setminus \{T\} \). On lines 9–11, the writes, \( (\bar{v}', \bar{t}') \in ((\bar{x}' x) T') \), for all \( T' \in \text{\textsc{Thrd}} \setminus \{T\} \), such that \( \bar{t}' \not\subset \bar{t}, \) are extracted (i.e., identified and removed) from \( ((\bar{x}' x) T') \) and put in the set \( \bar{W} \). Thus, \( \bar{W} \) contains all the writes specified by 1 in Definition 5.20.

On line 6, \( ((\bar{x}' x) T) \) is defined to contain all writes, \( (\bar{v}', \bar{t}') \in ((\bar{x} x) T) \), such that \( \min(\gamma_{t'}(\bar{t})) \geq \min(\gamma_{t'}(\bar{t}')), \) and \( ((\bar{x}' x) T') \), for each \( T' \in \text{\textsc{Thrd}} \setminus \{T\} \), now contains all the writes, \( (\bar{v}', \bar{t}') \in ((\bar{x} x) T') \), such that \( \bar{t}' \not\subset \bar{t}, \).

On line 13, the abstract time of the (global) most recent abstract write on \( x \) among all threads, i.e., the most recent write in \( \bigcup\{((\bar{x}' x) T') \mid T' \in \text{\textsc{Thrd}}\} \), is determined (Lemma 5.26), while at line 16, the abstract time of the most recent abstract write for each thread is determined (Lemma 5.25). If the abstract time of the most recent abstract write for a thread overlaps with the abstract time of the global most recent abstract write, then all abstract writes overlapping in abstract time with the most recent abstract write for that thread are added to \( \bar{W} \) (line 17). Thus, \( \bar{W} \) now also contains (at least) all the abstract writes specified by 2 in Definition 5.20.

Finally, on line 20, the least upper bound of the values of the writes in \( \bar{W} \) is determined. On the next line, it is returned if \( \bar{W} \neq \emptyset \). If \( \bar{W} = \emptyset \), then \( [-\infty, \infty] \) is returned, which trivially is a safe approximation of the corresponding value read (i.e., \( v \in \gamma_{int}([-\infty, \infty]) \)) in the concrete case (cf. Table 4.2).
An illustration of the timestamps of the writes that have occurred on \( x \) in \( T_1 \) and \( T_2 \), as given by some abstract variable store, \( \tilde{x} \), that must be considered by \( \text{READ}(\tilde{x}, x, T_1, \tilde{t}_1) \) (lines with arrow heads pointing left) and \( \text{READ}(\tilde{x}, x, T_2, \tilde{t}_2) \) (lines with arrow heads pointing right) as explained above is given in Figure 5.10. The returned value, \( \tilde{v} \), is the least upper bound of the values of the considered writes (note that these values are not shown in the figure).

Consider the first read operation, \( \text{READ}(\tilde{x}, x, T_1, \tilde{t}_1) \). The writes that fall into category 1 of Definition 5.20, presented on page 88, are \( \tilde{w}_2^1 \) and \( \tilde{w}_3^2 \). The writes that fall into category 2 of Definition 5.20 are \( \tilde{w}_1^1 \) and \( \tilde{w}_1^2 \) (note that \( \gamma_{mrw} \) is the timestamp of \( \tilde{w}_1^1 \)).

Next consider the second read operation, \( \text{READ}(\tilde{x}, x, T_2, \tilde{t}_2) \). The writes that fall into category 1 of Definition 5.20 are \( \tilde{w}_3^2 \), \( \tilde{w}_4^2 \) and \( \tilde{w}_5^2 \). The writes that fall into category 2 of Definition 5.20 are \( \tilde{w}_1^1 \) and \( \tilde{w}_3^1 \) (note that \( \gamma_{mrw} \) is the timestamp of \( \tilde{w}_3^1 \)).

Note that all writes which are not labeled (and do not have arrow-heads) in the figure have timestamps such that they fall outside both categories in Definition 5.20, for both read operations. The writes included in category 1 have timestamps such that it is not possible to determine whether or not they have occurred before the read operation takes place; they might have. The writes included in category 2 have timestamps such that they definitely precede the read operation. It is however not possible to determine in which order the writes in this category have occurred, though.

Since \( \text{READ}(\tilde{x}, x, T, \tilde{t}) \) discards writes from thread \( T' \in \text{Thrd} \) that are too old to be valid at abstract time \( \tilde{t} \) for its returned value, and since time is assumed to never progress negatively (i.e., backwards; cf. Lemma 4.2 and Assumption 5.51 that will be made in Section 5.8 on page 114), the discarded writes can safely be removed from \( ((\tilde{x}, x) T') \). \text{TRIM}, defined in Algorithm 5.9, safely (Lemma 5.28) removes the outdated writes from \( ((\tilde{x}, x) T') \) for all \( T' \in \text{Thrd} \). Thus, \text{TRIM} can be used to lower the space complexity of the analysis. Note that \( \text{SPLITSET}(\tilde{W}, \tilde{t}) \), as defined in Algorithm 5.10, is used to split a set of
writes into two sets where the first set contains all writes, \((\bar{v}, \bar{t}')\), such that \(\bar{t}' \cap \bar{t} \neq \bar{t} \), and the second set contains all other writes.

**Algorithm 5.9** Trim variable state

1. \(\textbf{function} \ \text{TRIM}((\tilde{x}, \bar{t}))\)
2. \(\tilde{x}' \leftarrow \bot_{\text{var}}\)
3. \(\tilde{x}'' \leftarrow \bot_{\text{var}}\)
4. \(\textbf{for all } x \in \text{Var} \textbf{ do}\)
5. \(\langle [F_T]_{T \in \text{Thrd}} \rangle \leftarrow \langle [\emptyset]_{T \in \text{Thrd}} \rangle\)
6. \(\langle [O_T]_{T \in \text{Thrd}} \rangle \leftarrow \langle [\emptyset]_{T \in \text{Thrd}} \rangle\)
7. \(\langle [N_T]_{T \in \text{Thrd}} \rangle \leftarrow \langle [\emptyset]_{T \in \text{Thrd}} \rangle\)
8. \(\textbf{for all } T \in \text{Thrd} \textbf{ do}\)
9. \(F_T \leftarrow \{(\bar{v}, \bar{t}') \in (\tilde{x} x) T \mid \bar{t} \ll \bar{t}'\}\)
10. \((O_T, N_T) \leftarrow \text{SPLIT}((\tilde{x} x) T, \bar{t})\)
11. \((\tilde{x} x) T) \leftarrow N_T \setminus F_T\)
12. \(\textbf{end for}\)
13. \(\bar{t}_{\text{max}} \leftarrow \text{MOSTRECENTWRITE}(\tilde{x}', x)\)
14. \(\textbf{for all } T \in \text{Thrd} \textbf{ do}\)
15. \(\bar{W}_T \leftarrow 0\)
16. \(\bar{t}_{\text{max}} \leftarrow \text{MOSTRECENTWRITE}(\tilde{x} x) T)\)
17. \(\textbf{if } \bar{t}_{\text{max}} \cap_T \bar{t}_{\text{max}} = \bot_T \land F_T = \emptyset \land O_T = \emptyset \textbf{ then}\)
18. \(\bar{W}_T \leftarrow \{(\bot_{\text{var}}, \bot_T)\}\)
19. \(\textbf{else}\)
20. \(\bar{W}_T \leftarrow \{(\bar{v}, \bar{t}') \in (\tilde{x} x) T) \mid \bar{t}' \cap_T \bar{t}_{\text{max}} \neq \bot_T \land \bar{t}_{\text{max}} \cap_T \bar{t}_{\text{max}} \neq \bot_T\}\)
21. \(\textbf{end if}\)
22. \((\tilde{x} x) T) \leftarrow F_T \cup O_T \cup \bar{W}_T\)
23. \(\textbf{end for}\)
24. \(\textbf{return } \tilde{x}''\)
25. \(\textbf{end function}\)

**Lemma 5.28 (Soundness of TRIM):**

*If \(\tilde{x}\) contains a safe write history at abstract time \(\bar{t}\) (cf. Definition 5.19), then so does \(\text{TRIM}(\tilde{x}, \bar{t})\).*

**Proof.** Given that \(\tilde{x}\) is safe, it must be shown that, for any variable, \(x \in \text{Var}\), and any thread, \(T \in \text{Thrd}\), \(((\text{TRIM}(\tilde{x}, \bar{t}) x) T)\) contains at least (cf. Definition 5.19)

1. all writes, \((\bar{v}, \bar{t}')\), of \(((\tilde{x} x) T)\) such that \(\bar{t}' \bar{\prec} \bar{t} \land \bar{t} \bar{\prec} \bar{t}'\), and
Algorithm 5.10 Split set of writes

1: function \text{SPLITSET}(\tilde{W}, \tilde{t})
2: \quad O \leftarrow \{ (\tilde{v}, \tilde{t}') \in \tilde{W} \mid \tilde{t}' \preceq \tilde{t} \wedge \tilde{t}' \not\succeq \tilde{t} \}
3: \quad N \leftarrow \{ (\tilde{v}, \tilde{t}') \in \tilde{W} \mid \tilde{t}' \prec \tilde{t} \lor \tilde{t}' \preceq \tilde{t} \}
4: \quad \text{return } (O, N)
5: \quad \text{end function}

2. any write, \((\tilde{v}, \tilde{t}')\), of \(((\tilde{x}, x) T)\) such that \(\tilde{t}' \prec \tilde{t}\), if \(\tilde{t}' \cap \tilde{t}'_{\text{mrw}} \neq \tilde{t}_{\text{rs}}\), where \(\tilde{t}'_{\text{mrw}}\) is the abstract time of the globally most recent abstract write of the writes preceding \(\tilde{t}\),

or,

3. \((\tilde{t}_{\text{val}}, \tilde{t}_{\text{rs}})\), if there are no writes fitting the definition of the previous two categories (e.g., if all writes made by \(T\) are outdated or no writes have occurred by \(T\) on \(x\); i.e., if \(((\tilde{x}, x) T) = \{ (\tilde{t}_{\text{val}}, \tilde{t}_{\text{rs}}) \}\)).

Before advancing to the proof procedure, note that \(\lnot(\tilde{t}' \not\succeq \tilde{t} \wedge \tilde{t}' \not\succeq \tilde{t}')\) whenever \(\tilde{t}\) or \(\tilde{t}'\) is \(\tilde{t}_{\text{rs}}\) or \(\tilde{t}_{\text{rs}}\). If they are not, note that (it is implicitly assumed that \(\text{Time} = \text{Intv}\)):

\[
\tilde{t}' \not\succeq \tilde{t} \wedge \tilde{t}' \not\succeq \tilde{t} \quad \overset{\text{Def.} \ 5.14}{\iff} \quad \max(\gamma_i(\tilde{t}')) < \min(\gamma_i(\tilde{t})) \land \max(\gamma_i(\tilde{t})) < \min(\gamma_i(\tilde{t}'))
\]

\[
\overset{\text{calc.}}{\iff} \quad \max(\gamma_i(\tilde{t}')) \geq \min(\gamma_i(\tilde{t})) \land \max(\gamma_i(\tilde{t})) \geq \min(\gamma_i(\tilde{t}'))
\]

\[
\overset{\text{calc.}}{\iff} \quad \min(\{ \max(\gamma_i(\tilde{t})), \max(\gamma_i(\tilde{t}')) \}) \geq \max(\{ \min(\gamma_i(\tilde{t})), \min(\gamma_i(\tilde{t}')) \})
\]

\[
\overset{\text{Def.} \ 3.34}{\iff} \quad \tilde{t} = \tilde{t}_{\text{rs}} \quad \overset{\text{def.} \ 5.14}{\iff} \quad \tilde{t}' \neq \tilde{t}_{\text{rs}}
\]

Now, assume that \(\tilde{x}\) contains a safe write history. The structure of the algorithm gives that for each \(x \in \text{Var}\):

- For each thread, \(T \in \text{Thrd}\), the set \(F_T\) contains the writes, \((\tilde{v}, \tilde{t}')\), by \(T\) on \(x\) such that \(\tilde{t} \preceq \tilde{t}'\); i.e., writes that occur after \(\tilde{t}\). Note that this captures all writes, \((\tilde{v}, \tilde{t}')\), such that \(\tilde{t}' = \tilde{t}_{\text{rs}}\) as long as \(\tilde{t} \neq \tilde{t}_{\text{rs}}\).

- For each thread, \(T \in \text{Thrd}\), the set \(O_T\) contains the writes, \((\tilde{v}, \tilde{t}')\), by \(T\) on \(x\) such that \(\tilde{t}' \not\preceq \tilde{t} \wedge \tilde{t} \not\preceq \tilde{t}'\).

- For each thread, \(T \in \text{Thrd}\), the set \(N_T\) contains the writes, \((\tilde{v}, \tilde{t}')\), by \(T\) on \(x\) such that \(\tilde{t}' \not\preceq \tilde{t} \lor \tilde{t} \not\preceq \tilde{t}'\). Note that this captures all writes, \((\tilde{v}, \tilde{t}')\), such that \(\tilde{t}' = \tilde{t}_{\text{rs}}\) or \(\tilde{t}' = \tilde{t}_{\text{rs}}\).
• \( \bar{p}_{\text{mrw}} \) is determined from \( \bar{x}' \), for which all writes, \((\bar{v}, \bar{t}') \in ((\bar{x}' x) T)\), on \( x \) by each thread, \( T \in \text{Thrd} \), are such that \( \bar{t}' \preceq \bar{t} \).

• For each thread \( T \in \text{Thrd} \), \( \bar{W}_T = \{ (\bar{l}_{\text{val}}, \bar{l}_{\text{t}}) \} \) whenever \( \bar{p}_{\text{mrw}} \cap l \neq \emptyset \) if \( \bar{p}_{\text{mrw}} \cap l \neq \emptyset \) or \( \bar{p}_{\text{mrw}} \cap l = \emptyset \), where \( \bar{p}_{\text{mrw}} \) is the abstract time of the most recent write of the writes, \((\bar{v}, \bar{t}') \in ((\bar{x}' x) T)\); i.e., the writes, \((\bar{v}, \bar{t}') \in ((\bar{x}' x) T)\), such that \( \bar{t}' \preceq \bar{t} \).

Assume that \( \bar{p}_{\text{mrw}} \cap l \not\subseteq \bar{l}_{\text{t}} \vee F_T \neq \emptyset \), then \( ((\bar{x}' x) T) \) contains all writes, \((\bar{v}, \bar{t}')\), such that \( \bar{t}' \preceq \bar{t} \). The top element, \( \bar{t} \), if \( \bar{t}' \preceq \bar{t} \) are present in \( \bar{x}' \); i.e., they are not trimmed away from \( \bar{x} \).

Next, assume that \( \bar{p}_{\text{mrw}} \cap l \not\subseteq \bar{l}_{\text{t}} \wedge F_T = \emptyset \), then \(((\bar{x}'' x) T)\) contains all writes, \((\bar{v}, \bar{t}')\), occurring after \( \bar{t} \) (i.e., \( \bar{t} \preceq \bar{t} \)) are present in \( \bar{x}'' \); i.e., they are not trimmed away from \( \bar{x} \).

Thus, \( \bar{x}'' \) (and hence \( \text{TRIM}((\bar{x}, \bar{t})) \)) contains a safe write history for all variables, \( x \in \text{Var} \) and threads, \( T \in \text{Thrd} \).

5.6 Abstract Lock States

In this section, a Galois connection, \( (\alpha_{\text{lock}}, \gamma_{\text{lock}}) \), between the concrete domain \( \mathcal{P}(\text{Lck} \rightarrow (\text{Lck}_{\text{sst}} \times \text{Thrd}_{\perp} \times \text{Time} \times \text{Thrd}_{\perp} \times \text{Time})) \) and the abstract domain \( (\text{Lck} \rightarrow (\text{Lck}_{\text{sst}} \times \text{Thrd}_{\perp} \times \text{Time} \times \text{Thrd}_{\perp} \times \text{Time})) \cup \{ \bar{l}_{\text{lock}}, \bar{\bar{l}}_{\text{lock}} \} \supseteq \emptyset \), for lock states, will be defined. The definitions of \( \gamma_{\text{lock}} \) and \( \alpha_{\text{lock}} \) are presented in Definitions 5.29 and 5.30, respectively.

**Definition 5.29 (Concretization of an abstract lock state):**

\[
\begin{align*}
\gamma_{\text{lock}}(\bar{\bar{l}}_{\text{lock}}) &= \text{Lck} \\
\gamma_{\text{lock}}(\bar{l}_{\text{lock}}) &= \emptyset \\
\gamma_{\text{lock}}(\bar{l}) &= \gamma_{\text{lock}}(\lambda \text{lck} \in \text{Lck}.(u_{\text{lck}}, T_{\text{lck}}, \bar{t}_{\text{lck}}, T'_{\text{lck}}, i_{\text{lck}})) \\
&= \{ \bar{l} \in \text{Lck} \rightarrow (\text{Lck}_{\text{sst}} \times \text{Thrd}_{\perp} \times \text{Time} \times \text{Thrd}_{\perp} \times \text{Time}) \mid \forall \text{lck} \in \text{Lck} : \exists t \in \gamma_{\gamma}(i_{\text{lck}}) : \exists t' \in \gamma_{\gamma}(i'_{\text{lck}}) : \bar{l} \text{lck} = (u_{\text{lck}}, T_{\text{lck}}, t, T'_{\text{lck}}, t') \}
\end{align*}
\]
Table 5.11: Definition of STT, OWN, DL, POWN and REL – abstract versions of STT, OWN, DL, POWN and REL.

<table>
<thead>
<tr>
<th>Definition 5.30 (Abstraction of a set of lock states):</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \alpha_{lock}(\mathbb{I}<em>x) = \bigcap</em>{lock} { \mathbb{I}</td>
</tr>
</tbody>
</table>
| \( \mathbb{I} \) is the bottom element, \( \mathbb{I}_{lock} \), if \( \forall \text{lock} \in \text{Lck} : \mathbb{I}\text{lock} = (u, T, \mathbb{I}_r, T', \mathbb{I}_r) \) for some lock state, \( u \in \{\text{unlocked, locked}\} \), owner, \( T \in \text{Thrd} \), and previous owner, \( T' \in \text{Thrd} \). The top element, \( \mathbb{I}_{lock} \), identifies the mappings, \( \mathbb{I} \), such that \( \forall \text{lock} \in \text{Lck} : \mathbb{I}\text{lock} = (u, T, \mathbb{I}_r, T', \mathbb{I}_r) \), for any lock state, \( u \in \{\text{unlocked, locked}\} \), owner, \( T \in \text{Thrd} \), and previous owner, \( T' \in \text{Thrd} \).

The partial order, \( \sqsubseteq_{lock} \), greatest lower bound, \( \bigcap_{lock} \), and least upper bound, \( \bigcup_{lock} \), for abstract lock states follow naturally from Definitions 3.26, 3.27 and 3.28 and are presented in Definitions 5.31, 5.32 and 5.33, respectively. Note that STT, OWN, DL, POWN and REL, as defined in Table 5.11, are the abstract versions of the masking functions STT, OWN, DL, POWN and REL (defined in Table 4.7 on page 56), respectively.
Definition 5.31 (Partial order of abstract lock states):

\[
\begin{align*}
\top \sqsubseteq \text{lock} & \Rightarrow \top \overset{\text{lock}}{\sqsubseteq} \\
\bot & \sqsubseteq \text{lock} \\
\top & \sqsubseteq \text{lock} \\
\top & \sqsubseteq \text{lock} \\
\top & \sqsubseteq \text{lock}
\end{align*}
\]

\[\forall \text{lock} \in \text{Lck} : (S\Rightarrow\top \text{lock}) = S\Rightarrow\top \text{lock} \land \\
\text{OWN}(\top \text{lock}) = \text{OWN}(\top \text{lock}) \land \\
\text{Dl}(\top \text{lock}) \sqsubseteq_t \text{Dl}(\top \text{lock}) \land \\
\text{Pown}(\top \text{lock}) = \text{Pown}(\top \text{lock}) \land \\
\text{Rel}(\top \text{lock}) \sqsubseteq_t \text{Rel}(\top \text{lock})
\]

Definition 5.32 (Greatest lower bound of abstract lock states):

\[
\begin{align*}
\top \sqcap \text{lock} & \Rightarrow \top \overset{\text{lock}}{\sqcap} \\
\bot & \sqcap \text{lock} \\
\top & \sqcap \text{lock} \\
\top & \sqcap \text{lock}
\end{align*}
\]

\[\forall \text{lock} \in \text{Lck} : (u_{\text{lock}}^{\text{lck}} \land T_{\text{lck}}^{\text{lck}} \land t_{\text{lck}}^{\text{lck}} \land T_{\text{lck}}^{\text{lck}} \land t_{\text{lck}}^{\text{lck}}) \Rightarrow \text{lock}
\]

\[\lambda \text{lock} \in \text{Lck}.
\]

\[
\begin{align*}
\lambda \text{lock} & \in \text{Lck}. \quad \text{if } \forall \text{lock} \in \text{Lck} :
\end{align*}
\]

\[
\begin{align*}
(u_{\text{lock}}^{\text{lck}}, T_{\text{lck}}^{\text{lck}}, t_{\text{lck}}^{\text{lck}}, T_{\text{lck}}^{\text{lck}}, t_{\text{lck}}^{\text{lck}}) = \text{lock}
\end{align*}
\]

\[\bot \sqcup \text{lock} \Rightarrow \bot \overset{\text{lock}}{\sqcup} \\
\top & \sqcup \text{lock} \\
\top & \sqcup \text{lock} \\
\top & \sqcup \text{lock} \\
\top & \sqcup \text{lock}
\]

\[\lambda \text{lock} \in \text{Lck}.
\]

\[
\begin{align*}
\lambda \text{lock} & \in \text{Lck}. \quad \text{if } \forall \text{lock} \in \text{Lck} :
\end{align*}
\]

\[
\begin{align*}
(u_{\text{lock}}^{\text{lck}}, T_{\text{lck}}^{\text{lck}}, t_{\text{lck}}^{\text{lck}}, T_{\text{lck}}^{\text{lck}}, t_{\text{lck}}^{\text{lck}}) = \text{lock}
\end{align*}
\]

\[\top \overset{\text{lock}}{\sqcup} \\
\top & \overset{\text{lock}}{\sqcup} \\
\top & \overset{\text{lock}}{\sqcup} \\
\top & \overset{\text{lock}}{\sqcup} \\
\top & \overset{\text{lock}}{\sqcup}
\]

Definition 5.33 (Least upper bound of abstract lock states):

\[
\begin{align*}
\top & \sqcup \text{lock} \Rightarrow \top \overset{\text{lock}}{\sqcup} \\
\bot & \sqcup \text{lock} \\
\top & \sqcup \text{lock} \\
\top & \sqcup \text{lock} \\
\top & \sqcup \text{lock}
\end{align*}
\]

\[\lambda \text{lock} \in \text{Lck}.
\]

\[
\begin{align*}
\lambda \text{lock} & \in \text{Lck}. \quad \text{if } \forall \text{lock} \in \text{Lck} :
\end{align*}
\]

\[
\begin{align*}
(u_{\text{lock}}^{\text{lck}}, T_{\text{lck}}^{\text{lck}}, t_{\text{lck}}^{\text{lck}}, T_{\text{lck}}^{\text{lck}}, t_{\text{lck}}^{\text{lck}}) = \text{lock}
\end{align*}
\]

\[\bot \sqcap \text{lock} \Rightarrow \bot \overset{\text{lock}}{\sqcap} \\
\top & \sqcap \text{lock} \\
\top & \sqcap \text{lock} \\
\top & \sqcap \text{lock} \\
\top & \sqcap \text{lock}
\]

\[\lambda \text{lock} \in \text{Lck}.
\]

\[
\begin{align*}
\lambda \text{lock} & \in \text{Lck}. \quad \text{if } \forall \text{lock} \in \text{Lck} :
\end{align*}
\]

\[
\begin{align*}
(u_{\text{lock}}^{\text{lck}}, T_{\text{lck}}^{\text{lck}}, t_{\text{lck}}^{\text{lck}}, T_{\text{lck}}^{\text{lck}}, t_{\text{lck}}^{\text{lck}}) = \text{lock}
\end{align*}
\]
Since \( \gamma_{\text{lock}} \) is monotone (Lemma 5.34), it is easily established that \( \langle \alpha_{\text{lock}}, \gamma_{\text{lock}} \rangle \) is a Galois connection (Theorem 5.35).

**Lemma 5.34 (Monotonicity of \( \gamma_{\text{lock}} \))**: 
\( \gamma_{\text{lock}} \), as given by Definition 5.29, is monotone. □

**PROOF.** Assume that 1 \( \trianglerighteq_{\text{lock}} \) 1'. If 1 = 1_{\text{lock}} or 1' = 1_{\text{lock}}, then trivially, \( \gamma_{\text{lock}}(1) \subseteq \gamma_{\text{lock}}(1') \). Otherwise, assume that 1 \( \in \gamma_{\text{lock}}(1) \), 1locks = (\( u_{\text{lock}}, T_{\text{lock}}, t_{\text{lock}}, T'_{\text{lock}}, T''_{\text{lock}} \)) and 1'locks = (\( u'_{\text{lock}}, T'_{\text{lock}}, t'_{\text{lock}}, T''_{\text{lock}} \)). Since 1 \( \trianglerighteq_{\text{lock}} \) 1', it must be that \( \forall \)locks 1 (1locks = ((\( u_{\text{lock}} \land T_{\text{lock}} \land T'_{\text{lock}} \land T''_{\text{lock}} \)) = (\( u'_{\text{lock}} \land T'_{\text{lock}} \land T''_{\text{lock}} \)). But then, since 1 \( \in \gamma_{\text{lock}}(1) \) and \( \gamma_{\text{t}} \) is monotone (Theorem 3.39), it must be that 1 \( \in \gamma_{\text{lock}}(1') \). This proves the lemma. □

**Theorem 5.35 (Galois connection – Lock states)**:
\( \langle \alpha_{\text{lock}}, \gamma_{\text{lock}} \rangle \) where \( \gamma_{\text{lock}} \) and \( \alpha_{\text{lock}} \) are given by Definitions 5.29 and 5.30, respectively, is a Galois connection. □

**PROOF.** First it will be shown that \( \gamma_{\text{lock}} \) is completely multiplicative. Thus note that \( \gamma_{\text{lock}} \) is monotone (Lemma 5.34). Next observe that \( \gamma_{\text{lock}}(1_{\text{lock}}) = \text{Lck} \rightarrow (\text{Lck}_{\text{stt}} \times \text{Thrd}_{\bot} \times \text{Time} \times \text{Thrd}_{\bot} \times \text{Time}) = 1_{\text{lock}} \).

Now, assume that \( \tilde{1}, \tilde{1}' \in \text{Lck} \rightarrow (\text{Lck}_{\text{stt}} \times \text{Thrd}_{\bot} \times \text{Thrd}_{\bot} \times \text{Time}) \) are such that \( \tilde{1} \trianglerighteq_{\text{lock}} \tilde{1}' \) and \( \tilde{1}' \trianglerighteq_{\text{lock}} \tilde{1} \). From Definition 5.31, it follows that neither of \( \tilde{1} \) and \( \tilde{1}' \) can be \( 1_{\text{lock}} \) or \( 1_{\text{lock}} \). Thus, it is safe to assume that these states can be expressed as \( 1_{\text{lock}} = (u_{\text{lock}}, T_{\text{lock}}, t_{\text{lock}}, T'_{\text{lock}}, T''_{\text{lock}}) \) and \( 1'_{\text{lock}} = (u'_{\text{lock}}, T'_{\text{lock}}, t'_{\text{lock}}, T''_{\text{lock}}) \).

Based on the above assumptions, it will be shown that:
\[
\gamma_{\text{lock}}(\tilde{1} \cap_{\text{lock}} \tilde{1}') = \gamma_{\text{lock}}(\tilde{1}) \cap \gamma_{\text{lock}}(\tilde{1}')
\]

First, assume that \( \exists \text{lock} \in \text{Lck} : (u_{\text{lock}} \neq u'_{\text{lock}} \lor T_{\text{lock}} \neq T'_{\text{lock}} \lor T''_{\text{lock}} \neq T''_{\text{lock}}) \). Then, \( \tilde{1} \cap_{\text{lock}} \tilde{1}' = 1_{\text{lock}} \), and thus the L.H.S. becomes \( \gamma_{\text{lock}}(1) \cap \gamma_{\text{lock}}(1') = \gamma_{\text{lock}}(1_{\text{lock}}) = \emptyset \). The R.H.S. becomes \( \gamma_{\text{lock}}(\tilde{1}) \cap \gamma_{\text{lock}}(\tilde{1}') = \emptyset \), because it must be that \( \forall \)locks \( \in \gamma_{\text{lock}}(\tilde{1}) : \forall \gamma_{\text{lock}}(\tilde{1}') = 1 \neq 1' \) since \( \exists \text{lock} \in \text{Lck} : (u_{\text{lock}} \neq u'_{\text{lock}} \lor T_{\text{lock}} \neq T'_{\text{lock}} \lor T''_{\text{lock}} \neq T''_{\text{lock}}) \). Thus, L.H.S. = R.H.S.

Next, assume that \( \forall \text{lock} \in \text{Lck} : (u_{\text{lock}} = u'_{\text{lock}} \land T_{\text{lock}} = T''_{\text{lock}} \land T'_{\text{lock}} = T''_{\text{lock}}) \) and note that \( \langle \alpha_{\text{lock}}, \gamma_{\text{lock}} \rangle = \langle \alpha_{\text{int}}, \gamma_{\text{int}} \rangle \) is a Galois connection (Theorem 3.39). Then, \( (\tilde{1} \cap_{\text{lock}} \tilde{1}')_{\text{lock}} = (u_{\text{lock}}, T_{\text{lock}}, t_{\text{lock}} \cap_{\text{lock}} T'_{\text{lock}}, T''_{\text{lock}}) \) and thus the following can be calculated.
lysis is presented in Chapter 6). The abstract configuration, \( \alpha \), function, completely multiplicative. Then, by Lemma 3.15, it is obvious that an abstraction where

\[ \text{Lemma 3.14, the definition of this} \]

\[ \text{5.30. Thus,} \]

\[ \langle \text{C} \rangle \]

In this section, a Galois connection between the concrete and abstract domains for configurations, \( \mathcal{P} (\text{Conf}) \) and \( \text{Cônf} \), respectively, will be defined. \( \text{Cônf} \) is defined as:

\[ \text{Cônf} := (\prod_{\text{Thrd}_c} ((\{T\} \times \text{Lbl}_T \times (\text{Reg}_T \rightarrow \text{Vál}) \times \text{Tiñe})) \times \]

\[ (\text{Var} \rightarrow \text{Thrd} \rightarrow \mathcal{P}(\text{Vál} \times \text{Tiñe})) \times \]

\[ (\text{Lck} \rightarrow (\text{Lck}_{\text{stt}} \times \text{Thrd}_{\perp} \times \text{Tiñe} \times \text{Thrd}_{\perp} \times \text{Tiñe}))) \cup \]

\[ \{ \text{Cônf}, \text{Cônf} \} \]

Thus, it has been shown that \( \gamma_{\text{lock}}(\tilde{c} \cap_{\text{lock}} \tilde{c}') = \gamma_{\text{lock}}(\tilde{c}) \cap \gamma_{\text{lock}}(\tilde{c}') \). Now, all the three conditions in Lemma 3.4 are fulfilled, which means that \( \gamma_{\text{lock}} \) is completely multiplicative. Then, by Lemma 3.15, it is obvious that an abstraction function, \( \alpha \), such that \( \langle \alpha, \gamma_{\text{lock}} \rangle \) is a Galois connection can be defined. Using Lemma 3.14, the definition of this \( \alpha \) is the same as that of \( \alpha_{\text{lock}} \) in Definition 5.30. Thus, \( \langle \alpha_{\text{lock}}, \gamma_{\text{lock}} \rangle \) is a Galois connection.

5.7 Abstract Configurations

In this section, a Galois connection between the concrete and abstract domains for configurations, \( \mathcal{P} (\text{Conf}) \) and \( \text{Cônf} \), respectively, will be defined. \( \text{Cônf} \) is defined as:

\[ \gamma_{\text{lock}}(\tilde{c} \cap_{\text{lock}} \tilde{c}') = \gamma_{\text{lock}}(\tilde{c}) \cap \gamma_{\text{lock}}(\tilde{c}') \]
denoted in the same manner as concrete configurations:

\[ \tilde{c} := \langle [T, pc_T, \tilde{r}_T, \tilde{r}^a_T]_{T \in \text{Thrd}_\tilde{c}}, \tilde{x}, \tilde{l} \rangle \]

The concretization function for abstract configurations, \( \gamma_{\text{conf}} : \text{Conf} \rightarrow \mathcal{P}(\text{Conf}) \), is given by Definition 5.36.

**Definition 5.36 (Concretization of an abstract configuration):**

\[
\begin{align*}
\gamma_{\text{conf}}(\tilde{\text{conf}}) &= \text{Conf} \\
\gamma_{\text{conf}}(\tilde{\text{confs}}) &= \emptyset \\
\gamma_{\text{conf}}(\{ [T, pc_T, \tilde{r}_T, \tilde{r}^a_T]_{T \in \text{Thrd}_\tilde{c}}, \tilde{x}, \tilde{l} \}) &= \\
\{ [T, pc_T, \tilde{r}_T, \tilde{r}^a_T]_{T \in \text{Thrd}_\tilde{c}}, \tilde{x}, \tilde{l} \} \\
\text{such that} \ x \in \gamma_r(\tilde{x}) \land \tilde{x} \in \gamma_{\text{var}}(\tilde{x}) \land \tilde{l} \in \gamma_{\text{lock}}(\tilde{l})
\end{align*}
\]

The partial ordering of abstract configurations, \( \subseteq_{\text{conf}} \), follows naturally using Definition 3.26 and is given by Definition 5.37. Note that this relation cannot be directly used within the analysis since \( \subseteq_{\text{var}} \) cannot. A safe relation, \( \subseteq'_{\text{conf}} \), is obtained by replacing \( \subseteq_{\text{var}} \) with \( \subseteq'_{\text{var}} \) in the definition of \( \subseteq_{\text{conf}} \).

**Definition 5.37 (Partial ordering of two abstract configurations):**

\[
\begin{align*}
\tilde{c} &\subseteq_{\text{conf}} \tilde{\text{conf}} \\
\tilde{\text{confs}} &\subseteq_{\text{conf}} \tilde{\text{confs}} \\
\{ [T, pc_T, \tilde{r}_T, \tilde{r}^a_T]_{T \in \text{Thrd}_\tilde{c}}, \tilde{x}, \tilde{l} \} &\subseteq_{\text{conf}} \\
\{ [T, pc_T, \tilde{r}_T, \tilde{r}^a_T]_{T \in \text{Thrd}_\tilde{c}}, \tilde{x}, \tilde{l} \} &\subseteq_{\text{conf}} \\
\forall T \in \text{Thrd}_\tilde{c} : (pc_T = pc'_{\tilde{c}} \land \tilde{x} \subseteq_{\text{var}} \tilde{x}' \land \tilde{l} \subseteq_{\text{lock}} \tilde{l}' \land \text{Thrd}_\tilde{c} = \text{Thrd}_{\tilde{c}}') \\
\tilde{x} &\subseteq_{\text{var}} \tilde{x}' \land \tilde{l} \subseteq_{\text{lock}} \tilde{l}' \land \forall T \in \text{Thrd}_\tilde{c} : (pc_T = pc'_{\tilde{c}} \land \tilde{x} \subseteq_{\text{var}} \tilde{x}' \land \tilde{l} \subseteq_{\text{lock}} \tilde{l}')
\end{align*}
\]

The function \( \gamma_{\text{conf}} \) is monotone with respect to \( \subseteq_{\text{conf}} \) (Lemma 5.38).

**Lemma 5.38 (Monotonicity of \( \gamma_{\text{conf}} \)):**
The function \( \gamma_{\text{conf}} : \text{Conf} \rightarrow \mathcal{P}(\text{Conf}) \) is monotone with respect to \( \subseteq_{\text{conf}} \). I.e., if \( \tilde{c}, \tilde{c}' \in \text{Conf} \) and \( \tilde{c} \subseteq_{\text{conf}} \tilde{c}' \), then \( \gamma_{\text{conf}}(\tilde{c}) \subseteq \gamma_{\text{conf}}(\tilde{c}') \).

**Proof.** Assume that \( \tilde{c}, \tilde{c}' \in \text{Conf} \) such that \( \tilde{c} \subseteq_{\text{conf}} \tilde{c}' \). If \( \tilde{c} = \tilde{\text{confs}} \) or \( \tilde{c}' = \tilde{\text{confs}} \), the lemma holds trivially. Otherwise, \( \tilde{c} \) and \( \tilde{c}' \) can be expressed as \( \tilde{c} = \langle [T, pc_T, \tilde{r}_T, \tilde{r}^a_T]_{T \in \text{Thrd}_\tilde{c}}, \tilde{x}, \tilde{l} \rangle \) and \( \tilde{c}' = \langle [T, pc_T, \tilde{r}_T, \tilde{r}^a_T]_{T \in \text{Thrd}_\tilde{c}'}, \tilde{x}', \tilde{l}' \rangle \) respectively.
The greatest lower bound operator for abstract configurations, $\sqcap_{\text{conf}}$, follows naturally using Definition 3.27 and is given by Definition 5.39. Note that this operator cannot be directly used within the analysis since $\sqcap_{\text{var}}$ cannot. A safe operator, $\sqcap'_{\text{conf}}$, is obtained by replacing $\sqcap_{\text{var}}$ by $\sqcap'_{\text{var}}$ in the definition of $\sqcap_{\text{conf}}$.

**Definition 5.39 (Greatest lower bound for two abstract configurations):**

\[
\begin{align*}
\overline{c} \sqcap_{\text{conf}} \overline{c'} &= \overline{c} \\
\overline{c} \sqcap_{\text{conf}} \bot &= \bot \\
\langle [T, pc_T, \overline{t}_T, \overline{t}'_T] \in \text{Thrd}_T, \overline{x}, \overline{y} \rangle \sqcap_{\text{conf}} \langle [T, pc'_T, \overline{t}'_T, \overline{t}'_T] \in \text{Thrd}'_T, \overline{x}', \overline{y}' \rangle &= \\
\begin{cases}
\langle [T, pc_T, \overline{t}_T \cap \overline{t}'_T, \overline{t}'_T] \in \text{Thrd}_T, \overline{x}, \overline{y} \rangle & \text{if } \text{Thrd}_T = \text{Thrd}'_T \\
\overline{I}_{\text{conf}} \cap_{\text{reg}} \overline{I}'_{\text{conf}} & \forall T \in \text{Thrd}_c : pc_T = pc'_T \cap_{\text{var}} \overline{x}', \overline{y}' \rangle & \text{otr} \text{w}
\end{cases}
\end{align*}
\]

The least upper bound operator for abstract configurations, $\sqcup_{\text{conf}}$, follows naturally using Definition 3.28 and is given by Definition 5.40. Note that this operator cannot be directly used within the analysis since $\sqcup_{\text{var}}$ cannot. A safe operator, $\sqcup'_{\text{conf}}$, is obtained by replacing $\sqcup_{\text{var}}$ by $\sqcup'_{\text{var}}$ in the definition of $\sqcup_{\text{conf}}$. 

\[
\begin{align*}
\langle [T, pc_T, \overline{t}_T, \overline{t}'_T] \in \text{Thrd}_T, \overline{x}, \overline{y} \rangle & \sqcup_{\text{conf}} \langle [T, pc'_T, \overline{t}'_T, \overline{t}'_T] \in \text{Thrd}'_T, \overline{x}', \overline{y}' \rangle = \\
\begin{cases}
\langle [T, pc_T, \overline{t}_T \cap \overline{t}'_T, \overline{t}'_T] \in \text{Thrd}_T, \overline{x}, \overline{y} \rangle & \text{if } \text{Thrd}_T = \text{Thrd}'_T \\
\overline{I}_{\text{conf}} \cup \overline{I}'_{\text{conf}} & \forall T \in \text{Thrd}_c : pc_T = pc'_T \\
\overline{I}_{\text{conf}} \cap_{\text{reg}} \overline{I}'_{\text{conf}} & \forall T \in \text{Thrd}_c : pc_T = pc'_T \\
\overline{I}_{\text{conf}} \cap_{\text{var}} \overline{x}', \overline{y}' \rangle & \text{otr} \text{w}
\end{cases}
\end{align*}
\]
Definition 5.40 (Least upper bound for two abstract configurations):

\[
\begin{align*}
\bar{c} \sqcup_{\text{conf}} \bar{c} &= \bar{c} \\
\bar{c} \sqcup_{\text{conf}} \top_{\text{conf}} &= \top_{\text{conf}} \\
\langle T, pc_T, \bar{c} \rangle &\subseteq \text{Thrd}_{\bar{c}}: \begin{cases} 
\text{if } \text{Thrd}_{\bar{c}} =\text{Thrd}_{\bar{c}'} & \forall T \in \text{Thrd}_{\bar{c}} : pc_T = pc'_T \\
\text{otherwise} & 
\end{cases}
\end{align*}
\]

The abstraction function, \(\alpha_{\text{conf}} : \mathcal{P}(\text{Conf}) \rightarrow \text{CConf}\), is given by Definition 5.41 and \(\langle \alpha_{\text{conf}}, \gamma_{\text{conf}} \rangle\) is indeed a Galois connection (Theorem 5.42).

Definition 5.41 (Abstraction of a set of configurations):

\[
\alpha_{\text{conf}}(C) = \bigcap_{\text{conf}} \{ \bar{c} \mid C \subseteq \gamma_{\text{conf}}(\bar{c}) \}
\]

Theorem 5.42 (Galois connection – Configurations):

\(\langle \alpha_{\text{conf}}, \gamma_{\text{conf}} \rangle\), where \(\gamma_{\text{conf}}\) and \(\alpha_{\text{conf}}\) are given by Definitions 5.36 and 5.41, respectively, is a Galois connection.

Proof. First it will be shown that \(\gamma_{\text{conf}}\) is completely multiplicative. Thus note that \(\gamma_{\text{conf}}\) is monotone (Lemma 5.38). Next observe that \(\gamma_{\text{conf}}(\top_{\text{conf}}) = \text{Conf} = \top_{\text{conf}}\).

Now, assume that \(\bar{c}, \bar{c}' \in \text{CConf}\) are such that \(\bar{c} \sqsubseteq_{\text{conf}} \bar{c}' \wedge \bar{c} \not\sqsubseteq_{\text{conf}} \bar{c}'\). From Definition 5.37, it follows that neither of \(\bar{c}\) and \(\bar{c}'\) can be \(\bot_{\text{conf}}\) or \(\top_{\text{conf}}\). Thus, it is safe to assume that these configurations can be expressed as \(\bar{c} = \langle |T|, pc_T, \bar{c} \rangle\) and \(\bar{c}' = \langle |T|, pc'_T, \bar{c}' \rangle\).

Based on the above assumptions, it will be shown that:

\[
\gamma_{\text{conf}}(\bar{c} \sqcap_{\text{conf}} \bar{c}') = \gamma_{\text{conf}}(\bar{c}) \cap \gamma_{\text{conf}}(\bar{c}')
\]

First, assume that \(\text{Thrd}_{\bar{c}} \neq \text{Thrd}_{\bar{c}'} \lor \exists T \in \text{Thrd}_{\bar{c}} : pc_T \neq pc'_T\). Then, \(\bar{c} \sqcap_{\text{conf}} \bar{c}' = \bot_{\text{conf}}\), and thus the L.H.S. becomes \(\gamma_{\text{conf}}(\bar{c} \sqcap_{\text{conf}} \bar{c}') = \gamma_{\text{conf}}(\bot_{\text{conf}}) = \emptyset\).

The R.H.S. becomes \(\gamma_{\text{conf}}(\bar{c}) \cap \gamma_{\text{conf}}(\bar{c}') = \emptyset\), because it must be that \(\forall c \in \gamma_{\text{conf}}(\bar{c}) : \forall c' \in \gamma_{\text{conf}}(\bar{c}') : c \neq c'\), since \(\text{Thrd}_{\bar{c}} \neq \text{Thrd}_{\bar{c}'} \lor \exists T \in \text{Thrd}_{\bar{c}} : pc_T \neq pc'_T\). Thus, L.H.S. = R.H.S.
Next, assume that $\text{Thrd}_{\varepsilon} = \text{Thrd}_{\varepsilon'} \land \forall T \in \text{Thrd}_{\varepsilon}$ : $pc_{T} = pc'_{T}$ and note that $\langle \alpha_{t}, \gamma_{t} \rangle = \langle \alpha_{\text{int}}, \gamma_{\text{int}} \rangle$, $\langle \alpha_{\text{reg}}, \gamma_{\text{reg}} \rangle$, $\langle \alpha_{\text{var}}, \gamma_{\text{var}} \rangle$ and $\langle \alpha_{\text{lock}}, \gamma_{\text{lock}} \rangle$ are Galois connections (Theorems 3.39, 5.6, 5.11 and 5.35, respectively). Then, $\varepsilon \cap_{\text{conf}} \varepsilon' = \langle \{T, pc_{T}, \varepsilon T, \Gamma \cap_{\text{reg}} \varepsilon' T, \tilde{c}_{T}, \tilde{d}_{T} \mid T \in \text{Thrd}_{\varepsilon}, \tilde{x} \cap_{\text{var}} \tilde{x}' \cap_{\text{lock}} \tilde{l}' \cap_{\text{lock}} \tilde{p}' \rangle \cap_{\gamma_{\text{lock}}} \rangle$ and

$$\gamma_{\text{conf}}(\varepsilon \cap_{\text{conf}} \varepsilon') \overset{\text{Def. 5.36}}{=} \{ \langle [T, pc_{T}, \varepsilon T, t_{T}] \mid T \in \text{Thrd}_{\varepsilon}, \varepsilon, 0 \rangle \mid
\begin{align*}
\exists T \in \gamma_{\text{reg}}(\varepsilon T \cap_{\text{reg}} \varepsilon' T) \land t_{T} \in \gamma_{T}(\tilde{c}_{T} \cap_{\text{lock}} \tilde{d}_{T}) \land
\exists \alpha \in \gamma_{\text{var}}(\tilde{x} \cap_{\text{var}} \tilde{x}') \land \exists \text{lock}(\tilde{p} \cap_{\text{lock}} \tilde{l}) \}
\}
$$

$$\overset{\text{Lem. 3.14}}{=} \{ \langle [T, pc_{T}, \varepsilon T, t_{T}] \mid T \in \text{Thrd}_{\varepsilon}, \varepsilon, 0 \rangle \mid
\begin{align*}
\forall T \in \gamma_{\text{reg}}(\varepsilon T) \land t_{T} \in \gamma_{T}(\tilde{c}_{T}) \land
\exists \alpha \in \gamma_{\text{var}}(\tilde{x} \cap_{\text{var}} \tilde{x}') \land \exists \text{lock}(\tilde{p} \cap_{\text{lock}} \tilde{l}) \}
\}
$$

$$\overset{\text{calc.}}{=} \{ \langle [T, pc_{T}, \varepsilon T, t_{T}] \mid T \in \text{Thrd}_{\varepsilon}, \varepsilon, 0 \rangle \mid
\begin{align*}
\exists T \in \gamma_{\text{reg}}(\varepsilon T) \land t_{T} \in \gamma_{T}(\tilde{c}_{T}) \land
\exists \alpha \in \gamma_{\text{var}}(\tilde{x} \cap_{\text{var}} \tilde{x}') \land \exists \text{lock}(\tilde{p} \cap_{\text{lock}} \tilde{l}) \}
\}
$$

$$\overset{\text{Def. 5.36}}{=} \gamma_{\text{conf}}(\varepsilon) \cap \gamma_{\text{conf}}(\varepsilon')$$

Thus, it has been shown that $\gamma_{\text{conf}}(\varepsilon \cap_{\text{conf}} \varepsilon') = \gamma_{\text{conf}}(\varepsilon) \cap \gamma_{\text{conf}}(\varepsilon')$. Now, all the three conditions in Lemma 3.4 are fulfilled, which means that $\gamma_{\text{conf}}$ is completely multiplicative. Then, by Lemma 3.15, it is obvious that an abstraction function, $\alpha$, such that $\langle \alpha_{\varepsilon}, \gamma_{\varepsilon} \rangle$ is a Galois connection can be defined. Using Lemma 3.14, the definition of this $\alpha$ is the same as that of $\alpha_{\text{conf}}$ in Definition 5.41. Thus, $\langle \alpha_{\text{conf}}, \gamma_{\text{conf}} \rangle$ is a Galois connection.

An alternative approach to derive a Galois connection here could be to use Theorems 3.16, 3.17, 3.20, 3.22, 3.24, 3.25 and 3.39, but the presented Galois connection is easier to understand.

Now, consider the abstract domains, $\alpha_{\text{Conf}_{\varepsilon}}^\varepsilon \ni \alpha_{\text{Conf}_{\varepsilon}}^\varepsilon'$ and $\alpha_{\text{Conf}_{\varepsilon}}^\varepsilon \ni \alpha_{\text{Conf}_{\varepsilon}}^\varepsilon'$, which will be used for the abstract axiom transition rules as presented in Table 5.12 in Section 5.8 on page 111. These domains are defined as:

$$\alpha_{\text{Conf}_{\varepsilon}}^\varepsilon := \langle \{T\} \times \text{Lbl}_{T} \times (\text{Reg}_{T} \rightarrow \text{Väl}) \times
\begin{align*}
\text{Var} \rightarrow \text{Thrd} \rightarrow \mathcal{P}(\text{Väl} \rightarrow \text{Tiñe}) \times
\text{Lck} \rightarrow (\text{Lck}_{\varepsilon'} \times \text{Thrd}_{\varepsilon'} \times \text{Tiñe} \times \text{Thrd}_{\varepsilon'} \times \text{Tiñe}) \times
\text{Tiñe} \cup \{ \alpha_{\varepsilon'}^\varepsilon, \alpha_{\varepsilon'}^\varepsilon \}
\end{align*}
\rangle$$

$$\alpha_{\varepsilon}^\varepsilon := \langle T, pc_{T}, \tilde{c}, \tilde{x}, \tilde{l}, \tilde{p} \rangle$$
5.41. Thus, Table 5.12 in Section 5.8 on page 111. These domains are defined as:

\[ \text{axC} \circ \text{conf} \]

Now, consider the abstract domains, \( \text{axC} \circ \text{conf} \) and \( \text{axC} \circ \text{conf} \). An alternative approach to derive a Galois connection here could be to use

\[ \text{axC} \circ \text{conf} \]

which will be used for the abstract axiom transition rules as presented in

\[ \text{axC} \circ \text{conf} \]

Then, \( T \), which will be used for the abstract axiom transition rules as presented in

\[ \text{axC} \circ \text{conf} \]

Theorems 3.39, 5.6, 5.11 and 5.35, respectively. Then, \( T \) may be used for the abstract axiom transition rules as presented in

\[ \text{axC} \circ \text{conf} \]

\[ \text{axC} \circ \text{conf} \]

\[ \text{axC} \circ \text{conf} \]

\[ \text{axC} \circ \text{conf} \]

\[ \text{axC} \circ \text{conf} \]

Definition 5.44 (Concretization of an abstract axiom input configuration):

\[ \text{axC} \circ \text{conf} \]

Definition 5.45 (Abstraction of a set of axiom output configurations):

\[ \text{axC} \circ \text{conf} \]

Definition 5.46 (Concretization of an abstract axiom output configuration):

\[ \text{axC} \circ \text{conf} \]

Theorem 5.47 (Galois connection – Axiom input configurations):

\[ \text{axC} \circ \text{conf} \]

where \( \text{axC} \circ \text{conf} \) and \( \text{axC} \circ \text{conf} \) are given by Definitions 5.43 and 5.44, respectively, is a Galois connection.

\[ \text{axC} \circ \text{conf} \]
PROOF. Similar to the proof of Theorem 5.42. ■

Theorem 5.48 (Galois connection – Axiom output configurations):
\langle ax^{\text{out}}_T, ax^{\text{out}}_T \rangle, where \(ax^{\text{out}}_T\) and \(ax^{\text{out}}_T\) are given by Definitions 5.45 and 5.46, respectively, is a Galois connection.

PROOF. Similar to the proof of Theorem 5.42. ■

5.8 Abstract Semantics

The abstract transition rules for axiom statements in Table 5.12 are safe approximations of the rules in Table 4.2 with respect to Definition 5.49 (Lemma 5.50).

Definition 5.49 (Soundness of the abstract axiom transition relation):
Assuming that \(\approx\) contains a safe write history (cf. Definition 5.19), the transition relation \(\overset{\sim}{\rightarrow}\) is a safe approximation of \(\rightarrow\) iff

\[
\forall ax^{\text{in}}_T \in ax^{\text{Conf}}^{\text{in}}_T : \forall ax^{\text{in}}_T \in ax^{\text{Conf}}^{\text{in}}(ax^{\text{in}}_T) : \forall ax^{\text{out}}_T \in ax^{\text{Conf}}^{\text{out}}_T : (ax^{\text{in}}_T \overset{\sim}{\rightarrow} ax^{\text{out}}_T \Rightarrow \exists ax^{\text{out}}_T \in ax^{\text{Conf}}^{\text{out}}_T : (ax^{\text{in}}_T \overset{\sim}{\rightarrow} ax^{\text{out}}_T \wedge ax^{\text{out}}_T \in ax^{\text{Conf}}^{\text{out}}(ax^{\text{out}}_T)))
\]

where \(ax^{\text{in}}_T\) is generated (cf. Table 4.5) from a valid configuration (cf. Definition 4.4); i.e., the lock state is valid with respect to the accumulated time of the given thread.

PROOF. This proof will be conducted by showing for each defined transition that it is safe according to Definition 5.49.

Assume that \(ax^{\text{in}}_T @ \langle T, pc, \hat{r}, x, l, t \rangle \in ax^{\text{Conf}}^{\text{in}}_T\) and \(ax^{\text{in}}_T @ \langle T, pc, \hat{r}, x, l, t \rangle \in ax^{\text{Conf}}^{\text{in}}_T\), such that \(ax^{\text{in}}_T \in ax^{\text{Conf}}^{\text{in}}(ax^{\text{in}}_T)\), that \(l\) is valid with respect to \(t\) and that \(x\) contains a safe write history. Now consider each defined axiom statement.

1. Assume that STM(T, pc) = [halt]_{pc}. From the concrete semantics, it must be that \(ax^{\text{in}}_T \overset{\rightarrow}{\rightarrow} ax^{\text{out}}_T\), where \(ax^{\text{out}}_T = \langle pc, \hat{r}, x, l \rangle\). Choose \(ax^{\text{out}}_T\) so that \(ax^{\text{in}}_T \overset{\sim}{\rightarrow} ax^{\text{out}}_T\), i.e., \(ax^{\text{out}}_T = \langle pc, \hat{r}, x, l \rangle\). Thus, \(ax^{\text{out}}_T \in ax^{\text{Conf}}^{\text{out}}(ax^{\text{out}}_T)\).
Table 5.12: $\langle T, pc, \bar{r}, \bar{x}, \bar{I}, \bar{i} \rangle \xrightarrow{\alpha} \langle pc', \bar{r}', \bar{x}', \bar{I}' \rangle$, semantics of abstract axiom transitions.

<table>
<thead>
<tr>
<th>STM($T, pc$)</th>
<th>$\langle pc', \bar{r}', \bar{x}', \bar{I}' \rangle$</th>
<th>If</th>
</tr>
</thead>
<tbody>
<tr>
<td>$[\text{halt}]^{pc}$</td>
<td>$\langle pc, \bar{r}, \bar{x}, \bar{I} \rangle$</td>
<td></td>
</tr>
<tr>
<td>$[\text{skip}]^{pc}$</td>
<td>$\langle pc + 1, \bar{r}, \bar{x}, \bar{I} \rangle$</td>
<td></td>
</tr>
<tr>
<td>$[r := a]^{pc}$</td>
<td>$\langle pc + 1, \bar{r} \cdot [r \mapsto \mathcal{A}[a], \bar{x}, \bar{I}] \rangle$</td>
<td>$\mathcal{A}[b] \bar{r} \not= \bot_{reg}$</td>
</tr>
<tr>
<td>$[\text{if } b \text{ goto } l]^{pc}$</td>
<td>$\langle pc + 1, \bar{r}, \bar{x}, \bar{I} \rangle$</td>
<td>$\mathcal{A}[b] \bar{r} \not= \bot_{reg}$</td>
</tr>
<tr>
<td>$[\text{if } b \text{ goto } l]^{pc}$</td>
<td>$\langle pc + 1, \bar{r} \cdot [l \mapsto \mathcal{B}[b] \bar{r}, \bar{x}, \bar{I}] \rangle$</td>
<td>OWN($\bar{lck}$) = $T \land$</td>
</tr>
<tr>
<td>$[\text{store } r \text{ to } x]^{pc}$</td>
<td>$\langle pc + 1, \bar{r}, \bar{x}, \bar{I} \rangle$</td>
<td>(STM($\bar{lck}$) = unlocked $\Rightarrow$</td>
</tr>
<tr>
<td>$[\text{load } r \text{ from } x]^{pc}$</td>
<td>$\langle pc + 1, \bar{r} \cdot [r \mapsto \text{READ}(\bar{x}, x, T, \bar{i}), \bar{I}] \rangle$</td>
<td>(STM($\bar{lck}$) = unlocked $\land$</td>
</tr>
<tr>
<td>$[\text{lock } lck]^{pc}$</td>
<td>$\langle pc + 1, \bar{r}, \bar{x}, \bar{I} \rangle$</td>
<td>OWN($\bar{lck}$) = $T \land$</td>
</tr>
<tr>
<td>$[\text{unlock } lck]^{pc}$</td>
<td>$\langle pc + 1, \bar{r}, \bar{x}, \bar{I} \rangle$</td>
<td>STM($\bar{lck}$) = unlocked</td>
</tr>
<tr>
<td>$[\text{unlock } lck]^{pc}$</td>
<td>$\langle pc + 1, \bar{r}, \bar{x}, \bar{I} \rangle$</td>
<td>OWN($\bar{lck}$) = unlocked</td>
</tr>
<tr>
<td>$[\text{unlock } lck]^{pc}$</td>
<td>$\langle pc + 1, \bar{r}, \bar{x}, \bar{I} \rangle$</td>
<td>STM($\bar{lck}$) = unlocked</td>
</tr>
</tbody>
</table>
2. Assume that $\text{STM}(T, pc) = [\text{skip}]^{pc}$. From the concrete semantics, it must be that $\text{axcin}_T \xrightarrow{\text{ax}} \text{axcout}_T$, where $\text{axcout}_T = \langle pc + 1, \bar{r}, \bar{x}, \bar{l} \rangle$. Choose $\text{axcout}_T$ so that $\text{axcin}_T \xrightarrow{\text{ax}} \text{axcout}_T$, i.e., $\text{axcout}_T = \langle pc + 1, \bar{r}, \bar{x}, \bar{l} \rangle$. Thus, $\text{axcout}_T \subseteq \text{axout}_T(\text{axcout}_T)$. 

3. Assume that $\text{STM}(T, pc) = [r := a]^{pc}$. From the concrete semantics, it must be that $\text{axcin}_T \xrightarrow{\text{ax}} \text{axcout}_T$, where $\text{axcout}_T = \langle pc + 1, r \mid r \rightarrow \mathcal{A}[a]^{\bar{r}}, \bar{x}, \bar{l} \rangle$. Choose $\text{axcout}_T$ so that $\text{axcin}_T \xrightarrow{\text{ax}} \text{axcout}_T$, i.e., $\text{axcout}_T = \langle pc + 1, \bar{r} \mid r \rightarrow \mathcal{A}[a]^{\bar{r}}, \bar{x}, \bar{l} \rangle$. Since $\mathcal{A}$ is safely induced from $\mathcal{A}$ (see Section 5.3), it must be that $\mathcal{A}[a]^{\bar{r}} \in \mathcal{Y}_{\bar{r}}(\mathcal{A}[a]^{\bar{r}})$, and hence, $\bar{r} \mid r \rightarrow \mathcal{A}[a]^{\bar{r}} \in \mathcal{Y}_{\bar{r}}(\mathcal{A}[a]^{\bar{r}})$. Thus, $\text{axcout}_T \subseteq \text{axout}_T(\text{axcout}_T)$. 

4. Assume that $\text{STM}(T, pc) = [\text{if } b \text{ goto } l]^{pc}$. Then two cases must be considered.

(a) In the first case, $\mathcal{B}[b]^{\bar{r}}$. This means that $\text{axcin}_T \xrightarrow{\text{ax}} \text{axcout}_T$, where $\text{axcout}_T = \langle l, \bar{r}, \bar{x}, \bar{l} \rangle$. Now, choose $\text{axcout}_T$ so that $\text{axcin}_T \xrightarrow{\text{ax}} \text{axcout}_T$ by the corresponding branch (i.e., $\mathcal{B}[b]^{\bar{r}} \neq \top_{\text{reg}}$); i.e., $\text{axcout}_T = \langle l, \mathcal{B}[b]^{\bar{r}}, \bar{x}, \bar{l} \rangle$. Since $\mathcal{B}$ is safely induced from $\mathcal{B}$ (see Section 5.4), it must be that $\mathcal{B}[b]^{\bar{r}} \neq \top_{\text{reg}}$ and that $\text{axout}_T(\text{axcout}_T)$ contains, at least, all cases where $\mathcal{B}[b]^{\bar{r}}$. Thus, it must be the case that $\text{axcout}_T \subseteq \text{axout}_T(\text{axcout}_T)$. 

(b) In the second case, $\neg\mathcal{B}[b]^{\bar{r}}$. This means that $\text{axcin}_T \xrightarrow{\text{ax}} \text{axcout}_T$, where $\text{axcout}_T = \langle pc + 1, \bar{r}, \bar{x}, \bar{l} \rangle$. Now, choose $\text{axcout}_T$ so that $\text{axcin}_T \xrightarrow{\text{ax}} \text{axcout}_T$ by the corresponding branch (i.e., $\mathcal{B}[b]^{\bar{r}} \neq \top_{\text{reg}}$); i.e., $\text{axcout}_T = \langle pc + 1, \mathcal{B}[b]^{\bar{r}}, \bar{x}, \bar{l} \rangle$. Since $\mathcal{B}$ is safely induced from $\mathcal{B}$ (see Section 5.4), it must be that $\mathcal{B}[b]^{\bar{r}} \neq \top_{\text{reg}}$ and that $\text{axout}_T(\text{axcout}_T)$ contains, at least, all cases where $\neg\mathcal{B}[b]^{\bar{r}}$. Thus, it must be the case that $\text{axcout}_T \subseteq \text{axout}_T(\text{axcout}_T)$. 

5. Assume that $\text{STM}(T, pc) = [\text{store } r \text{ to } x]^{pc}$. From the concrete semantics, it must be that $\text{axcin}_T \xrightarrow{\text{ax}} \text{axcout}_T$, where $\text{axcout}_T = \langle pc + 1, \bar{r}, \bar{x} \mid x \rightarrow (x \mid x) \mid T \rightarrow \{(\bar{r}, r, t)\}], \bar{l} \rangle$. Choose $\text{axcout}_T$ so that $\text{axcin}_T \xrightarrow{\text{ax}} \text{axcout}_T$, i.e., $\text{axcout}_T = \langle pc + 1, \bar{r}, \text{WRITE}(T, \bar{x}, x, (\bar{r}, r, t)), \bar{l} \rangle$. It is easy to see that (cf. Algorithm 5.5) $\bar{x} \mid x \rightarrow (x \mid x) \mid T \rightarrow \{(\bar{r}, r, t)\}] \in \mathcal{Y}_{\bar{r}}(\text{WRITE}(T, \bar{x}, x, (\bar{r}, r, t)))$, thus $\text{axcout}_T \subseteq \text{axout}_T(\text{axcout}_T)$. 

112   Chapter 5. Abstractly Interpreting PPL
6. Assume that $\text{STM}(T, pc) = [\text{load } r \text{ from } x]^p_c$. From the concrete semantics, $ax^\text{in}_T \xrightarrow{ax} ax^\text{out}_T$, where $ax^\text{out}_T = \langle pc + 1, t[r \mapsto x], \emptyset \rangle$ for some $v$ such that $\exists t' \in \text{Time} : (v, t') \in \bigcup_{T' \in \text{Thrd}} ((\exists x) T')$ if $\bigcup_{T' \in \text{Thrd}} ((\exists x) T') \neq \emptyset$ and $v \in \gamma_{int}(\lbrack -\infty, \infty \rbrack)$ otherwise. Choose $ax^\text{out}_T$ so that $ax^\text{out}_T \xrightarrow{ax} ax^\text{out}_T$, i.e., $ax^\text{out}_T = \langle pc + 1, \vec{x}[r \mapsto \text{READ}(\vec{x}, x, T, i)], \vec{\bar{x}}, \vec{\bar{y}} \rangle$. Since $\vec{x}$ is safe at abstract time $i$ and READ then returns a safe value (Lemma 5.27), it must be that $v \in \gamma_{val}(\text{READ}(\vec{x}, x, T, i))$ and thus $ax^\text{out}_T \in ax^\gamma_T(\text{ax}^\text{out}_T)$.

7. Assume that $\text{STM}(T, pc) = [\text{lock } lck]^p_c$. Then two cases must be considered.

(a) In the first case, $\text{OWN}(\lbrack lck \rbrack) = T$. From the concrete semantics, it must be that $ax^\text{in}_T \xrightarrow{ax} ax^\text{in}_T$, where $ax^\text{in}_T = \langle pc + 1, t, \vec{x}, \emptyset[lck \mapsto (\text{locked}, T, DL\lbrack lck \rbrack), \text{POWN}(\lbrack lck \rbrack), \text{REL}(\lbrack lck \rbrack)] \rangle$. Choose $ax^\text{out}_T$ so that $ax^\text{out}_T \xrightarrow{ax} ax^\text{out}_T$ by the corresponding branch, $(\text{OWN}(\lbrack lck \rbrack) = T \land (\text{STT}(\lbrack lck \rbrack) = \text{unlocked} \Rightarrow (i \not\in \vec{\bar{x}}, \text{REL}(\lbrack lck \rbrack) \land DL\lbrack lck \rbrack) \not\in \vec{\bar{y}}, \vec{\bar{z}}));$ i.e., $ax^\text{out}_T = \langle pc + 1, \vec{x}, \vec{\bar{x}}, \vec{\bar{z}}, \vec{\bar{y}}[lck \mapsto (\text{locked}, T, DL\lbrack lck \rbrack), \text{POWN}(\lbrack lck \rbrack), \text{REL}(\lbrack lck \rbrack)] \rangle$. Note that if $\text{STT}(\lbrack lck \rbrack) = \text{unlocked}$, it is implied that $t \not\in \text{REL}(\lbrack lck \rbrack) \land \text{DL}(\lbrack lck \rbrack) \not\in t$ (Lemma 4.6). Thus, it must be the case that $ax^\text{out}_T \in ax^\gamma_T(\text{ax}^\text{out}_T)$.

(b) In the second case, $\text{OWN}(\lbrack lck \rbrack) \neq T$. From the concrete semantics, it must be that $ax^\text{in}_T \xrightarrow{ax} ax^\text{out}_T$, where $ax^\text{out}_T = \langle pc, t, \vec{x}, \emptyset \rangle$. Choose $ax^\text{out}_T$ so that $ax^\text{out}_T \xrightarrow{ax} ax^\text{out}_T$ by the corresponding branch, $\text{OWN}(\lbrack lck \rbrack) \neq T \lor (\text{STT}(\lbrack lck \rbrack) = \text{unlocked} \land (i \not\in \vec{\bar{x}}, \text{REL}(\lbrack lck \rbrack) \lor DL\lbrack lck \rbrack) \not\in \vec{\bar{y}}, \vec{\bar{z}}));$ i.e., $ax^\text{out}_T = \langle pc, \vec{x}, \vec{\bar{x}}, \vec{\bar{y}} \rangle$. Thus, it must be the case that $ax^\text{out}_T \in ax^\gamma_T(\text{ax}^\text{out}_T)$.

8. Assume that $\text{STM}(T, pc) = [\text{unlock } lck]^p_c$. Then two cases must be considered.

(a) In the first case, $\text{OWN}(\lbrack lck \rbrack) = T$. From the concrete semantics, it must be that $ax^\text{in}_T \xrightarrow{ax} ax^\text{out}_T$, where $ax^\text{out}_T = \langle pc + 1, t, \vec{x}, \emptyset[lck \mapsto (\text{unlocked}, \bot, DL\lbrack lck \rbrack), T, t]) \rangle$. Choose $ax^\text{out}_T$ so that $ax^\text{out}_T \xrightarrow{ax} ax^\text{out}_T$ by the corresponding branch, $\text{OWN}(\lbrack lck \rbrack) = T \land \text{STT}(\lbrack lck \rbrack) = \text{locked}$; i.e., $ax^\text{out}_T = \langle pc + 1, \vec{x}, \vec{\bar{x}}, \vec{\bar{y}}[lck \mapsto (\text{unlocked}, \bot, DL\lbrack lck \rbrack), T, t]) \rangle$. Note that in the concrete case, $\text{STT}(\lbrack lck \rbrack) = \text{locked}$ whenever $\text{OWN}(\lbrack lck \rbrack) \neq \bot$ for a
valid configuration (Definition 4.4). Thus, it must be the case that 
\[ ax_{cT}^{out} \in ax_{cT}^{in} (ax_{cT}^{out}). \]

(b) In the second case, \( \text{own}(\mathbb{l} \ lck) \neq T \). From the concrete semantics, it must be that 
\[ \frac{ax_{cT}^{in} \rightarrow ax_{cT}^{out}}{ax_{cT}^{out} = (pc + 1, \bar{r}, \bar{x}, \bar{l})}. \]

Choose \( ax_{cT}^{out} \) so that \( \frac{ax_{cT}^{in} \rightarrow ax_{cT}^{out}}{ax_{cT}^{out}} \) by the corresponding branch, 
\( \text{own}(\mathbb{l} \ lck) \neq T \lor \text{stm}(\mathbb{l} \ lck) = \text{unlocked} \); i.e., 
\( ax_{cT}^{out} = (pc + 1, \bar{r}, \bar{x}, \bar{l}) \). Thus, it must be the case that 
\( ax_{cT}^{out} \in ax_{cT}^{in} (ax_{cT}^{out}) \). 

The abstract transition rule for program configurations in Table 5.13 is an approximation of the concrete rule in Table 4.5. The abstract rule now defines a window, i.e., interval, in time, \( t \), (since \( \text{time} = \text{intv} \)) that determines which threads are included in \( \text{thr}_{\text{exe}} \). The window reaches from the earliest point in time when some thread might execute its active statements, to the earliest point in time when some thread must execute its active statements. Note that \( \text{dlock} \) and \( \text{acctime} \) are defined in Algorithms 5.11 and 5.12 (\( \triangleright \) begins a comment; cf. Appendix A), respectively, and that \( \text{abstime} \) is assumed to be a safe approximation of \( \text{time} \), as specified in Assumption 5.51. The definition of \( \text{abstime} \) is outside the scope of this thesis but very simple instances of it (look-up tables) will be given when presenting instantiating examples in Chapter 7.

**Assumption 5.51 (abstime is safe and non-negative):**

It is assumed that \( \text{abstime} \) is a “non-negative” function that safely approximates \( \text{time} \) in the interval domain. More formally, it is assumed that 
\[ \forall \bar{c} \in ([T, pc_{T}, \bar{x}, \bar{r}, \bar{l}]_{T} \in \text{thr}_{\bar{x}}) \in \text{conf} : \]
\[ \forall T \in \text{thr}_{\bar{x}} : 0 \leq \min(\gamma_{\text{abstime}}(\bar{c}, T)) \]
and
\[ \forall \bar{c} \in ([T, pc_{T}, \bar{x}, \bar{r}, \bar{l}]_{T} \in \text{thr}_{\bar{x}}) \in \text{conf} : \]
\[ \forall \bar{c} \in ([T, pc_{T}, \bar{x}, \bar{r}, \bar{l}]_{T} \in \text{thr}_{\bar{x}}) \in \text{conf} : (\text{thr}_{\bar{x}} \subseteq \text{thr} \Rightarrow \forall T \in \text{thr}_{\bar{x}} : (pc_{T} = pc_{T} \lor t_{T}^{d} = t_{T}^{d} \in \gamma_{\text{abstime}}(\bar{c}, T))) \Rightarrow \text{time}(c, T) \in \gamma_{\text{abstime}}(\bar{c}, T)) \]
Table 5.13: $\tilde{c} \xrightarrow{prg} \tilde{c}'$, semantics of abstract program transitions.

<table>
<thead>
<tr>
<th>Transition Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\tilde{c} \xrightarrow{prg} \tilde{c}'$</td>
<td>$\langle [T, pc_T, \tilde{c}_T, \tilde{\tau}, \tilde{\tau}'] \mapsto \langle pc'_T, \tilde{c}'_T, \tilde{\tau}'_T, \tilde{\tau}'_T \rangle \rangle$</td>
</tr>
</tbody>
</table>

where

- $\tilde{\tau}'_T = \text{ABSTIME}(\tilde{c}, T)$
- $\tilde{\tau}_T = \alpha_t(\{\min(\{\min(\{\gamma, \tilde{\tau}'_T, \tilde{\tau}_T\}) \mid B\}), \min(\{\max(\{\gamma, \tilde{\tau}'_T, \tilde{\tau}_T\}) \mid B\})\}$
- $\tilde{\tau}' = \alpha_t(\{\min(\{\min(\{\gamma, \tilde{\tau}'_T, \tilde{\tau}_T\}) \mid B\}), \min(\{\max(\{\gamma, \tilde{\tau}'_T, \tilde{\tau}_T\}) \mid B\})\}$

- $\text{Thrd}_{\text{exe}} \neq \emptyset \land \forall T \in \text{Thrd}_{\text{exe}} : (T, pc_T, \tilde{c}_T, \tilde{\tau}, \tilde{\tau}') = \tilde{c}' \xrightarrow{prg} (pc'_T, \tilde{c}'_T, \tilde{\tau}'_T, \tilde{\tau}'_T)$

5.8 Abstract Semantics
Algorithm 5.11 Determine deadline for lock owner assignment

1: function dllock(\( \bar{c} @ \langle [T', pc_T], \bar{T}, \bar{I}_{dl} | T \in \text{Thrd}_t, \bar{x}, \bar{I} \rangle, lck \) )
2: \( \bar{I}_{dl} \leftarrow \bar{I}_t \)
3: for all \( T \in \text{Thrd}_t \) do
4: if \( STM(T, pc_T) = [lock lck]^{pc_T} \) then
5: \( c' \leftarrow \bar{c} \)
6: \( \bar{I}_T' \leftarrow \bar{I}_T \)
7: \( \bar{I}_{dl}' \leftarrow [-\infty, -\infty] \)
8: repeat
9: \( \bar{I}' \leftarrow \text{ABSTIME}(c', T) \)
10: \( \bar{I}_{dl}' \leftarrow \bar{I}_{dl}' \cap \bar{t} (\bar{I}_T' + \bar{I}') \)
11: \( \bar{I}_{dl}' \leftarrow \bar{I}_{dl}' \cap (R\text{EL}(lck) \cup [-\infty, -\infty]) \)
12: \( c' \leftarrow \langle [T', pc_T], \bar{T}, (T = T' ? \bar{I}_T' : \bar{I}_T) | T \in \text{Thrd}_t, \bar{x}, \bar{I} \rangle \)
13: until \( 0 \in \gamma_I(\bar{I}') \lor \bar{I}_{dl}' = \bot \)
14: if \( \bar{I}_{dl}' \neq \bot \land 0 \in \gamma_I(\bar{I}') \) then
15: \( \bar{I}_{dl}' \leftarrow (\bar{I}_{dl}' \cap \bar{t} (\bar{I}_T' + \bar{I}') \cap (R\text{EL}(lck) \cup [\infty, \infty]) \)
16: \( c' \leftarrow \langle [T', pc_T], \bar{T}, (T = T' ? \bar{I}_T' : \bar{I}_T) | T \in \text{Thrd}_t, \bar{x}, \bar{I} \rangle \)
17: \( \bar{I}_{dl}' \leftarrow \bar{I}_{dl}' \cap \bar{t} \text{ABSTIME}(c', T) \)
18: \( \bar{I}_{dl}' \leftarrow \bar{I}_{dl}' \cap \bar{I}_{dl}' \)
19: \( \bar{I}_{dl}' \leftarrow \bar{I}_{dl}' \cap \bar{I}_{dl}' \)
20: else
21: \( \bar{I}_{dl}' \leftarrow \bar{I}_{dl}' \cap \bar{I}_{dl}' \)
22: end if
23: end if
24: end for
25: return \( \bar{I}_{dl}' \)
26: end function

Algorithm 5.12 Determine accumulated execution time

1: function accTime(\( \bar{c} @ \langle [T', pc_T], \bar{T}, \bar{I}_{dl} | T \in \text{Thrd}_t, \bar{x}, \bar{I} \rangle, \text{Thrd}_exe, T \) )
2: \( \bar{I}_{dl}'' \leftarrow \bar{I}_{dl}'' \)
3: if \( T \in \text{Thrd}_exe \) then
4: \( \bar{I}_T' \leftarrow \text{ABSTIME}(\bar{c}, T) \)
5: if \( \forall lck \in Lck : STM(T, pc_T) \neq [lock lck]^{pc_T} \) then
6: \( \bar{I}_{dl}'' \leftarrow \bar{I}_{dl}'' + \bar{I}_T' \)
7: else
8: end if
9: end if
10: return \( \bar{I}_{dl}'' \)
11: end function
Algorithm 5.12 Cont. Determine accumulated execution time

\(\text{for all } lck \in \text{Lck do} \)

\(\text{if } \text{STM}(T, pc_T) = [\text{lock } lck]^{pc_T} \land \text{OWN}(\bar{t} lck) = T \text{ then} \)

\(\text{if } S \bar{t} T(\bar{t} lck) = \text{locked} \text{ then} \)

\(\tilde{t} a T' \leftarrow \tilde{t} a T' + \tilde{t} a T \)

\(\text{else if } \text{DL}(\bar{t} lck) \preceq \gamma_t (\tilde{t} a T' + \tilde{t} a T) \text{ then} \)

\(\tilde{t} a T' \leftarrow \tilde{t} a T' + \tilde{t} a T \)

\(\text{else if } (\tilde{t} a T' + \tilde{t} a T) \preceq \gamma_t \text{ REL}(\bar{t} lck) \text{ then} \)

\(\tilde{c'} \leftarrow \tilde{c} \)

\(\text{while } (\tilde{t} a T' + \tilde{t} a T, \text{ABSTIME}(\tilde{c'}, T)) \preceq \gamma_t \text{ REL}(\bar{t} lck) \text{ do} \)

\(\tilde{t} a T' \leftarrow (\tilde{t} a T' + \tilde{t} a T, \text{ABSTIME}(\tilde{c'}, T)) \)

\(\tilde{c'} \leftarrow ([T', pc_T, \bar{t} T', (T = T' ? \tilde{t} a T' : \tilde{t} a T); \forall T' \in \text{Thrd}, \tilde{x}, \tilde{z})] \)

\(\text{end while} \)

\(\text{else if } \text{POWN}(\bar{t} lck) = T \lor \text{REL}(\bar{t} lck) \preceq \gamma_t (\tilde{t} a T' + \tilde{t} a T) \text{ then} \)

\(\tilde{t} a T' \leftarrow (\tilde{t} a T' + \tilde{t} a T) \text{ REL}(\bar{t} lck) \)

\(\text{else } \gamma_t S \bar{t} T(\bar{t} lck) = \text{unlocked} \land \text{REL}(\bar{t} lck) \cap (\tilde{t} a T' + \tilde{t} a T) \neq \bar{t} lck \land \text{POWN}(\bar{t} lck) \neq T \land \text{DL}(\bar{t} lck) \preceq \gamma_t (\tilde{t} a T' + \tilde{t} a T) \}

\(\tilde{t} a T' \leftarrow \tilde{t} a T' + \tilde{t} a T \)

\(\tilde{c'} \leftarrow \tilde{c} \)

\(\text{repeat} \)

\(\text{if } \text{DL}(\bar{t} lck) \preceq \gamma_t (\tilde{t} a T' + \tilde{t} a T, \text{ABSTIME}(\tilde{c'}, T)) \text{ then} \)

\(\tilde{t} a T' \leftarrow \tilde{t} a T' \)

\(\text{else if } 0 \in \gamma_t (\text{ABSTIME}(\tilde{c'}, T)) \text{ then} \)

\(\tilde{t} = (\tilde{t} a T' \cup \gamma_t [0, \infty]) \cap \gamma_t \text{ REL}(\bar{t} lck) \)

\(\tilde{c'} \leftarrow ([T', pc_T, \bar{t} T', (T = T' ? \tilde{t} a T' : \tilde{t} a T); \forall T' \in \text{Thrd}, \tilde{x}, \tilde{z})] \)

\(\tilde{t} a T' \leftarrow \tilde{t} a T' + \tilde{t} a T, \text{ABSTIME}(\tilde{c'}, T) \)

\(\tilde{t} a T' \leftarrow \tilde{t} a T' \)

\(\text{else} \)

\(\tilde{t} a T' \leftarrow \tilde{t} a T' + \tilde{t} a T, \text{ABSTIME}(\tilde{c'}, T) \)

\(\tilde{c'} \leftarrow ([T', pc_T, \bar{t} T', (T = T' ? \tilde{t} a T' : \tilde{t} a T); \forall T' \in \text{Thrd}, \tilde{x}, \tilde{z})] \)

\(\text{end if} \)

\(\text{until } \tilde{t} a T' = \tilde{t} a T' \lor \text{REL}(\bar{t} lck) \preceq \gamma_t \tilde{t} a T' \)

\(\tilde{t} a T' \leftarrow (\tilde{t} a T' \cup \tilde{t} a T') \cap \gamma_t \text{ DL}(\bar{t} lck) \cap \gamma_t (\text{REL}(\bar{t} lck) \cup [0, \infty]) \)

\(\text{end if} \)

\(\text{end if} \)

\(\text{return } \tilde{t} a T' \)

\(\text{end function} \)
Since \textbf{Time} is approximated using \textbf{Ti\text{\textit{m}}e} = \textbf{Intv}, it is not possible to determine the exact ordering of events in the abstract case. This renders \( \xrightarrow{\text{prg}} \) an unsafe approximation of \( \xrightarrow{\text{prg}} \) (cf. Tables 4.5 and 5.13) in the general case; the specific issues are listed below.

1. The sets of threads to execute, i.e., \textbf{Thrd}_{\text{exe}}, might differ between \( c \in \textbf{Conf} \) and \( \tilde{c} \in \textbf{Conf} \), even if \( c \in \gamma_{\text{conf}}(\tilde{c}) \). Because of this, different program points might be “visited” in the concrete and abstract cases, and thus, fixed-point calculations on \( \xrightarrow{\text{prg}} \) in the traditional sense [27, 41] cannot be used to find a safe over-approximation of the concrete collecting semantics.

2. The execution of \texttt{load}-statements cannot be safely approximated using \( \xrightarrow{\text{prg}} \) if \(|\textbf{Thrd}_{\text{exe}}| > 1\) and the value of a global variable is to be loaded. The reason for this is that executing \texttt{load}-statements introduces data-dependencies between the threads, and the \texttt{READ}-function could return a value for which all possible writes have not been taken into account; i.e., all \texttt{store}-statements that could affect the variable have not yet been executed (and thus, \( \tilde{x} \) does not contain a safe write history). To see this, assume that for some abstract configuration, \( \textbf{Thrd}_{\text{exe}} = \{T_1, T_2\} \), \( \text{STM}(T_1, pc_{T_1}) = [\texttt{load } r \text{ from } x]^{pc_{T_1}} \), \( \text{STM}(T_2, pc_{T_2}) = [\texttt{skip}]^{pc_{T_2}} \) and \( \text{STM}(T_2, pc_{T_2} + 1) = [\texttt{store } r' \text{ to } x]^{pc_{T_2} + 1} \). When a transition occurs, the \texttt{load}- and \texttt{skip}-statements are considered. However, if the execution time of the \texttt{store}-statement (the interval in time when the thread’s \textit{pc} is updated) overlaps with the execution time of the \texttt{load}-statement, then the resulting value of \( r \) in \( T_1 \) should be affected by the value of \( r' \) in \( T_2 \), but this will not be the case.

3. A similar reasoning to that for \texttt{load}-statements holds for \texttt{lock}-statements: an unlocked lock, \( lck \in \textbf{Lck} \), cannot simply be assigned to one of the threads in \textbf{Thrd}_{\text{exe}} that issues \texttt{lock lck}. This is because in the concrete case, the lock might be assigned to another thread in \textbf{Thrd}_{\text{exe}} (that might not yet be executing \texttt{lock lck} in the abstract case). Thus, the only safe option is to make assignments to, at least, each thread specified in the considered abstract configuration that at some point might acquire \( lck \). This is because these threads (even if currently not in \textbf{Thrd}_{\text{exe}}) could compete for \( lck \) with subsequent statements. If a thread that has been assigned \( lck \) actually does not compete for \( lck \), this can be detected if the thread reaches a \texttt{halt}-statement or using the deadline parameter...
in the state for \( lck \).

4. A transition sequence containing deadlocked configurations will not be safely approximated. In the concrete case, the threads included in the deadlock are spinning on the locks they are waiting to acquire. This means that time moves forward for these threads (given that \( \text{TIME} \) is non-zero). However, in the abstract case, the threads will be frozen and their accumulated times do not increase on transitions.

To handle these issues, the analysis will be proven to (whenever it terminates) safely approximate the timing bounds of any concrete configuration, \( c \in \langle [T, pc_T, \varnothing_T, T_{I_T}]_{T \in \text{Thrd}} : x, [] \rangle \in \text{Conf} \), in the finite collecting semantics, \( \mathcal{C}(C) \), of a program in the initial states described by the configurations in \( C \), such that \( \forall T \in \text{Thrd} : \text{STM}(T, pc_T) = [\text{halt}]^{pc_T} \). The analysis will also be proven to (whenever it terminates) safely approximate the timing bounds of any infinite collecting semantics; which of course renders an infinite WCET. More on this in Chapter 6.

\( \xrightarrow{\text{prg}} \) will be proven to be a safe approximation of \( \xrightarrow{\text{prg}} \) in any finite collecting semantics, with respect to each thread individually, for any left-hand side (i.e., input) configuration, \( \tilde{c} @ (([T, pc_T, \varnothing_T, T_{I_T}]_{T \in \text{Thrd}_t}, \tilde{x}, []) \in \text{Conf} \), such that \( |\text{Thrd}_{\text{exe}}| \neq 1 \vee \{ T \in \text{Thrd}_{\text{exe}} | \exists r \in \text{Reg}_T : \exists x \in \text{Var}_g : \text{STM}(T, pc_T) = [\text{load } r \text{ from } x]^{pc_T} = \emptyset \}, \) where \( \text{Var}_g \) is the set of all global variables (i.e., variables that might transfer data between threads and hence could be subject to data races); i.e., either no thread issues a load-statement on a global variable, or there is such a thread and it is the sole thread that is executed, which means that \( \tilde{x} \) must contain a safe write history since no more writes on \( x \) can occur before the given load-statement has been executed. It should be noted that outdated writes (i.e., writes that will never be considered by a load-statement in any thread) are trimmed away from the variable store resulting from a transition, \( \tilde{x}' \), given that \( \text{Thrd}_c = \text{Thrd} \) (the reason for this condition will become apparent in Chapter 6 where a recursive worklist algorithm, encapsulating \( \xrightarrow{\text{prg}} \), is presented).

One thing to notice from how \( \xrightarrow{\text{prg}} \) is defined is that an abstract configuration cannot have the same restrictions for being valid as a concrete configuration has (cf. Definition 4.4). When a thread (in \( \text{Thrd}_{\text{exe}} \)) wants to acquire some free lock, \( \xrightarrow{\text{prg}} \) can assign the lock to any thread that at some point in the program wants to acquire the lock, as discussed in 3 above. However (quite obviously), the assigned thread might not acquire the lock with its current statement (it is also possible that the
thread never acquires the lock at all with its future statements). Therefore, an abstract configuration, \( \vec{c}@\langle [T, pc_T, \bar{\varepsilon}_T, \vec{t}_T]_{T \in \text{Thrd}}, \bar{x}, \bar{l} \rangle \in \text{Conf} \), must be considered temporarily valid even if \( \exists \text{lck} \in \text{Lck} : (\text{OWN} (\vec{l} \text{lck}) \neq \bot) \). As also discussed in 3 above, however, such an abstract configuration can be considered invalid if \( \exists \text{lck} \in \text{Lck} : (\text{OWN} (\vec{l} \text{lck}) \neq \bot \land \text{STM} (\text{OWN} (\vec{l} \text{lck}), pc_{\text{OWN}(\vec{l} \text{lck})}) = [\text{halt}]^{pc_{\text{OWN}(\vec{l} \text{lck})})} \) (i.e., there is a lock that is assigned to some thread that has not acquired the lock before the deadline has expired or before the thread has terminated its execution), given that \( \text{DLlck} (\vec{l} \text{lck}) \) is a safe approximation of when \( \text{lck} \) must have been taken by some thread in the corresponding concrete cases (cf. Lemma 5.54), if any.

Another difference is that in the abstract case, a lock-issuing thread will be frozen (i.e., not at all considered on transitions) if the given lock is, or has already been, assigned to some other thread. The issuing thread remains frozen until the lock is assigned to it. (Cf. the definitions of \( \vec{t}_{\text{all}}, \text{Thrd}_{\text{all}}, \vec{i}^\prime, \text{Thrd}_{\text{hold}}, \vec{i} \) and \( \text{Thrd}_{\text{exe}} \) in Table 5.13.) Note that if the lock’s release time is in the future when a thread, \( T \in \text{Thrd} \), is assigned the ownership of it (which can be the case for threads that have been frozen), then \( \vec{t}_T^\prime \) will be increased to safely approximate the concrete spin-waiting (cf. Lemma 5.58).

In the concrete case, a free (i.e., not assigned, and thus unlocked) lock is acquired as soon as some thread tries to do so (cf. Tables 4.2 and 4.5 and Lemma 4.5). The purpose of \( \text{DLlck} \), defined in Algorithm 5.11 on page 116, is to derive a safe approximation of this point in time (Lemma 5.54). Note that Lemma 5.52 states that accumulating time for each thread individually is safe and that Lemma 5.53 states that the timing of a thread can be analyzed in isolation from other threads.

**Lemma 5.52 (Time accumulation):**

*Given the two configurations \( \vec{c}@\langle [T, pc_T, \bar{\varepsilon}_T, \vec{t}_T]_{T \in \text{Thrd}}, \bar{x}, \bar{l} \rangle \in \text{Conf} \) and \( \vec{c}@\langle [T, pc_T, \bar{\varepsilon}_T, \vec{t}_T]_{T \in \text{Thrd}}, \bar{x}, \bar{l} \rangle \in \text{Conf} \), such that \( \text{Thrd}_T \subseteq \text{Thrd} \), let \( \text{Thrd}' = \{ T \in \text{Thrd}_T \mid \vec{t}_T^\prime \in \gamma_5(\vec{t}_T^\prime) \land pc_T = pc_T^\prime \} \). Then \( \forall T \in \text{Thrd}' : (t_T^\prime + \text{TIME}(c, T)) \in \gamma_5(\vec{t}_T^\prime \downarrow, \text{ABSTIME}(\vec{c}, T)) \).*

**PROOF.** Assume that the configurations \( \vec{c}@\langle [T, pc_T, \bar{\varepsilon}_T, \vec{t}_T]_{T \in \text{Thrd}}, \bar{x}, \bar{l} \rangle \in \text{Conf} \) and \( \vec{c}@\langle [T, pc_T, \bar{\varepsilon}_T, \vec{t}_T]_{T \in \text{Thrd}}, \bar{x}, \bar{l} \rangle \in \text{Conf} \) are such that \( \text{Thrd}_T \subseteq \text{Thrd} \), and let \( \text{Thrd}' = \{ T \in \text{Thrd}_T \mid \vec{t}_T^\prime \in \gamma_5(\vec{t}_T^\prime) \land pc_T = pc_T^\prime \} \). Then, according to Assumption 5.51, \( \forall T \in \text{Thrd}' : \text{TIME}(c, T) \in \gamma_5(\text{ABSTIME}(\vec{c}, T)) \). Since \( \forall T \in \text{Thrd}' : t_T^\prime \in \gamma_5(\vec{t}_T^\prime) \), it is easy to see that \( \forall T \in \text{Thrd}' : (t_T^\prime + \text{TIME}(c, T)) \in \gamma_5(\vec{t}_T^\prime \downarrow, \text{ABSTIME}(\vec{c}, T)) \).
Lemma 5.53 (Thread isolation):
If the two configurations $c^0 @ \langle [T, pc^0_T, \bar{z}^0_T, \bar{t}^0_T] \rangle_{T \in \text{Thrd}_c} \in \text{Conf}$ and $\bar{c}^0 @ \langle [T, pc^0_T, \bar{z}^0_T, \bar{t}^0_T] \rangle_{T \in \text{Thrd}_c} \in \text{Conf}$, and some thread, $T \in \text{Thrd}_c$, are such that $\text{Thrd}_c \subseteq \text{Thrd}$, $t^0_T \in \gamma_i(\tilde{t}^0_T)$ and $pc^0_T = pc^0_T$, and if for the configuration $c^{n+1} @ \langle [T, pc^{n+1}_T, \bar{z}^{n+1}_T, \bar{t}^{n+1}_T] \rangle_{T \in \text{Thrd}_c} \in \text{Conf}$, then

$$
c^0 \xrightarrow{\text{prg}} \ldots \xrightarrow{\text{prg}} c^1 \xrightarrow{\text{prg}} \ldots \xrightarrow{\text{prg}} c^2 \xrightarrow{\text{prg}} \ldots \xrightarrow{\text{prg}} c^n \xrightarrow{\text{prg}} c^{n+1}
$$

for some $n \geq 0$, then

$$
t^{n+1}_T \in \gamma_i(\tilde{t}^0_T) \uparrow_i, \text{ABSTIME}(\tilde{c}^0, T) \uparrow_i,
\text{ABSTIME}(\tilde{c}^1, T) \uparrow_i,
\text{ABSTIME}(\tilde{c}^2, T) \uparrow_i,
\ldots \uparrow_i,
\text{ABSTIME}(\tilde{c}^n @ \langle [T, pc^n_T, \bar{z}^n_T, \bar{t}^n_T] \rangle_{T \in \text{Thrd}_c} \in \text{Conf}),
$$

given that $\forall i \in \{1, 2, \ldots, n\} : \tilde{t}^i_T = \tilde{t}^{i-1}_T \uparrow_i, \text{ABSTIME}(\tilde{c}^{i-1}, T), \forall i \in \{0, 1, 2, \ldots, n\} : pc^i_T = pc^i_T, \forall i \in \{0, 1, 2, \ldots, n\} : T \in \text{Thrd}_c$, and $\forall c \in \{c^0, 1, c^1, \ldots, c^n\} \setminus \{c^0, c^1, \ldots, c^n\} : T \in \text{Thrd}_c$, where $\text{Thrd}_c$ and $\text{Thrd}_c$ are as defined in Table 4.5 for all $c^i$ and all other $c$ on the trace from $c^0$ to $c^{n+1}$. There might exist intermediate configurations on the trace from $c^0$ to $c^1$ etc. but for any such configuration, $T \not\in \text{Thrd}_c$.

PROOF. Assume that the configurations $c^0 @ \langle [T, pc^0_T, \bar{z}^0_T, \bar{t}^0_T] \rangle_{T \in \text{Thrd}_c} \in \text{Conf}$ and $\bar{c}^0 @ \langle [T, pc^0_T, \bar{z}^0_T, \bar{t}^0_T] \rangle_{T \in \text{Thrd}_c} \in \text{Conf}$, and some thread, $T \in \text{Thrd}_c$, are such that $\text{Thrd}_c \subseteq \text{Thrd}$, $t^0_T \in \gamma_i(\tilde{t}^0_T)$ and $pc^0_T = pc^0_T$. Also assume that $c^0 \xrightarrow{\text{prg}} \ldots \xrightarrow{\text{prg}} c^1 \xrightarrow{\text{prg}} \ldots \xrightarrow{\text{prg}} c^2 \xrightarrow{\text{prg}} \ldots \xrightarrow{\text{prg}} c^n \xrightarrow{\text{prg}} c^{n+1}$ for some configuration $c^{n+1} @ \langle [T, pc^{n+1}_T, \bar{z}^{n+1}_T, \bar{t}^{n+1}_T] \rangle_{T \in \text{Thrd}_c} \in \text{Conf}$, and $n \geq 0$, for which $\forall i \in \{0, 1, 2, \ldots, n\} : T \in \text{Thrd}_c$, and $\forall c \in \{c^0, c^1, \ldots, c^n\} \setminus \{c^0, 1, c^1, \ldots, c^n\} : T \not\in \text{Thrd}_c$, where $\text{Thrd}_c$ and $\text{Thrd}_c$ are as defined in Table 4.5 for all $c^i$ and all other $c$ on the trace from $c^0$ to $c^{n+1}$.
From Table 4.5, it is easy to see that:

\[
\begin{align*}
    t_T^0 &= t_T^0 \\
    t_T^1 &= t_T^0 + \text{time}(c^0, T) \\
    t_T^2 &= t_T^1 + \text{time}(c^1, T) = t_T^0 + \text{time}(c^0, T) + \text{time}(c^1, T) \\
    &\vdots \\
    t_T^{n+1} &= t_T^n + \text{time}(c^n, T) = t_T^0 + \Sigma_{i=0}^n \text{time}(c^i, T)
\end{align*}
\]

Let \( \{\tilde{c}^0, \tilde{c}^1, \tilde{c}^2, \ldots, \tilde{c}^n \} \) be a set of some abstract configurations such that \( \tilde{c}^0 \) has the properties assumed above, \( \forall i \in \{1, 2, \ldots, n\} : \text{pc}_{\tilde{c}^i} = \text{pc}_{\tilde{c}^i} \) and \( \forall i \in \{1, 2, \ldots, n\} : \tilde{a}^{c^i} = \tilde{a}^{c^{i-1}} + \gamma_i, \text{ABSTIME}(\tilde{c}^{i-1}, T) \). Then, according to Lemma 5.52:

\[
\begin{align*}
    (t_T^0 + \text{time}(c^0, T)) &\in \gamma_t(\tilde{a}^0 \mod\gamma_t, \text{ABSTIME}(\tilde{c}^0, T)) \\
    \vdots \\
    (t_T^n + \text{time}(c^n, T)) &\in \gamma_t(\tilde{a}^n \mod\gamma_t, \text{ABSTIME}(\tilde{c}^n, T))
\end{align*}
\]

Since \( t_T^{n+1} = t_T^n + \text{time}(c^n, T) \), this concludes the proof.

Lemma 5.54 (Soundness of DLLOCK):

If the valid concrete configurations (cf. Definition 4.4), abstract configurations and lock

\[
\begin{align*}
    c^0 @ \langle [T, pc_T^0, x_T^0, \tilde{a}^0] \rangle_{T \in \text{Thr}_\text{d}} &\in \text{Conf}, \\
    c^m @ \langle [T, pc_T^m, x_T^m, \tilde{a}^m] \rangle_{T \in \text{Thr}_\text{d}} &\in \text{Conf}, \\
    c^n @ \langle [T, pc_T^n, x_T^n, \tilde{a}^n] \rangle_{T \in \text{Thr}_\text{d}} &\in \text{Conf}, \\
    \tilde{c}^0 @ \langle [T, pc_T^0, x_T^0, \tilde{a}^0] \rangle_{T \in \text{Thr}_\text{d}} &\in \text{Conf}, \\
    \tilde{c}^0 @ \langle [T, pc_T^0, x_T^0, \tilde{a}^0] \rangle_{T \in \text{Thr}_\text{d}} &\in \text{Conf}, \\
    \tilde{c}^j @ \langle [T, pc_T^j, x_T^j, \tilde{a}^j] \rangle_{T \in \text{Thr}_\text{d}} &\in \text{Conf}, \text{ and } lck \in \text{Lck},
\end{align*}
\]
are such that
\[
0 \leq m \leq n,
\]
\[
c_0 \xrightarrow{prg} \ldots \xrightarrow{prg} c_m \xrightarrow{prg} \ldots \xrightarrow{prg} c^n,
\]
\[
0 \leq j,
\]
\[
\text{Thrd}_{\tilde{c}^j} \subseteq \text{Thrd} \subseteq \text{Thrd}_{\tilde{c}},
\]
\[
\forall i \in \{m, \ldots, n\} : \text{OWN}(\llbracket lck \rrbracket) = \bot_{\text{thrd}},
\]
\[
\text{REL}(\llbracket^m lck) \in \gamma_i(\text{REL}(\llbracket^i lck)),
\]
\[
\exists T \in \text{Thrd}_{\tilde{c}^j} : (\text{STM}(T, pc^0) = [\text{lock } lck])^T \land \gamma^0_T \in \gamma_i(\llbracket T \rrbracket) \land \gamma^j_T = \gamma^0_T \land \gamma^j_{\text{conf}} = \gamma^0_{\text{conf}}
\]
\[
\forall i \in \{0, \ldots, n\} : \text{OWN}(\llbracket lck \rrbracket) \neq T,
\]
\[
\forall i \in \{m, \ldots, n - 1\} : \forall T \in \text{Thrd}^j_{\text{exe}} : \text{STM}(T, pc^i_T) = [\text{lock } lck]^{pc^i_T},
\]
where \text{Thrd}^j_{\text{exe}} is as defined in Table 4.5 for \( c^j \), then DLLOCK satisfies:
\[
\min\{\{r^i_T + \text{TIME}(c^n, T) \mid T \in \text{Thrd}\}\} \in \gamma_i(\text{DLLOCK}(\tilde{c}^j, lck))
\]

**Explanation of the Lemma.** Thread T might be spin-waiting to acquire the lock \( lck \) which is released in a transition to \( c^m \); in the transitions between configurations in-between \( c^m \) and \( c^n \), no thread executes \( \text{lock } lck \); T executes \( \text{lock } lck \) and might be assigned the ownership of \( lck \) in a transition from \( c^n \); DLLOCK safely approximates the point in time when T (i.e., some arbitrary thread) might be assigned the ownership of \( lck \).

**Proof.** Assume that the valid concrete configurations (cf. Definition 4.4), abstract configurations and lock
\[
c^0 @ \llbracket [T, pc^0_T, t^0_T, r^0_T] \mid T \in \text{Thrd}, x^0, \llbracket 0 \rrbracket \rrbracket \in \text{Conf},
\]
\[
c^m @ \llbracket [T, pc^m_T, t^m_T, r^m_T] \mid T \in \text{Thrd}, x^m, \llbracket m \rrbracket \rrbracket \in \text{Conf},
\]
\[
c^n @ \llbracket [T, pc^n_T, t^n_T, r^n_T] \mid T \in \text{Thrd}, x^n, \llbracket n \rrbracket \rrbracket \in \text{Conf},
\]
\[
c^0 @ \llbracket [T, pc^0_T, t^0_T, r^0_T] \mid T \in \text{Thrd}_{\tilde{c}}, x^0, \llbracket 0 \rrbracket \rrbracket \in \text{Conf},
\]
\[
c^j @ \llbracket [T, pc^j_T, t^j_T, r^j_T] \mid T \in \text{Thrd}_{\tilde{c}^j}, x^j, \llbracket j \rrbracket \rrbracket \in \text{Conf},
\]
\[
lck \in \text{Lck},
\]
are as assumed in the lemma above. First note that:

- Since \( \forall i \in \{m, \ldots, n - 1\} : \text{OWN}(\llbracket^i lck \rrbracket) = \bot_{\text{thrd}} \), it must be that \( \text{REL}(\llbracket^0 lck) = \text{REL}(\llbracket^m lck) \) (Tables 4.2 and 4.5).
Chapter 5. Abstractly Interpreting PPL

- Since \( \forall i \in \{m, \ldots, n\} : \text{OWN}([i] \text{lck}) = \bot_{\text{thr}} \) and \( \text{REL}([m] \text{lck}) = \text{REL}([n] \text{lck}) \), it must be that \( \forall T \in \text{Thrd} : t^m_T \leq \text{REL}([n] \text{lck}) \) (Tables 4.2 and 4.5 and Lemma 4.2).

- Since time only moves forward (Lemma 4.2), it must be that for \( c^n \), \( \forall T \in \text{Thrd} : \text{REL}([n] \text{lck}) \leq t^m_T + \text{TIME}(c^n, T) \).

- Since \( \forall i \in \{m, \ldots, n-1\} : \forall T \in \text{Thrd}_{ex}^{i} : \text{STM}(T, pc^n_T) = [\text{lck}]pc^n_T \), it must be that \( \forall T \in \text{Thrd} : (\text{STM}(T, pc^n_T) = [\text{lck}]pc^n_T \Rightarrow t^m_T = t^m_T) \) (Tables 4.2 and 4.5).

- Since \( \exists T \in \text{Thrd}_{ej}^{i} : (\text{STM}(T, pc^0_T) = [\text{lck}]pc^0_T \land t^0_T \in \gamma_i(t^0_T) \land t^0_T = t^0_T \land pc^0_T = \gamma_T(T) \land \forall i \in \{0, \ldots, n\} : \text{OWN}([i] \text{lck} \neq T) \) \( \wedge 0 \leq m \leq n \) (there is a thread in \( \tilde{e}^{i} \) that has been frozen on \text{lck} since \( c^0 \) and has been spin-waiting since \( c^0 \) and might be assigned \text{lck} in a transition from \( c^n \)), \( \text{Thrd}_{ej}^{i} \subseteq \text{Thrd} \), \( \forall T \in \text{Thrd} : t^m_T \leq \text{REL}([n] \text{lck}), \forall T \in \text{Thrd} : \text{REL}([n] \text{lck}) \leq t^m_T + \text{TIME}(c^n, T) \) and \( \forall T \in \text{Thrd} : (\text{STM}(T, pc^n_T) = [\text{lck}]pc^n_T \Rightarrow t^m_T = t^m_T) \) (the accumulated execution times of the threads have the derived properties) it must be that \( \exists T \in \text{Thrd}_{ej}^{i} : t^m_T \leq \text{REL}([n] \text{lck}) \leq t^m_T + \text{TIME}(c^n, T) \land \text{STM}(T, pc^0_T) = [\text{lck}]pc^0_T \land t^0_T \in \gamma_i(t^0_T) \land t^0_T = t^0_T \land pc^0_T = \gamma_T(T) \land \forall i \in \{0, \ldots, n\} : \text{OWN}([i] \text{lck} \neq T) \) (there is at least one thread that might have been unsuccessfully trying to acquire \text{lck} from \( c^0 \) and will (perhaps again) try to acquire \text{lck}, which was released at the latest in a transition to \( c^m \), in a transition from \( c^n \)).

From here on, it will be assumed that \( T' \in \text{Thrd}_{ej}^{i} \) is one of the threads such that \( \text{STM}(T', pc^0_{T'}) = [\text{lck}]pc^0_{T'} \land t^m_{T'} \leq \text{REL}([n] \text{lck}) \leq t^m_{T'} + \text{TIME}(c^n, T') \land t^0_{T'} \in \gamma_i(t^0_{T'}) \land t^0_{T'} = t^0_{T'} \land pc^0_{T'} = \gamma_T(T') \land \forall i \in \{0, \ldots, n\} : \text{OWN}([i] \text{lck} \neq T') \).

- Let \( \{m_1, \ldots, m_2\} \) be the set of indices, such that \( 0 \leq m_1 \leq m_2 \leq m \), \( \forall i \in \{m_1, \ldots, m_2\} : T' \in \text{Thrd}_{ex}^{i} \) and \( \forall i \in \{0, \ldots, m\} \setminus \{m_1, \ldots, m_2\} : T' \notin \text{Thrd}_{ex}^{i} \), where \( \text{Thrd}_{ex}^{i} \) is as defined in Table 4.5 for \( c^i \) (note that it is possible that \( \{m_1, \ldots, m_2\} = \emptyset \); the only known relation is \( \{m_1, \ldots, m_2\} \subseteq \{0, \ldots, m\} \)). In other words, \( c^{m_1}, \ldots, c^{m_2} \) represent the configurations from which a transition increases \( T' \)’s accumulated execution time. Since \( \text{Thrd}_{ej}^{i} \subseteq \text{Thrd} \), \( t^0_{T'} \in \gamma_i(t^0_{T'}) \).
\[
\bar{t}_T^{d'} = \bar{t}_T^{d}, \quad c_0 \xrightarrow{prg} \ldots \xrightarrow{prg} c_m, \quad t_T^{d''} \leq \text{REL}(lck) \leq t_T^{d'''} + \text{TIME}(c^n, T'),
\]

\text{REL}(lck) \in \gamma_i(\text{REL}(lck)) \text{ and } 0 \leq m, \text{ it is easy to see that every configuration, } c', \text{ created by the } \text{repeat}-\text{loop within } \text{DLLOCK} \text{ fulfills the assumptions of Lemma 5.53. Furthermore, it is easy to see that (according to Lemma 5.52; cf. Lemma 5.53) the loop will iterate at least } \ceil{|\{m_1, \ldots, m_2\}| + 1 \text{ times (given that } 0 \not\in \text{ABSTIME}(c', T') \text{) and for each of the iterations, the derived } c' \text{ safely approximates the corresponding concrete configuration (since in the concrete case, } t_T^{d''} \leq \text{REL}(lck) \text{ for } i \in \{m_1, \ldots, m_2\}, \text{ the abstract execution time can safely be trimmed as done on line 11).}

For the sake of readability, let

\[
\begin{cases}
\bar{t}_T^{d'''} = \bar{t}_T^{d'} \text{ at the } \text{repeat}-\text{loop exit, and} \\
c'' = c' \text{ at the } \text{repeat}-\text{loop exit.}
\end{cases}
\]

Assuming that \(\bar{t}_{dl}\) is safe at the start of each iteration of the for all \(T \in \text{Thrd}_i\)-loop within \(\text{DLLOCK}\), where \(T\) is such that \(\text{STM}(T, prg) = \{\text{lck} \mid lck \}^{\text{prg}}\) (cf. \(T'\)), it should be shown that \(\min(\{t_T^{d'''} + \text{TIME}(c^n, T) \mid T \in \text{Thrd}_i) \wedge \text{STM}(T, prg) = \{\text{lck} \mid lck \}^{\text{prg}}\}) \in \gamma_i(\bar{t}_{dl})\) is always fulfilled at the end of each loop iteration.

It is easy to see that the initial value of \(\bar{t}_{dl}\) (i.e., \(\bar{t}_T\)) is trivially safe since \(\forall T \in \text{Thrd}_i : t_T^{d'''} + \text{TIME}(c^n, T) \in \gamma_i(\bar{t}_T)\). It is also easy to see that for any thread, \(T \in \text{Thrd}_i\), such that \(\text{STM}(T, prg) \neq \{\text{lck} \mid lck \}^{\text{prg}}\), \(\bar{t}_{dl}\) is not at all affected by that loop iteration and is thus trivially safe at the end of that iteration. Note that:

- Since the initial value of \(\bar{t}_{dl}\) is \([-\infty, -\infty]\) for each considered thread and only the \(\bar{\cup}_i\) operator, where \(\bar{t}_{dl}\) itself is one of the arguments, is used to change the value of \(\bar{t}_{dl}\), it must be that after the \(\text{repeat}-\text{loop}\), \(-\infty \in \gamma_i(\bar{t}_{dl})\).

- The \(\text{repeat}\)-loop terminates for \(T'\) given that \(\text{REL}(lck)\) is not infinite since it is either terminated if \(0 \not\in \gamma_t(\text{ABSTIME}(c', T'))\) or \(\text{REL}(lck) \prec_T t_T^{d'''} + \text{ABSTIME}(c', T')\) (cf. Assumption 5.51), and thus \(\bar{t}_T^{d'''} + \text{ABSTIME}(c', T') \bar{t}_T \bar{t}_{\text{REL}(lck)} \cup_T [-\infty, -\infty] = T_T; \text{i.e., in the latter case, it is safely determined that enough iterations have been considered.}

- Since \(-\infty \in \gamma_t(\bar{t}_{dl})\), and \(t_T^{d'}\) (where \(i \in \{m_1, \ldots, m_2\}\) as defined above) is safely approximated, in each iteration of the \(\text{repeat}\)-loop, it must be
that $\tilde{t}_{dl} \sqcup \tilde{t} (\tilde{t}_{dl}''', \tilde{t}_l, \text{ABSTIME}(c', T'))$ (occurring in the next iteration of the loop) is a safe approximation of $t_{dl}''' + \text{TIME}(c', T')$; i.e., after the loop exits, $\tilde{t}_{dl}'$ is a safe approximation of when $T'$ acquires $lck$ unless the loop terminated prematurely due to $0 \in \text{ABSTIME}(c', T')$ since the loop then iterates at least $|\{m_1, \ldots, m_2\}| + 1$ times. However, note that if the loop exits and $0 \in \text{ABSTIME}(c', T')$ but $\tilde{t}_{dl}' = \bot_l$, it is actually the case that $\tilde{REL} (\tilde{t}_l lck) \preceq_l \tilde{t}_{dl}'$ and $\tilde{t}_{dl}'$ is thus safely approximating the point in time when $T'$ acquires $lck$ since the loop must have iterated at least $|\{m_1, \ldots, m_2\}| + 1$ times.

- $\forall c \in \text{Conf} : \forall T \in \text{Thrd} : \text{TIME}(c, T) \geq 0$ (Assumption 4.1).

Two (or rather, three) cases must now be considered for $T'$.

1. If $\text{TIME}(c^n, T') = 0$, then $t_{dl}'' + \text{TIME}(c^n, T') = t_{dl}'''$ (remember that $t_{dl}'' = t_{dl}'''$) and $0 \in \text{ABSTIME}(c'', T')$ (Assumption 5.51). Since $0 \in \gamma_i(\text{ABSTIME}(c'', T'))$, the repeat-loop might exit after too few iterations. If the loop exits after too few iterations, it must be that $\tilde{REL} (\tilde{t}_l lck) \not\preceq_l \tilde{t}_{dl}'$, and thus $\tilde{t}_{dl}' \neq \bot_l$. However, $\tilde{REL} (\tilde{t}_l lck)$ provides safe information for when $T'$ would acquire $lck$ since $\tilde{REL} (\tilde{t}_l lck) \in \gamma_i(\tilde{REL} (\tilde{t}_l lck))$. Thus, according to Assumption 5.51, it must be that $(t_{dl}'' + \text{TIME}(c^n, T')) \in \gamma_i (\tilde{t}_{dl}' \sqcup (\tilde{t} : \tilde{t}_{dl}'), \text{ABSTIME}(c'', T'))$, where $c'' = \langle [t, pc_T, \tilde{x}, T, \gamma_i(T = T' \wedge \tilde{t} : \tilde{t}_{dl}'), \tilde{t} \text{ABSTIME}(c'', T') \rangle$, where $\tilde{c}'' = ([t, pc_T, \tilde{x}, T, \gamma_i(T = T' \wedge \tilde{t} : \tilde{t}_{dl}'), \tilde{t} \text{ABSTIME}(c'', T'))$ (as defined on line 16 in DLLOCK) and $\tilde{t} = (\tilde{t}_{dl}' \sqcup \tilde{t}, \text{REL} (\tilde{t}_l lck) \sqcup_t (\tilde{t}_{dl}' \sqcup \tilde{t} [\infty, \infty])$ (as defined on line 15 in DLLOCK). But then, it is easy to see that $\min\{t_{dl}'' + \text{TIME}(c^n, T) \mid T \in \text{Thrd}_i \wedge \text{STM}(T, pc_T^n) = \{lck lck\} \} \in \gamma_i (\tilde{t}_{dl}' \sqcup (\tilde{t} : lck lck \text{ABSTIME}(c'', T')) \sqcup_t \tilde{t}_{dl})$.

2. If $\text{TIME}(c^n, T') > 0$, then two cases must be considered.

(a) If $0 \in \gamma_i(\text{ABSTIME}(c'', T'))$ (or for any other $c'$ of the repeat-loop for that matter), the proof is the same as that of 1 above.

(b) If $0 \notin \gamma_i(\text{ABSTIME}(c'', T'))$ (as well as for all $c'$ of the repeat-loop), then a $\tilde{t}_{dl}''$ such that $\tilde{REL} (\tilde{t}_l lck) \preceq_l \tilde{t}_{dl}''$ (and thus $\tilde{t}_{dl}' \neq \bot_l$) is derived. Since the loop then has iterated at least $|\{m_1, \ldots, m_2\}| + 1$ times (and thus, $t_{dl}'' = \tilde{t}_{dl}'$), it is easy to see that $\min\{t_{dl}'' + \text{TIME}(c^n, T) \mid T \in \text{Thrd}_i \wedge \text{STM}(T, pc_T^n) = \{lck lck\} \} \in \gamma_i (\tilde{t}_{dl}' \sqcup \tilde{t}_{dl})$. 

Chapter 5. Abstractly Interpreting PPL
Two (or rather, three) cases must now be considered for $T$.

1. If $\text{TAM} \cdot \forall m \text{loop exits}$, $\tilde{\tau}_t^{\alpha} (t a \in \text{TAM})$ in time when $T$ that $\tilde{\tau}_t^{\alpha}$ defined on line 16 in DLLOCK.

$\text{Conf} 1 \text{tam} \in \text{loop}$ then $\text{iterates at least} \tilde{\tau}_t^{\alpha}$ when $T$ $\text{TAM} \in \text{dl}$ is a safe approximation of when $T$ $\text{TAM} \in \text{TAM} (t a \in \text{TAM})$ occurs in the next iteration of $\text{TAM}$.

$\text{TAM} 0$ (Assumption 4.1).

For some thread $\tilde{\tau}_t^{\alpha}$ and $\tilde{\tau}_t^{\alpha}$ since the $\text{TAM}$ became free (i.e., was released).

If the lock, $lck \in \text{Lck}$, is not currently assigned to some other thread when some thread issues $lck$, the behavior is the same in both the concrete and abstract semantics in case the $lck$-issuing thread successfully acquires $lck$ (Lemma 5.55); i.e., the thread’s execution time is advanced based on when $lck$ became free (i.e., was released).

In the concrete semantics, the $lck$-statement is just considered to finish its execution, without successfully acquiring $lck$, after the (relative) time given by $\text{TAM}$, then a new instance of the same $lck$-statement is executed (cf. Tables 4.2 and 4.5); i.e., the thread is actively spinning on the lock. However, in the abstract semantics (cf. Tables 5.12 and 5.13 and Algorithms 5.11 and 5.12), a thread issuing $lck$ for some lock, $lck \in \text{Lck}$, that is currently acquired by some other thread would be frozen until it is assigned $lck$, if this ever occurs; i.e., the thread’s accumulated time would not be increased while it is waiting to be assigned $lck$. When (and if) the thread is later assigned $lck$, its accumulated execution time is advanced based on when $lck$ became free (i.e., was released).

![Note](Image)

**Note.** $\text{TAM}$ is not directly safe for the case that $\text{STM}(T, pc_T) = [\text{lck} \ to lck]^{pc_T} \land \text{OWN}(lck) \not\in T$. In the concrete case, $T$ will be executed in a spin-lock fashion, while in the corresponding abstract case, $T$ will be frozen (i.e., its accumulated time will not be updated). This case is further considered in the proof of Lemma 5.58.

$\text{TAM}$ is also not directly safe for the case that $T$ has been waiting but is now assigned $lck$ and $\tilde{\tau}_t^{\alpha} \rightarrow_t \text{TAM}(\tilde{\tau}, T) \not\in_t \text{R} \cup \text{L} \cup \text{W} (lck)$ since this would generate an extra abstract configuration for which $\tilde{\tau}_t^{\alpha} \not\in_t \text{R} \cup \text{L} \cup \text{W} (lck)$ and $\tilde{\tau}_t^{\alpha} \rightarrow_t \text{TAM}(\tilde{\tau}, T) \not\in_t \text{R} \cup \text{L} \cup \text{W} (lck)$; i.e., a catchup of the thread’s accumulated execution time would occur to approximate the concrete spin-waiting. This case is also further considered in the proof of...
Lemma 5.55 (Partial soundness of ACC TIME):
If the valid concrete configuration \( c @ \langle [T',pc_{T'},\bar{\tau}_{T'},T'] \rangle_{T' \in \text{Thrd}} \in \text{Conf} \) (cf. Definition 4.4), the abstract configuration \( c^0 @ \langle [T',pc_{T'},\bar{\tau}_{T'},\bar{\tau}^a_{T'}] \rangle_{T' \in \text{Thrd}_c} \in \text{Conf} \), and some thread, \( T \in \text{Thrd}_c \), are such that

\[
\text{Thrd}_c \subseteq \text{Thrd} \land
pc_T = pc_{T'} \land
\tau_{T'}^d \in \gamma_l(T_c') \land
((T \in \text{Thrd}^c_{ex} \land \forall lck \in \text{Lck} : (\text{STM}(T,pc_T) = [\text{lock} lck])^pcT \Rightarrow
(\text{OWN}(\bar{lck}) = T \land \text{OWN}(\bar{lck}) = T)) \iff T \in \text{Thrd}^c_{ex}) \land
\forall lck \in \text{Lck} : (\text{OWN}(\overline{\text{lck}}) = T \Rightarrow (\text{OWN}(\overline{\text{lck}}) = \overline{\text{OWN}(\text{lck})}) \land
\text{DL}(\overline{\text{lck}}) \not\in \gamma_l(\overline{\text{DL}(\overline{\text{lck}})}) \land
\text{POWN}(\overline{\text{lck}}) = \overline{\text{POWN}(\text{lck})} \land
\text{REL}(\overline{\text{lck}}) \in \gamma_l(\overline{\text{REL}(\overline{\text{lck}})}) \land
\min(\gamma_l(\overline{\text{DL}(\overline{\text{lck}})})) = -\infty),
\]

where \( \text{Thrd}^c_{ex} \) and \( \overline{lck} \), and \( \text{Thrd}^c_{ex} \) and \( \overline{lck} \), are as defined in Tables 4.5 and 5.13, respectively, then

\[
\tau_{T'} = \gamma_l(\text{ACC TIME}(\langle [T',pc_{T'},\bar{\tau}_{T'},\bar{\tau}^a_{T'}] \rangle_{T' \in \text{Thrd}_c} \in \text{Conf} ), \text{Thrd}^c_{ex}, T))
\]

where \( \tau_{T'} \) is as defined in Table 4.5.

**Explanation of the Lemma.** A thread must own, or be assigned, a given lock, \( lck \), in the given transition in case the thread executes \( \text{lock} lck \), and the properties of all locks owned by the thread in the concrete case must be safely approximated by the abstract case, for ACC TIME to safely approximate the concrete timing behavior of the thread.

**Proof.** Assume that the (valid; cf. Definition 4.4) configurations \( c @ \langle [T',pc_{T'},\bar{\tau}_{T'},T'] \rangle_{T' \in \text{Thrd}} \in \text{Conf} \) and \( c^0 @ \langle [T',pc_{T'},\bar{\tau}_{T'},\bar{\tau}^a_{T'}] \rangle_{T' \in \text{Thrd}_c} \in \text{Conf} \) and the thread \( T \in \text{Thrd}_c \) are as assumed in the lemma above.

For the sake of readability, let \( \bar{c} = \langle [T',pc_{T'},\bar{\tau}_{T'},\bar{\tau}^a_{T'}] \rangle_{T' \in \text{Thrd}_c} \in \text{Conf} \) when considering the following cases. Note that \( \text{Time} = \text{Intv} \).

1. If \( T \notin \text{Thrd}^c_{ex} \) (and thus, \( T \notin \text{Thrd}^c_{ex} \)), then \( \tau_{T'} = \tau_{T}^d \) (Table 4.5) and \( \text{ACC TIME}(\bar{c}, \text{Thrd}^c_{ex}, T) = \bar{\tau}_{T}^d \). Thus, \( \tau_{T'}^d \in \gamma_l(\text{ACC TIME}(\bar{c}, \text{Thrd}^c_{ex}, T)) \).
2. If \( T \in \text{Thrd}_{\text{exe}}^c \) and for some \( a \in \text{Aexp} \), \( b \in \text{Bexp} \), \( l \in \text{Lbl}_T \), \( r \in \text{Reg}_T \), \( x \in \text{Var} \) and \( \text{lck} \in \text{Lck} \), \( \text{STM}(T, pc_T) \in \{ \{ \text{skip} \}^{pc_T}, \{ \text{if} b \text{ goto } l \}^{pc_T}, \{ \text{store } r \text{ to } x \}^{pc_T}, \{ \text{load } r \text{ from } x \}^{pc_T}, \{ \text{unlock } lck \}^{pc_T} \} \), i.e., \( \forall lck' \in \text{Lck} : \text{STM}(T, pc_T) \neq \{ \text{lock } lck' \}^{pc_T} \) (and thus, \( T \in \text{Thrd}_{\text{exe}}^c \) since \( (T \in \text{Thrd}_{\text{exe}}^c \land \forall lck' \in \text{Lck} : (\text{STM}(T, pc_T) = \{ \text{lock } lck' \}^{pc_T} \Rightarrow (\text{OWN}(\text{lck}') = T \land \text{OWN}(\text{lck}) = T))) \iff T \in \text{Thrd}_{\text{exe}}^c \)), then \( \text{ACCTIME}(\tilde{c}, \text{Thrd}_{\text{exe}}^c, T) = \frac{a}{T} + _T \text{ABSTIME}(\tilde{c}, T) \) and by Table 4.5, \( \frac{a}{T} = \frac{a}{T} + \text{TIME}(c, T) \). Thus, by Lemma 5.52, \( \frac{a}{T} \in \gamma_t(\text{ACCTIME}(\tilde{c}, \text{Thrd}_{\text{exe}}^c, T)) \).

3. If \( T \in \text{Thrd}_{\text{exe}}^c \) and for some \( lck \in \text{Lck} \), \( \text{STM}(T, pc_T) = \{ \text{lock } lck \}^{pc_T} \) and \( \text{OWN}(\text{lck}) = T \), and thus, \( T \in \text{Thrd}_{\text{exe}}^c \) since \( (T \in \text{Thrd}_{\text{exe}}^c \land \forall lck' \in \text{Lck} : (\text{STM}(T, pc_T) = \{ \text{lock } lck' \}^{pc_T} \Rightarrow (\text{OWN}(\text{lck}') = T \land \text{OWN}(\text{lck}) = T))) \iff T \in \text{Thrd}_{\text{exe}}^c \), then several cases need to be considered. Note that \( \min(\gamma_t(\text{DL}(\text{lck}))) = -\infty \), for \( \forall lck' \in \text{Lck} : (\text{OWN}(\text{lck}') = T \Rightarrow \min(\gamma_t(\text{DL}(\text{lck}))) = -\infty \) and \( \text{OWN}(\text{lck}) = T \), and that \( T \) cannot acquire \( lck \) at any abstract time, \( i \), such that \( i < r \text{Lk}(\text{lck}) \), since \( lck \) has not been released at \( i \), or \( \text{DL}(\text{lck}) < i \text{Lk}(\text{lck}) \), since \( lck \) has not yet acquired it and the deadline for acquiring \( lck \) has expired). Then, in the concrete case, it must be that \( T \) cannot be the thread acquiring \( lck \) since \( \text{DL}(\text{lck}) \in \gamma_t(\text{DL}(\text{lck})), i \in \gamma_t(i), \text{TIME}(c, T) \in \gamma_t(\text{ABSTIME}(\tilde{c}, T)) \) and \( i + _T \text{TIME}(c, T) = \text{DL}(\text{lck}) \) whenever \( T \) acquires \( lck \) (Tables 4.2 and 4.5). But, then it cannot be that \( \tilde{sT}T(\text{lck}) = \text{unlocked} \land \text{DL}(\text{lck}) < i \text{Lk}(\text{lck}) \) since the corresponding branch cannot apply for the given case. (Note that such a \( \tilde{c} \) will not be further considered; cf. Algorithm 6.10 and Tables 5.12 and 5.13.)

(c) Note that the \( \tilde{sT}T(\text{lck}) = \text{unlocked} \land \text{DL}(\text{lck}) \not< i \text{Lk}(\text{lck}) \).
\textsc{abstime}(\tilde{c}, T)) \land (\tilde{t}_T^a \vdash_{\bar{t}} \textsc{abstime}(\tilde{c}, T)) \preceq_{\bar{t}} \textsc{rel}(\tilde{\tau}'' lck) \text{ conditioned branch, which applies to cases where T has been frozen for sure while waiting to acquire \textit{lck} but has now been assigned \textit{lck} (a catch-up calculation of the execution time of T would be performed), cannot be taken either. To see this, note that since \textit{c} is valid, it must be that \textsc{rel}(\tilde{\tau}'' lck) \leq t_T^a + \text{time}(c, T) \text{ (Definition 4.4). Then, since } t_T^a \in c, t_T^{\textsc{time}}(c, T) \in c(\textsc{abstime}(\tilde{c}, T)) \text{ (cf. Assumption 5.51), } \text{own}(\tilde{\tau}'' lck) = T \text{ and } \text{own}(\tilde{\tau}'' lck) = T \Rightarrow \text{rel}(\tilde{\tau}'' lck) \in c(\text{rel}(\tilde{\tau}'' lck)), \text{ it must be that } \tilde{t}_T^a \vdash_{\bar{t}} \textsc{abstime}(\tilde{c}, T) \not\preceq_{\bar{t}} \textsc{rel}(\tilde{\tau}'' lck). This branch is further considered when the freezing of threads is proven to be safe (cf. the proof of Lemma 5.58).

(d) If \textsc{sit}(\tilde{\tau}'' lck) = \text{unlocked} \land \textsc{dl}(\tilde{\tau}'' lck) \preceq_{\bar{t}} (\tilde{t}_T^a \vdash_{\bar{t}} \textsc{abstime}(\tilde{c}, T)) \land (\tilde{t}_T^a \vdash_{\bar{t}} \textsc{abstime}(\tilde{c}, T)) \preceq_{\bar{t}} \textsc{rel}(\tilde{\tau}'' lck) \land (\textsc{po\textsc{wn}}(\tilde{\tau}'' lck) = T \lor \textsc{rel}(\tilde{\tau}'' lck) \preceq_{\bar{t}} (\tilde{t}_T^a \vdash_{\bar{t}} \textsc{abstime}(\tilde{c}, T))) \text{ (T is assigned \textit{lck} but has not yet acquired it; the deadline for acquiring \textit{lck} has not expired; T has either been assigned \textit{lck} without having to wait (without being frozen), or T has been frozen but a catch-up calculation of the execution time for T has already been performed or T was the previous owner of \textit{lck}), then two cases must be considered.

i. If \textsc{po\textsc{wn}}(\tilde{\tau}'' lck) = T, then the sequential execution of the statements of a thread (cf. Tables 4.2 and 4.5) gives that T must acquire \textit{lck} at \tilde{t}_T^a \vdash_{\bar{t}} \textsc{abstime}(\tilde{c}, T), but not at an interval in time, \bar{t}, such that \textsc{dl}(\tilde{\tau}'' lck) \preceq_{\bar{t}} \bar{t}, because by then, some other thread must have already acquired \textit{lck} (since \textsc{dl}(\tilde{\tau}'' lck) \in c(\text{dl}(\tilde{\tau}'' lck))). Thus, it must be that \tilde{t}_T^a \in c((\tilde{t}_T^a \vdash_{\bar{t}} \textsc{abstime}(\tilde{c}, T)) \bar{t} \textsc{dl}(\tilde{\tau}'' lck)).

ii. If \textsc{rel}(\tilde{\tau}'' lck) \preceq_{\bar{t}} (\tilde{t}_T^a \vdash_{\bar{t}} \textsc{abstime}(\tilde{c}, T)), then Lemma 5.52 gives that \tilde{t}_T^a \in c((\tilde{t}_T^a \vdash_{\bar{t}} \textsc{abstime}(\tilde{c}, T)) \bar{t} \textsc{dl}(\tilde{\tau}'' lck)) since \tilde{t}_T^a = \textsc{dl}(\tilde{\tau}'' lck) \text{ (cf. Tables 4.2 and 4.5), } \textsc{dl}(\tilde{\tau}'' lck) \in c(\text{dl}(\tilde{\tau}'' lck)) \text{ and } \text{rel}(\tilde{\tau}'' lck) \in c(\text{rel}(\tilde{\tau}'' lck)).

(e) If \textsc{sit}(\tilde{\tau}'' lck) = \text{unlocked} \land \textsc{dl}(\tilde{\tau}'' lck) \preceq_{\bar{t}} (\tilde{t}_T^a \vdash_{\bar{t}} \textsc{abstime}(\tilde{c}, T)) \land (\tilde{t}_T^a \vdash_{\bar{t}} \textsc{abstime}(\tilde{c}, T)) \bar{t} \textsc{rel}(\tilde{\tau}'' lck) \neq \bot \land T \neq \textsc{po\textsc{wn}}(\tilde{\tau}'' lck) \text{ (T wants to acquire \textit{lck} and might have been assigned \textit{lck} without having to wait (without being frozen), or T has been frozen but a catch-up calculation of the execution time for T has already been performed; the deadline for acquiring \textit{lck} has not expired; the interval in time when \textit{lck} was released overlaps with the interval in
time when T can first acquire lck; T was not the previous owner of lck), then let \( \hat{t}'_T = \hat{t}'_T +_{\text{ABSTIME}} (\hat{c}, T) \), which is obviously a safe approximation of the first point in time at which T can acquire lck. Also let \( \hat{c}' \) be any configuration derived before (i.e., \( \hat{c}' = \hat{c} \)) or inside the repeat-loop. Note that \( \hat{t}'_T = \hat{t}_T \) is used to exit the loop in case \( \hat{DL}(\hat{t}'_T lck) \preceq_T (\hat{t}'_T +_{\text{ABSTIME}} (\hat{c}', T)) \) or \( 0 \in \gamma_T(\text{ABSTIME}(\hat{c}', T)) \), where the latter case means that a \( \hat{t}'_T \) such that \( \hat{REL}(\hat{t}'_T lck) \preceq_T \hat{t}'_T \) cannot be derived.

i. If \( \hat{DL}(\hat{t}'_T lck) \preceq_T \hat{t}'_T +_{\text{ABSTIME}} (\hat{c}', T) \), then it must be that at \( \hat{t}'_T +_{\text{ABSTIME}} (\hat{c}', T) \), some other thread will have acquired lck (hence, \( \hat{t}'_T \) safely approximates the last point in time when T can acquire lck). Thus, it must be that \( t''_T \in \gamma_T(\hat{DL}(\hat{t}'_T lck) \cap (R_El(\hat{t}'_T lck) \cup [\infty, \infty])) \) since \( \hat{DL}(\hat{t}'_T lck) \in \gamma_T(\hat{DL}(\hat{t}'_T lck)) \) and \( \hat{REL}(\hat{t}'_T lck) \in \gamma_T(\hat{REL}(\hat{t}'_T lck)) \).

ii. If \( 0 \in \gamma_T(\text{ABSTIME}(\hat{c}', T)) \) and also \( \hat{DL}(\hat{t}_T lck) \preceq_T \hat{t}_T +_{\text{ABSTIME}} (\hat{c}', T) \), then it must be that \( \hat{t}_T +_{\text{ABSTIME}} (\hat{c}', T) \), where \( \hat{t}_T = (\hat{t}'_T \cap [\infty, \infty]) \cap_T \hat{REL}(\hat{t}_T lck) \) and \( \hat{c}' = (\hat{T}' \cap_{\text{Thr}} T(\hat{T}' \cap \hat{c}_T \cup (T = T' \cap (T = T')) \cup \hat{c}_T \cap [\infty, \infty])) \) since \( \hat{REL}(\hat{t}_T lck) \in \gamma_T(\hat{REL}(\hat{t}_T lck)) \). Thus, it must be that \( t''_T \in \gamma_T(\hat{DL}(\hat{t}_T lck) \cap (R_El(\hat{t}_T lck) \cup [\infty, \infty])) \) since REL(\( \hat{t}_T lck \)) \( \in \gamma_T(\hat{REL}(\hat{t}_T lck)) \) and \( \hat{DL}(\hat{t}_T lck) \in \gamma_T(\hat{DL}(\hat{t}_T lck)) \).

iii. If \( 0 \notin \gamma_T(\text{ABSTIME}(\hat{c}', T)) \) and also \( \hat{DL}(\hat{t}_T lck) \preceq_T \hat{t}_T +_{\text{ABSTIME}} (\hat{c}', T) \), then it must be that, at some point, \( \hat{REL}(\hat{t}_T lck) \preceq_T \hat{t}_T \). Since \( \hat{REL}(\hat{t}_T lck) \in \gamma_T(\hat{REL}(\hat{t}_T lck)) \) and \( \hat{DL}(\hat{t}_T lck) \in \gamma_T(\hat{DL}(\hat{t}_T lck)) \), it is thus easy to see that \( t''_T \in \gamma_T((\hat{t}'_T \cup \hat{t}_T) \cap_T \hat{DL}(\hat{t}_T lck) \cap (R_El(\hat{t}_T lck) \cup [\infty, \infty])) \).

This concludes the proof.

It is important to notice that all the possible orders in which threads can acquire a lock in the concrete case are covered by the abstract transition relations, even though \( \text{Time} = \text{Intv} \). Since \( \text{Time} = \text{Intv}, \text{Thr}\_\text{exe} \), and thus the interleaving of the executed statements in the different threads of the program, might differ for concrete and abstract cases, as previously discussed. This means that even if some thread is the first (time-wise) in a set of threads to issue a lock-statement acting on some lock, \( lck \in \text{Lck} \), in the concrete case, some other
thread could issue its corresponding lock lck-statement first in the abstract case (note that the first case is covered by the abstraction as well).

The possible abstract combinations of the owner and the state for some lock, lck ∈ Lck, given a reference thread, T ∈ Thrd, in a lock state, ℓck, resulting from a transition using → prg are by definition as follows (cf. Tables 5.12 and 5.13).

1. \( \text{OWN}(\ellck) \notin \{ \bot_{\text{thrd}}, T \} \) – This means that T will be frozen if it issues lock lck and occurs when \( \text{OWN}(\ellck) \neq T \).

2. \( \text{OWN}(\ellck) = \bot_{\text{thrd}} \) – This occurs when \( \text{OWN}(\ellck) = \bot_{\text{thrd}} \). A safe (over-approximate) owner assignment will occur if T issues lock lck. The soundness is given by that it is trivially the case that for all concrete and abstract configurations consisting of the threads in Thrd, \( \{ T' \in \text{Thrd}_{\text{exe}} | \text{STM}(T', pc_{T'}) = [\text{lock lck}]^{pc_{T'}} \} \subseteq \{ T' \in \text{Thrd} | \exists l \in Lbl_{T'} : \text{STM}(T', l) = [\text{lock lck}] \} \); cf. Table 4.5.

3. \( \text{OWN}(\ellck) = T \land \text{STT}(\ellck) = \text{unlocked} \) – This means that T has not yet done lock lck, but some other thread has (with the result that T was assigned lck; cf. the discussion for state 2). If T issues lock lck within the deadline, it will successfully acquire lck. If it does not, there is no corresponding concrete situation described by the owner assignment, given that \( DL(\ellck) \in DL(\ellck) \), and thus, the configuration will be discontinued; cf. Algorithms 6.1 and 6.10, which are discussed in Chapter 6. This occurs when \( \text{OWN}(\ellck) = \bot_{\text{thrd}} \).

4. \( \text{OWN}(\ellck) = T \land \text{STT}(\ellck) = \text{locked} \) – This occurs when \( \text{OWN}(\ellck) = T \).

The possible transitions between these abstract states (as defined by → ax and → prg) are depicted in Figure 5.14. State 3 (a result from the over-approximate owner assignment performed by → prg) is needed since even if some thread acquires a lock first in the abstract case, it could be that some other thread wants to acquire the lock first in the corresponding concrete case. Lemma 5.56 gives that → prg covers all the possible concrete situations for lock owner assignments, regardless of which thread issues lock lck first in the abstract case; cf. a transition from state 2 to state 4, possibly via state 3.
Lemma 5.56 (Properties of owner assignment for lock-transitions):

If the valid concrete configurations (cf. Definition 4.4), abstract configurations, lock and threads

\[
\begin{align*}
c^0 &@ \langle [T, pc_i^0, z_i^0, t_i^0]_{T \in \text{Thrd}}, x_0, t^0 \rangle \in \text{Conf}, \\
c^i &@ \langle [T, pc_i^0, z_i^0, t_i^0]_{T \in \text{Thrd}}, x_i', t' \rangle \in \text{Conf}, \\
c^n &@ \langle [T, pc_n^0, z_n^0, t_n^0]_{T \in \text{Thrd}}, x_n, t^n \rangle \in \text{Conf}, \\
c^0 &@ \langle [T, pc_i^0, z_i^0, t_i^0]_{T \in \text{Thrd}}, x_0, t^0 \rangle \in \text{Coif}, \\
c^i &@ \langle [T, pc_i^0, z_i^0, t_i^0]_{T \in \text{Thrd}}, x_i', t' \rangle \in \text{Coif}, \\
c^k &@ \langle [T, pc_k^0, z_k^0, t_k^0]_{T \in \text{Thrd}}, x_k, t^k \rangle \in \text{Coif}, \\
lck' &\in \text{Lck}, \\
T' &\in \text{Thrd}_{\text{ck}} \text{ and} \\
T'' &\in \text{Thrd}_{\text{ck}},
\end{align*}
\]

are such that

\[
\begin{align*}
0 &\leq i < n, \\
\cdots &\rightarrow \cdots \rightarrow c^i \rightarrow \cdots \rightarrow c^n, \\
0 &\leq j < k, \\
\cdots &\rightarrow \cdots \rightarrow c^j \rightarrow \cdots \rightarrow c^k, \\
\text{Thrd}_{ck} &\subseteq \text{Thrd}_{cj} \subseteq \text{Thrd}_{co} \subseteq \text{Thrd}, \\
\text{STM}(T'', pc_{n'}^T) &= [\text{lock } lck']^{pc_{n'}^T}, \\
T'' &\in \text{Thrd}_{\text{exe}}, \\
\text{STM}(T', pc_{k'}^T) &= [\text{lock } lck']^{pc_{k'}^T}, \\
T' &\in \text{Thrd}_{ck}, \\
\forall h \in \{0, \ldots, k - 1\}: (T' \in \text{Thrd}_{\text{exe}} \Rightarrow \text{STM}(T', pc_{h'}^T) \neq [\text{lock } lck']^{pc_{h'}^T})
\end{align*}
\]
Table 4.5, and $\tilde{c}$ is defined in Table 5.13, then $\tilde{c}$ satisfies:

$$\begin{align*}
\text{OWN}(\tilde{c}) &= \text{OWN}(\tilde{c}^i) = \text{OWN}(\tilde{c}^j) = T' \land \\
\text{STT}(\tilde{c}) &= \text{STT}(\tilde{c}^i) = \text{STT}(\tilde{c}^j) = T' \land \\
\text{DL}(\tilde{c}) &= \text{DL}(\tilde{c}^i) = \text{DL}(\tilde{c}^j) \land \\
\min(\gamma(\text{DL}(\tilde{c}))) &= -\infty \land \\
t_{T'} + \text{TIME}(c',T') &\in \gamma(\text{DL}(\tilde{c}))
\end{align*}$$

**Explanation of the Lemma.** In the abstract case, $T'$ executes $\text{lock} \ lck'$ in a transition from $\tilde{c}^k$ and does not do so in any transition between $\tilde{c}^0$ and $\tilde{c}^k$. In the concrete case, $T'$ executes $\text{lock} \ lck'$ in a transition from $c^i$ and $\tilde{c}^k$. In the abstract case, $T''$ might execute $\text{lock} \ lck'$ in a transition from $\tilde{c}^j$, thus preceding $T'$. However, in the concrete case, $T''$ executes $\text{lock} \ lck'$ in a transition from $c^i$, thus succeeding $T'$. The case that $T'$ is assigned $lck'$ in a transition from $\tilde{c}^j$ is covered by $\tilde{c}$.

**Proof.** Assume that the valid concrete configurations, abstract configura-
to determine $\text{DL}(\llbracket_{j} lck'\rrbracket)$ and $\text{DL}(\llbracket_{k} lck')$, it is easy to see that $\min(\gamma_{i}(\text{DL}(\llbracket_{k} lck'))) = -\infty$ since $\text{DL}$ is used only if $\exists T \in \text{Thr}_e: \text{STM}(T, pc^{ej}_{i}) = [\text{lock}\ lck']pc^{ej}_{i}$ (cf. Ta-
ble 5.13) which is the case since \(\text{pc}^n_{\text{loc}} = \text{pc}^j_{\text{loc}}\), \(\text{OWN}(\text{lck}^j) = \bot_{\text{thrd}}\) and \(\text{OWN}(\text{lck}^{j+1}) = T'\) (cf. Algorithm 5.11).

Since \(T' \in \text{Thrd}^c_{\text{exe}}\), it must be that \(t^d_{\text{loc}} + \text{TIME}(c^i, T') = \min(\{t^d_{\text{loc}} + \text{TIME}(c^i, T) \mid T \in \text{Thrd}\})\), and since \(T'' \in \text{Thrd}^a_{\text{exe}}\), it must be that \(t^a_{\text{loc}} + \text{TIME}(c^n, T'') = \min(\{t^a_{\text{loc}} + \text{TIME}(c^n, T) \mid T \in \text{Thrd}\})\). But since \(c^i \xrightarrow{\text{prg}} \ldots \xrightarrow{\text{prg}} c^n\), it must be that \(t^d_{\text{loc}} + \text{TIME}(c^i, T') \leq t^a_{\text{loc}} + \text{TIME}(c^n, T'')\) (Lemma 4.2). Note that by choosing \(c^0\), \(c^m\), \(c^n\), \(\tilde{c}^0\) and \(\tilde{c}^i\) (defined by Lemma 5.54) to be \(c^0\), \(c^n\), \(\tilde{c}^i\) and \(\tilde{c}^j\) (defined by this proof), respectively, and assuming that \(\text{OWN}(\text{lck}^i) = \bot_{\text{thrd}}\) and \(\text{REL}(\text{lck}^i) \in \gamma_i(\text{REL}(\text{lck}^j))\) (which is actually not necessarily the case since \(T'\) acquires lck\(') in the transition between \(c^i\) and \(c^{i+1}\); however, note that this assumption is okay since if \(T'\) would not acquire lck\(')\), then \(\text{OWN}(\text{lck}^i) = \bot_{\text{thrd}}\) and \(\text{REL}(\text{lck}^i) \in \gamma_i(\text{REL}(\text{lck}^j))\) would hold since \(\text{OWN}(\text{lck}^i) = \bot_{\text{thrd}}\) and \(\text{REL}(\text{lck}^i) \in \gamma_i(\text{REL}(\text{lck}^j))\)), it is easy to see that \(t^a_{\text{loc}} + \text{TIME}(c^n, T'') \in \gamma_i(\text{DLLOCK}(\tilde{c}^j, \text{lck}^j))\) since \(\text{Thrd}^c_{\text{ex}} \subseteq \text{Thrd}^c_{\tilde{c}} \subseteq \text{Thrd}_{\text{exe}}, \text{STM}(T'', \text{pc}^n_{\text{loc}}) = \text{lock}\text{lck}^k\text{pc}^n_{\text{loc}}, T'' \in \text{Thrd}^a_{\text{exe}}\) and \(t^a_{\text{loc}} \in \gamma_i(\text{DLOCK}(\tilde{c}^j, \text{lck}^j))\) (Lemma 5.54). But then, since \(\min(\gamma_i(\text{DLOCK}(\tilde{c}^j, \text{lck}^j))) = -\infty\), \(\text{DLOCK}(\text{lck}^k) = \text{DLLOCK}(\tilde{c}^j, \text{lck}^j)\) and \(t^a_{\text{loc}} + \text{TIME}(c^n, T'') \leq t^a_{\text{loc}} + \text{TIME}(c^n, T'')\), it must be that \(t^d_{\text{loc}} + \text{TIME}(c^i, T') \in \gamma_i(\text{DLOCK}(\text{lck}^k))\) which concludes the proof. \(\blacksquare\)

Three lemmas will be presented in order to prove that the abstract transitions described by \(\xrightarrow{\text{prg}}\) safely approximate the concrete transitions described by \(\xrightarrow{\text{prg}}\). The lemmas hold given that the concrete transition sequences are finite in length (i.e., given that they terminate) and that either no thread issues a load-statement on a global variable or that the thread issuing the load-statement is the sole thread in \(\text{Thrd}_{\text{exe}}\) in any step of the transition sequence. The first lemma (Lemma 5.57) states that the halt-, skip-, :=-, if-, load-, store- and unlock-statements, and also the lock-statement if the issuing thread immediately is assigned the lock, are safely approximated. Note that a variable is considered global if it could transfer data between two or more threads (cf. Algorithm 6.5, defined on page 169).

**Lemma 5.57 (Soundness of \(\xrightarrow{\text{prg}}\), no frozen thread):**

*If the valid concrete configurations (cf. Definition 4.4), abstract configurations*
and thread

\[\begin{align*}
c^0 & @ \langle ]T, pc^0_T, c^0_T, t^0_T T \in \text{Thrd}, x^0, \bar{0} \rangle \in \text{Conf}, \\
c^n & @ \langle ]T, pc^n_T, c^n_T, t^n_T T \in \text{Thrd}, x^n, \bar{n} \rangle \in \text{Conf}, \\
c^0 & @ \langle ]T, pc^0_T, c^0_T, t^0_T T \in \text{Thrd}, x^0, \bar{0} \rangle \in \text{Conf}, \\
c^k & @ \langle ]T, pc^k_T, c^k_T, t^k_T T \in \text{Thrd}, x^k, \bar{k} \rangle \in \text{Conf}, \quad \text{and} \\
T' & \in \text{Thrd}_c^k
\end{align*}\]

are such that

\[\begin{align*}
0 & \leq n, \\
c^0 & \xrightarrow{\text{prog}} \ldots \xrightarrow{\text{prog}} c^n, \\
0 & \leq k, \\
c^0 & \xrightarrow{\text{prog}} \ldots \xrightarrow{\text{prog}} c^k, \\
\text{Thrd}_c^k & \subseteq \text{Thrd}_c^0 \subseteq \text{Thrd}, \\
p c^0_T & = pc^0_T, \\
\nu^0_T & \in \gamma_{\text{reg}}(\bar{x}^0), \\
t^0_T & \in \gamma_l(\bar{1}^0_T), \\
\exists x' & \in \gamma_{\text{var}}(\bar{x}^0) : \forall x \in \text{Var} : \forall T \in \text{Thrd} : ((x^0 & x) T) \subseteq ((x' & x) T), \\
\forall lck & \in \text{Lck} : ((\text{OWN}(\bar{0} lck) \neq \perp_{\text{thrd}} \Rightarrow (\text{SST}(\bar{0} lck) = \text{SST}(\bar{0} lck) \land \\
\text{OWN}(\bar{0} lck) = \text{OWN}(\bar{0} lck) \land \\
\text{DL}(\bar{0} lck) \in \gamma_l(\text{DL}(\bar{0} lck)) \land \\
\text{POWN}(\bar{0} lck) = \text{POWN}(\bar{0} lck) \land \\
\text{REL}(\bar{0} lck) \in \gamma_l(\text{REL}(\bar{0} lck)) \land \\
\min(\gamma_l(\text{DL}(\bar{0} lck))) = -\infty)) \land \\
(\text{OWN}(\bar{0} lck) = \perp_{\text{thrd}} \Rightarrow ((\text{OWN}(\bar{0} lck) = \text{OWN}(\bar{0} lck) \lor \\
(\text{OWN}(\bar{0} lck) = T' \land \\
\text{SST}(\bar{0} lck) = \text{unlocked} \land \\
t^0_T + \text{TIME}(c^0, T') \in \gamma_l(\text{DL}(\bar{0} lck)) \land \\
\min(\gamma_l(\text{DL}(\bar{0} lck))) = -\infty)) \land \\
\text{POWN}(\bar{0} lck) = \text{POWN}(\bar{0} lck) \land \\
\text{REL}(\bar{0} lck) \in \gamma_l(\text{REL}(\bar{0} lck)) \land \\
\text{STM}(T', pc^0_T) = \lceil \text{lock lck} \rceil_{pc^0_T} \Rightarrow \\
(\bar{0} lck = \bar{0} lck \land \\
\bar{k} lck = \bar{0} lck))\).
\end{align*}\]
∀i ∈ \{0, \ldots, n - 1\} : T' \not\in \text{Thrd}^i_{exe},
STM(T', pc^i_{T'}) \not\in [\text{halt}]^{pc^i_{T'}} \Rightarrow T' \in \text{Thrd}^n_{exe},
∀i ∈ \{0, \ldots, k - 1\} : T' \not\in \text{Thrd}^q_{exe},
STM(T', pc^k_{T'}) \not\in [\text{halt}]^{pc^k_{T'}} \Rightarrow T' \in \text{Thrd}^k_{exe}, \text{ and }
∀i ∈ \{0, \ldots, k\} : (|\text{Thrd}^i_{exe}| \not\in \{0\} \lor
\{T ∈ \text{Thrd}^i_{exe} | \exists r ∈ \text{Reg}_T : \exists x ∈ \text{Var}_x : STM(T, pc^i_{T'}) = [\text{load } r \text{ from } x]^{pc^i_{T'}} = \emptyset\}),

where for all i ∈ \{0, \ldots, n\}, \text{Thrd}^i_{exe} \text{ is as defined in Table 4.5, for all i ∈ } \{0, \ldots, k\}, \text{Thrd}^q_{exe} \text{ is as defined in Table 5.13, and Var}_x \text{ contains all } x ∈ \text{Var} \text{ such that } x \text{ can be written to by one thread and read from by another thread (i.e., there is a data dependency between the threads), then } \xrightarrow{\text{prg}} \text{ satisfies:}

∀c @ ([T, pc_T, r_T, t^a_T]_{T ∈ \text{Thrd}_k}; \tilde{x}, \emptyset) ∈ \text{Conf} :
(c^n \xrightarrow{\text{prg}} c \Rightarrow \exists \tilde{c} @ ([T, pc_T, r_T, t^a_T]_{T ∈ \text{Thrd}_k}; \tilde{x}, \emptyset) ∈ \text{Conf} :
(\tilde{c}^k \xrightarrow{\text{prg}} \tilde{c} \land
pc_T = pc_T^k \land
r_T \in \gamma_{\text{reg}}(\tilde{r}_T) \land
t^a_T \in \gamma_t(\tilde{t}^a_T) \land
\exists \tilde{x}' ∈ \gamma_{\text{var}}(\tilde{x}) : (\forall x ∈ \text{Var} : ((\tilde{x} x, T') \subseteq ((\tilde{x}' x, T')) \land
\forall lck ∈ \text{Lck} : ((\text{OWN}(\tilde{lck}) = T' \lor \text{OWN}(\tilde{lck}) = T') \Rightarrow
(\text{STM}(\tilde{lck}) = \text{STM}(\tilde{lck}) \land
\text{OWN}(\tilde{lck}) = \text{OWN}(\tilde{lck}) \land
\text{DL}(\tilde{lck}) ∈ \gamma_t(\text{DL}(\tilde{lck}) \land
\text{POWN}(\tilde{lck}) = \text{POWN}(\tilde{lck}) \land
\text{REL}(\tilde{lck}) ∈ \gamma_t(\text{REL}(\tilde{lck}) \land
\text{min}(\gamma_t(\text{REL}(\tilde{lck}))) = -\infty))))\) )

\textbf{Explanation of the Lemma.} Only thread T is considered. All scenarios, except loading the values of global variables in non-sequential situations and trying to acquire locks that are assigned to, or acquired by, other threads, are covered. In case T executes lock lck, the lemma only covers the case that lck is immediately assigned to T (or has already been so on a previous transition). Basically, the lemma says that for every possible (covered) concrete transition, there is a corresponding abstract transition which safely approximates the concrete one.
PROOF. Assume that the valid concrete configurations (cf. Definition 4.4), abstract configurations and thread
\[c^0 \in \{ [T, pc_T^0, z_T^0, t_T^0]_{T \in \text{Thrd}}, \infty^0, 0^0 \} \in \text{Conf},\]
\[c^n \in \{ [T, pc_T^n, z_T^n, t_T^n]_{T \in \text{Thrd}}, \infty^n, 0^n \} \in \text{Conf},\]
\[c^0 \in \{ [T, pc_T^0, z_T^0, t_T^0]_{T \in \text{Thrd}}, \infty^0, 0^0 \} \in \text{Conf},\]
\[c^k \in \{ [T, pc_T^k, z_T^k, t_T^k]_{T \in \text{Thrd}}, \infty^k, 0^k \} \in \text{Conf},\]
\[T' \in \text{Thrd}_{\infty}c^k\]
are as assumed in the lemma above.

First note that:
• Since \( \forall i \in \{0, \ldots, n - 1\} : T' \not\in \text{Thrd}_{\infty}c^i \), it must be that \( pc_{T'}^i = pc_{T'}^0 \), \( z_{T'}^i = z_{T'}^0 \), \( t_{T'}^i = t_{T'}^0 \) and \( \forall lck \in \text{Lck} : (\text{OWN}(\infty^0 lck) = T' \Rightarrow \infty^n lck = \infty^0 lck) \) (cf. Table 4.5).
• Since \( \forall i \in \{0, \ldots, k - 1\} : T' \not\in \text{Thrd}_{\infty}c^i \), it must be that \( pc_{T'}^k = pc_{T'}^0 \), \( z_{T'}^k = z_{T'}^0 \), \( t_{T'}^k = t_{T'}^0 \), and \( \forall lck \in \text{Lck} : (\text{OWN}(\infty^0 lck) = T' \Rightarrow (\infty^k lck = \infty^0 lck \land \min(\gamma(\text{DL}(\infty^k lck))) = -\infty)) \).
• Since \( pc_{T'}^n = pc_{T'}^0 \), \( z_{T'}^n = z_{T'}^0 \), \( t_{T'}^n = t_{T'}^0 \), \( \forall lck \in \text{Lck} : (\text{OWN}(\infty^0 lck) = T' \Rightarrow \infty^n lck = \infty^0 lck \land \min(\gamma(\text{DL}(\infty^k lck))) = -\infty)) \), \( pc_{T'}^0 = pc_{T'}^0 \), \( z_{T'}^0 = z_{T'}^0 \), \( t_{T'}^0 = t_{T'}^0 \), \( \forall lck \in \text{Lck} : (\text{OWN}(\infty^0 lck) = T' \Rightarrow (\text{STT}(\infty^0 lck) = \text{STT}(\infty^0 lck) \land \text{OWN}(\infty^0 lck) = \text{OWN}(\infty^0 lck) \land \text{DL}(\infty^0 lck) = \text{DL}(\infty^0 lck) \land \text{POWN}(\infty^0 lck) = \text{POWN}(\infty^0 lck) \land \text{REL}(\infty^0 lck) = \text{REL}(\infty^0 lck) \land \min(\gamma(\text{DL}(\infty^0 lck))) = -\infty)) \), it must be that (if a thread does not execute any statement, the state of its local properties and its owned locks do not change):
\[pc_{T'}^0 = pc_{T'}^k \land\]
\[z_{T'}^n \in \gamma(\text{DL}(\infty^k lck)) \land\]
\[t_{T'}^n \in \gamma(\text{DL}(\infty^k lck)) \land\]
\[\forall lck \in \text{Lck} : (\text{OWN}(\infty^0 lck) = T' \Rightarrow (\text{STT}(\infty^0 lck) = \text{STT}(\infty^0 lck) \land\]
\[\text{OWN}(\infty^0 lck) = \text{OWN}(\infty^0 lck) \land\]
\[\text{DL}(\infty^0 lck) = \text{DL}(\infty^0 lck) \land\]
\[\text{POWN}(\infty^0 lck) = \text{POWN}(\infty^0 lck) \land\]
\[\text{REL}(\infty^0 lck) = \text{REL}(\infty^0 lck) \land\]
\[\min(\gamma(\text{DL}(\infty^0 lck))) = -\infty)) \]
• Since \( e^0 \rightarrow \ldots \rightarrow e^n \) and \( \forall i \in \{0, \ldots, n-1\} : T' \not\in \text{Thrd}_{exe}^{e_i} \), it must be that for all \( x \in \text{Var} \), \(((x^n x) T') = (((x^0 x) T') if no thread writes to \( x \) in the sequence \( e^0 \rightarrow \ldots \rightarrow e^n \), or \(((x^n x) T') = \emptyset \) if some other thread has written to \( x \) in the given sequence (cf. Table 4.5). Thus, \( \forall x \in \text{Var} : ((x^n x) T') \subseteq (((x^0 x) T') \) (the possible concrete write history on a variable by some thread can only decrease when the thread does not execute any \( \text{store} \)-statement).

• Since \( \exists x' \in \gamma_{\text{var}}(\tilde{x}^0) : \forall x \in \text{Var} : (((x^0 x) T') \subseteq (((x' x) T'), e^0 \rightarrow \ldots \rightarrow e^n \), \( \tilde{c}^0 \rightarrow \ldots \rightarrow \tilde{c}^k \), \( \forall i \in \{0, \ldots, n-1\} : T' \not\in \text{Thrd}_{exe}^{e_i} \), \( \forall i \in \{0, \ldots, k - 1\} : T' \not\in \text{Thrd}_{exe}^{e_i} \) and TRIM is safe (Lemma 5.28), it must be that \( \exists x' \in \gamma_{\text{var}}(\tilde{x}^k) : \forall x \in \text{Var} : (((x^n x) T') \subseteq (((x' x) T') \) (a safe abstract write history on a variable by some thread remains safe when the thread does not execute any statement).

• Since \( \forall i \in \{0, \ldots, k\} : ((\text{Thrd}_{exe}^{e_i} \not\supset 1 \lor \{T \in \text{Thrd}_{exe}^{e_i} \mid \exists r \in \text{Reg}_T : \exists x \in \text{Var}_g : \text{STM}(T, pc^{e_i}_T) = [\text{load } r \text{ from } x]^{pc^{e_i}_T} = \emptyset \}), it must be that \( \forall i \in \{0, \ldots, k\} : ((\{T \in \text{Thrd}_{exe}^{e_i} \mid \exists r \in \text{Reg}_T : \exists x \in \text{Var}_g : \text{STM}(T, pc^{e_i}_T) = [\text{load } r \text{ from } x]^{pc^{e_i}_T} \} \neq \emptyset \Rightarrow [\text{Thrd}_{exe}^{e_i} = 1] \). This means that if some thread in \( \text{Thrd}_{exe}^{e_i} \), where \( i \in \{0, \ldots, k\} \), performs a \( \text{load} \)-statement, there is only one single thread in \( \text{Thrd}_{exe}^{e_i} \); thus that thread performs the \( \text{load} \)-statement. It is then easy to see, from the definition of \( \text{Thrd}_{exe}^{e_i} \), that there cannot occur any other write than those represented by \( \tilde{x}^i \) such that it could affect the \( \text{load} \)-statement of the thread in \( \text{Thrd}_{exe}^{e_i} \) (cf. Assumption 5.51) – thus, it must be that \( \tilde{x}^k \) (and also all \( \tilde{x}^i \), where \( i \in \{0, \ldots, k\} \) contains a safe write history (cf. Definition 5.19).

• Since, trivially, \( \forall lck \in \text{Lck} : \{T \in \text{Thrd}_{exe}^{e_i} \cap \text{Thrd}_{lck}^{ek} \mid \text{STM}(T, pc^{ek}_T) = [\text{lock } lck]^{pc^{ek}_T} \} \subseteq \{T \in \text{Thrd}_{lck}^{ek} \mid \exists l \in \text{Lbl}_T : \text{STM}(T, l) = [\text{lock } lck]^{l} \} \) (the set of threads executing \( \text{lock } lck \) is a subset of the set containing all threads that could execute \( \text{lock } lck \) somewhere in the program), it must be that if \( T' \) can be assigned a lock in the concrete case, it can also be assigned the lock in the corresponding abstract case.

• If, for some \( lck \in \text{Lck}, \text{STM}(T', pc^{ek}_T) = [\text{lock } lck]^{pc^{ek}_T} \), it must be that \( \text{own}(\tilde{x}^k lck) = T' \), since \( \forall i \in \{0, \ldots, k - 1\} : T' \not\in \text{Thrd}_{exe}^{e_i} \) and \( T' \in \)
\(\text{Thrd}_{\text{exe}}^{\delta}(T')\) is assigned the ownership of \(lck\).

- Since \(T' \in \text{Thrd}_{\text{ck}}\), \(\text{Thrd}_{\text{ck}} \subseteq \text{Thrd}\), \(pc_{\text{exe}}^{\delta} = pc_{\text{ck}}^{\delta}, t_{\text{ck}}^{\delta} \in \gamma_{t}(T')\), \(\forall lck \in \text{Lck}: (\text{OWN}(\delta^{0} lck) = T' \Rightarrow (\text{STM}(\delta^{0} lck) = \text{SIT}(\delta^{0} lck) \wedge \text{OWN}(\delta^{0} lck) = \text{OWN}(\delta^{k} lck) \wedge \text{DL}(\delta^{0} lck) \in \gamma_{t}(\text{DL}(\delta^{k} lck)) \wedge \text{POWN}(\delta^{0} lck) = \text{POWN}(\delta^{k} lck) \wedge \text{REL}(\delta^{0} lck) \in \gamma_{t}(\text{REL}(\delta^{k} lck)))\), \(T' \in \text{Thrd}_{\text{exe}}^{\delta}, T' \in \text{Thrd}_{\text{exe}}^{\delta}, \forall lck \in \text{Lck}: (\text{STM}(T', pc_{\text{exe}}^{\delta}) = [\text{lock } lck]^{\text{pc}_{\text{exe}}^{\delta}} \Rightarrow \text{OWN}(\delta^{kn} lck) = T')\), it must be that \(t_{\text{ck}}^{\delta} \in \gamma_{t}(\text{ACCTIME}(\{\text{STM}(\text{Thrd}_{\text{exe}}^{\delta}, x, lck)\})\), \(\forall lck \in \text{Lck}: (\text{STM}(T', pc_{\text{exe}}^{\delta}) = [\text{lock } lck]^{\text{pc}_{\text{exe}}^{\delta}} \Rightarrow (\text{OWN}(\delta^{k} lck) = T' \wedge \text{OWN}(\delta^{kn} lck) = T'))\) (Lemma 5.55, where \(t_{\text{ck}}^{\delta}\) is derived from \(c^{n} \rightarrow_{\text{prg}} [\text{STM}(\text{Thrd}_{\text{exe}}^{\delta}, x, lck)\})\) and \(\delta^{kn}\) are defined as in Tables 4.5 and 5.13, respectively (ACCTIME safely approximates the corresponding concrete timing behavior).

- Since \(\forall lck \in \text{Lck}: (\text{OWN}(\delta^{0} lck) = \bot_{\text{thrd}} \Rightarrow (\text{STM}(T', pc_{\text{ck}}^{0}) = [\text{lock } lck]^{pc_{\text{ck}}^{0}} \Rightarrow (\text{OWN}(\delta^{n} lck) \wedge \delta^{k} lck = \delta^{0} lck))\), \(\forall i \in \{0, \ldots, n-1\} : T' \not\in \text{Thrd}_{\text{exe}}^{\delta}, \text{STM}(T', pc_{\text{exe}}^{\delta}) \neq [\text{halt}^{pc_{\text{exe}}^{\delta}} \Rightarrow T' \in \text{Thrd}_{\text{exe}}^{\delta}\) and \(\text{OWN}(\delta^{kn} lck) = T'\), it must be that \(T'\) immediately acquires \(lck\) (i.e., without any other thread acquiring and possibly releasing \(lck\) in the sequence \(c^{0} \rightarrow_{\text{prg}} \ldots \rightarrow_{\text{prg}} c^{n}\) if \(\text{STM}(T', pc_{\text{ck}}^{\delta}) = [\text{lock } lck]^{pc_{\text{exe}}^{\delta}}\), both in the concrete and abstract cases (based on \(\delta^{n}\) and \(\delta^{\delta}\)).

- If, for some lock, \(lck' \in \text{Lck}\), \(\text{STM}(T', pc_{\text{ck}}^{0}) = [\text{lock } lck']^{pc_{\text{exe}}^{\delta}}\) and \(\text{OWN}(\delta^{0} lck') = \bot_{\text{thrd}}\), it must be that \(\min(\{t_{\text{ck}}^{\delta} + \text{TIME}(c^{n}, T) \mid T \in \text{Thrd}\})\) \(\gamma_{t}(\text{DLLOCK}(\delta^{j}, lck'))\), since \(T' \in \text{Thrd}_{\text{ck}}\), \(\text{Thrd}_{\text{ck}} \subseteq \text{Thrd}_{\text{exe}}\), \(T' \in \text{Thrd}_{\text{exe}}\), \(m, n\), and \(j\) (cf. Lemma 5.54) can be chosen to be \(0\), \(n\), and \(k\) (given by this proof), respectively, \(t_{\text{ck}}^{\delta} \in \gamma_{t}(T')\), \(t_{\text{ck}}^{\delta} = t_{\text{ck}}^{\delta}, \ T_{\text{ck}}^{\delta} = T_{\text{ck}}^{\delta}, \ pc_{\text{ck}}^{\delta} = pc_{\text{ck}}^{\delta}, \ T_{\text{ck}}^{\delta} = T_{\text{ck}}^{\delta}, \ T' \in \text{Thrd}_{\text{exe}}^{\delta}\) and (since \(\forall lck \in \text{Lck}: (\text{OWN}(\delta^{0} lck) = \bot_{\text{thrd}} \Rightarrow (\text{STM}(T', pc_{\text{ck}}^{0}) = [\text{lock } lck]^{pc_{\text{ck}}^{0}} \Rightarrow (\text{OWN}(\delta^{n} lck) \wedge \delta^{k} lck = \delta^{0} lck))\)) \(\text{REL}(\delta^{kn} lck') \in \gamma_{t}(\text{REL}(\delta^{kn} lck'))\) \(\text{REL}(\delta^{kn} lck') \in \gamma_{t}(\text{REL}(\delta^{kn} lck'))\) (Lemma 5.54). Thus, \(t_{\text{ck}}^{\delta} + \text{TIME}(c^{n}, T') \in \gamma_{t}(\text{DLLOCK}(\delta^{k}, lck'))\), since \(T' \in \text{Thrd}_{\text{exe}}^{\delta}\) which means that \(t_{\text{ck}}^{\delta} + \text{TIME}(c^{n}, T') = \min(\{t_{\text{ck}}^{\delta} + \text{TIME}(c^{n}, T) \mid T \in \text{Thrd}\})\) (cf. Table 4.5) (the abstract deadline for acquiring \(lck'\), derived from \(\delta^{\delta}\), covers the case that \(T'\) concretely acquires \(lck'\) in \(c^{n}\)).
The proof will now be conducted by considering the different statements that $T'$ could issue in $c^0$ (i.e., in $c^n$).

1. If $\text{STM}(T', pc^0_T) = [\text{halt}]^{pc^0_T}$, then it must be that $T' \not\in \text{Thrd}^{ck}_{\text{exe}}$. Thus, it must be that $c^n \xrightarrow{prg} c$, where $c@\langle[|T, pc_T, x_T, t^0_T|_T | T \in \text{Thrd}^l, \emptyset] \rangle$ is such that $pc_T = pc^0_T$, $x_T = x^0_T$, $t^0_T = t^0_T$, $\forall lck \in \text{Lck} : (\text{OWN}(\emptyset lck) = T' \Rightarrow \emptyset lck = \emptyset^n lck)$ and $\forall x \in \text{Var} : ((x^n x) T') \subseteq ((x^n x) T')$, provided that $\exists T \in \text{Thrd} : \text{STM}(T, pc^0_T) \neq [\text{halt}]^{pc^0_T}$ (otherwise $\xrightarrow{prg}$ is not applicable; cf. Table 4.5).

Note that $T' \not\in \text{Thrd}^{ck}_{\text{exe}}$ and choose $\tilde{c}@\langle[|T, pc^0_T, \tilde{x}_T, \tilde{t}^0_T|_T | T \in \text{Thrd}^l, \emptyset] \rangle$ such that $\tilde{c}^k \xrightarrow{prg} \tilde{c}$, i.e., $pc^k_T = pc^0_T$, $\tilde{x}_T = x^k_T$, $\tilde{t}^0_T = t^0_T$, $\forall lck \in \text{Lck} : (\text{OWN}(\emptyset lck) = T' \Rightarrow (\emptyset lck = \emptyset^k lck \land \text{min}(\gamma_t(\text{DL}(\emptyset lck))) = -\infty))$.

Note that $\tilde{x}$ must still be such that for all $x \in \text{Var}$, $(x^n x)$ is a safe approximation of the writes performed on $x$ by $T'$ since TRIM is safe (Lemma 5.28). Thus, it must be that:

\[
\begin{align*}
pc_T &= pc^k_T \land \\
\tau_T &\in \gamma_{\text{reg}}(\tilde{x}_T) \land \\
t^0_T &\in \gamma_t(\tilde{t}^0_T) \land \\
\exists \tilde{x} &\in \gamma_{\text{var}}(\tilde{x}) : \forall x \in \text{Var} : ((x^n x) T') \subseteq ((x^n x) T') \land \\
\forall lck &\in \text{Lck} : (\text{OWN}(\emptyset lck) = T' \lor \text{OWN}(\emptyset lck) = T') \Rightarrow \\
\text{STT}(\emptyset lck) &= \text{STT}(\emptyset lck) \land \\
\text{OWN}(\emptyset lck) &= \text{OWN}(\emptyset lck) \land \\
\text{DL}(\emptyset lck) &\in \gamma_t(\text{DL}(\emptyset lck)) \land \\
\text{POWN}(\emptyset lck) &\in \gamma_t(\text{POWN}(\emptyset lck)) \land \\
\text{REL}(\emptyset lck) &\in \gamma_t(\text{REL}(\emptyset lck)) \land \\
\text{min}(\gamma_t(\text{DL}(\emptyset lck))) &\in -\infty)
\end{align*}
\]

2. If, for some $a \in \text{Aexp}$, $b \in \text{Bexp}$, $l \in \text{Lbl}_T$, $r \in \text{Reg}_T$, $x \in \text{Var}$ and $lck \in \text{Lck}$, $\text{STM}(T', pc^0_T) \in \{|\text{skip}|^{pc^0_T}, |\text{r} := a|^{pc^0_T}, |\text{if b goto l}|^{pc^0_T}, |\text{store r to x}|^{pc^0_T}, |\text{unlock lck}|^{pc^0_T}\}$, then let the configuration $c@\langle[|T, pc_T, \tau_T, t^0_T|_T | T \in \text{Thrd}^l, \emptyset] \rangle$ be such that $c^n \xrightarrow{prg} c$ and choose $\tilde{c}@\langle[|T, pc^0_T, \tilde{x}_T, \tilde{t}^0_T|_T | T \in \text{Thrd}^l, \emptyset] \rangle$ such that $\tilde{c}^k \xrightarrow{prg} \tilde{c}$. Thus, since $\forall i \in \{0, \ldots, n - 1\} : T' \not\in \text{Thrd}^{ck}_{\text{exe}}$, $T' \in \text{Thrd}^{ck}_{\text{exe}}$, $\forall i \in \{0, \ldots, k - 1\} : T' \not\in \text{Thrd}^{ck}_{\text{exe}}$, $T' \in \text{Thrd}^{ck}_{\text{exe}}$, $\xrightarrow{ax}$ is a safe approximation of $\xrightarrow{ax}$ (Lemma
5.50), TRIM is safe (Lemma 5.28), $\text{Thrd}_{c0} \subseteq \text{Thrd}$ and $\text{ACCTIME}$ is safe (Lemma 5.55), it must be that:

$$pc_{T'} = pc_{T'}^c \land$$

$$\exists x' \in \gamma_{\text{var}}(\tilde{x}_{T'}) \land$$

$$\exists x' \in \gamma_{\text{var}}(\tilde{x}_{T'}) : (\forall x \in \text{Var} : ((\exists x') T') \subseteq ((\exists x') T') \land$$

$$\forall \text{lck} \in \text{Lck} : ((\text{OWN}(\emptyset \text{lck}) = T' \lor \text{OWN}(\emptyset \text{lck}) = T') \implies$$

$$(\text{STM}(\emptyset \text{lck}) = \text{STM}(\emptyset \text{lck}) \land$$

$$\text{OWN}(\emptyset \text{lck}) = \text{OWN}(\emptyset \text{lck}) \land$$

$$\text{DL}(\emptyset \text{lck}) = \gamma_{\text{DL}}(\emptyset \text{lck}) \land$$

$$\text{POWN}(\emptyset \text{lck}) = \text{POWN}(\emptyset \text{lck}) \land$$

$$\text{REL}(\emptyset \text{lck}) = \gamma_{\text{REL}}(\emptyset \text{lck}) \land$$

$$\text{min}(\gamma_{\text{DL}}(\emptyset \text{lck})) = -\infty))$$

Note that in the case $\text{STM}(T', pc_{T'}^0) = [\text{if } b \text{ goto } l] pc_{T'}^0$, $c$ can be chosen so that the corresponding branch to that taken in $c$ is taken since $T_{r0}^0 \in \gamma_{\text{reg}}(\tilde{e}_T^0)$ (cf. Table 5.12 and Definition 5.8).

3. If, for some $r \in \text{Reg}_{T'}$ and $x \in \text{Var}$, $\text{STM}(T', pc_{T'}^0) = [\text{load } r \text{ from } x] pc_{T'}^0$, then let $c @ \langle [T, pc_{T'}, z_{T'}, t_{T'}^0] | T \in \text{Thrd}_{c0}, z_{T'} \rangle$ be such that $c^n_{\text{prog}} \rightarrow c$ and choose $\tilde{c} @ \langle [T, pc_{T'}^c, \tilde{z}_{T'}, t_{T'}^{\tilde{a}}] | T \in \text{Thrd}_{c0}, \tilde{z}_{T'} \rangle$ such that $\tilde{c}^k_{\text{prog}} \rightarrow \tilde{c}$. Since $\forall i \in \{0, \ldots, n-1\} : T' \not\in \text{Thrd}_{exe}^i$, $T' \in \text{Thrd}_{exe}^{k+1}$, $\forall i \in \{0, \ldots, k-1\} : T' \not\in \text{Thrd}_{exe}^i$, $T' \in \text{Thrd}_{exe}^k$, $\frac{\tilde{x}}{ax}$ is a safe approximation of $\frac{\tilde{x}}{ax}$ (Lemma 5.50), $\tilde{z}_k$ contains a safe write history, $\text{Thrd}_{exe}^k \not\subseteq 1 \lor \{T \in \text{Thrd}_{exe}^k | \exists r \in \text{Reg}_{T} : \exists x \in \text{Var} :$ $\text{STM}(T, pc_{T}^{xk}) = [\text{load } r \text{ from } x] pc_{T}^{xk} \} = \emptyset$ (loading the value of $x$ is safe even if $x$ is a global variable since no other thread can affect its value in this case), TRIM is safe (Lemma 5.28), $\text{Thrd}_{ck} \subseteq \text{Thrd}$ and $\text{ACCTIME}$ is
safe (Lemma 5.55), it must be that:

\[
\begin{align*}
pc_{T'} &= pc_{T'}^c \\
\Gamma_{T'} &\in \gamma_{reg}(\bar{T}_{T'}) \\
t_a^{\bar{T}_{T'}} &\in \gamma_{lck}(\bar{T}_{T'}) \\
\exists x' \in \gamma_{var}(\bar{x}) : (\forall x \in \textbf{Var} : ((x x) \ T') \subseteq ((x' x) \ T')) \\
\forall lck \in \textbf{Lck} : ((\text{OWN}(\llbracket 0 lck \rrbracket) = T' \lor \text{OWN}(\llbracket lck \rrbracket) = T') \Rightarrow \\
(\text{STT}(\llbracket lck \rrbracket) = \text{STT}(\llbracket lck \rrbracket) \\
\text{OWN}(\llbracket lck \rrbracket) = \text{OWN}(\llbracket lck \rrbracket) \\
\text{DL}(\llbracket lck \rrbracket) \in \gamma_{lck}(\llbracket\text{DL}(\llbracket lck \rrbracket)\rrbracket) \\
\text{POWN}(\llbracket lck \rrbracket) = \text{POWN}(\llbracket lck \rrbracket) \\
\text{REL}(\llbracket lck \rrbracket) \in \gamma_{lck}(\text{REL}(\llbracket lck \rrbracket)) \\
\text{min}(\gamma_{lck}(\llbracket\text{DL}(\llbracket lck \rrbracket)\rrbracket)) = (-\infty)))
\end{align*}
\]

4. If, for some \(lck' \in \textbf{Lck}\), \(\text{STM}(T', pc_{T'}^0) = [\text{lock} \ lck']^{pc_{T'}^0}\), only the case that \(T'\) successfully and immediately acquires \(lck'\) needs to be considered. (Note that the remaining cases will be considered in the proofs of Lemmas 5.58 and 5.59.) Hence, \(\text{ACC TIME}\) is safe since it must be that \(\text{OWN}(\llbracket 0 lck' \rrbracket) = T'\) and \(\text{OWN}(\llbracket n' lck' \rrbracket) = T'\) (Lemma 5.55).

Since \(\text{OWN}(\llbracket 0 lck' \rrbracket) = T' \Rightarrow \text{OWN}(\llbracket 0 lck' \rrbracket) = T'\) and \(\text{OWN}(\llbracket 0 lck' \rrbracket) = T'\), there are three cases to consider.

(a) Assume that \(\text{OWN}(\llbracket 0 lck' \rrbracket) = T'\) (and thus, \(\text{OWN}(\llbracket 0 lck' \rrbracket) = T'\) and let \(c @ ([T, pc_{T}, \xi_{T}, t_a^{\bar{T}_{T}}]_{T \in \text{Thrd}} \cdot \bar{x}, \bar{l})\) be such that \(c_{a} \xrightarrow{\text{prg}} c\). Then choose \(\tilde{c} @ ([T, pc_{T}, \xi_{T}, t_a^{\bar{T}_{T}}]_{T \in \text{Thrd}_{\text{st}}} \cdot \tilde{x}, \tilde{l})\) such that \(\tilde{c}_{a} \xrightarrow{\text{prg}} \tilde{c}\). It is trivially the case that \(\text{OWN}(\llbracket 0 lck' \rrbracket) = \text{OWN}(\llbracket lck' \rrbracket) = T'\) and thus

\[
\begin{align*}
pc_{T'} &= pc_{T'}^c \\
\Gamma_{T'} &\in \gamma_{reg}(\bar{T}_{T'}) \\
t_a^{\bar{T}_{T'}} &\in \gamma_{lck}(\bar{T}_{T'}) \\
\exists x' \in \gamma_{var}(\bar{x}) : (\forall x \in \textbf{Var} : ((x x) \ T') \subseteq ((x' x) \ T')) \\
\forall lck \in \textbf{Lck} : ((\text{OWN}(\llbracket 0 lck \rrbracket) = T' \lor \text{OWN}(\llbracket lck \rrbracket) = T') \Rightarrow \\
(\text{STT}(\llbracket lck \rrbracket) = \text{STT}(\llbracket lck \rrbracket) \\
\text{OWN}(\llbracket lck \rrbracket) = \text{OWN}(\llbracket lck \rrbracket) \\
\text{DL}(\llbracket lck \rrbracket) \in \gamma_{lck}(\llbracket\text{DL}(\llbracket lck \rrbracket)\rrbracket) \\
\text{POWN}(\llbracket lck \rrbracket) = \text{POWN}(\llbracket lck \rrbracket) \\
\text{REL}(\llbracket lck \rrbracket) \in \gamma_{lck}(\text{REL}(\llbracket lck \rrbracket)) \\
\text{min}(\gamma_{lck}(\llbracket\text{DL}(\llbracket lck \rrbracket)\rrbracket)) = (-\infty)))
\end{align*}
\]
since \( \forall i \in \{0, \ldots, n-1\} : T' \not\in \text{Thrd}^i_{\text{exe}}, \ T' \in \text{Thrd}^n_{\text{exe}}, \forall i \in \{0, \ldots, k-1\} : T' \not\in \text{Thrd}^i_{\text{exe}}, \ T' \in \text{Thrd}^k_{\text{exe}} \), \( \xrightarrow{ax} \) is a safe approximation of \( \xrightarrow{ax} \) (Lemma 5.50) and TRIM is safe (Lemma 5.28).

(b) Assume that \( \text{OWN}(\tilde{I}^0 lck') = \text{OWN}(\tilde{I}^0 lck') = \bot_{\text{thrd}} \) and let \( c \) be such that \( c \xrightarrow{prg} c \land \text{OWN}(\tilde{I} lck') = T' \) and choose \( \tilde{c} \) such that \( \tilde{c} \xrightarrow{prg} c \). Then it must be that \( \text{OWN}(\tilde{I} lck') = T' \) (since \( T' \in \text{Thrd}^k_{\text{exe}} \) and thus \( \text{OWN}(\tilde{I}^{k'} lck') = T' \)) and

\[
\begin{align*}
&pc_{T'} = pc_{\tilde{T}'}, \\
&\tau_{T'} \in \gamma_{\text{reg}}(\tilde{\tau}_{T'}) \land \\
&\tau_{prg} \in \gamma_{\text{reg}}(\tilde{\tau}_{prg}) \land \\
&\exists x' \in \gamma_{\text{var}}(\tilde{x}) : (\forall x \in \text{Var} : ((x, x) T') \subseteq ((x', x) T')) \land \\
&\forall lck \in \text{Lck} : ((\text{OWN}(\tilde{I}^0 lck) = T' \lor \text{OWN}(\tilde{I} lck) = T') \Rightarrow \\
&(\text{STT}(\tilde{I} lck) = \text{STT}(\tilde{I} lck) \land \\
&\text{OWN}(\tilde{I} lck) = \text{OWN}(\tilde{I} lck) \land \\
&\text{DL}(\tilde{I} lck) \in \gamma_{\text{DL}}(\tilde{I} lck) \land \\
&\text{POWN}(\tilde{I} lck) = \text{POWN}(\tilde{I} lck) \land \\
&\text{REL}(\tilde{I} lck) \in \gamma_{\text{REL}}(\tilde{I} lck) \land \\
&\text{min}(\gamma_{\text{DL}}(\tilde{I} lck)) = -\infty))
\end{align*}
\]

since \( \forall i \in \{0, \ldots, n-1\} : T' \not\in \text{Thrd}^i_{\text{exe}}, \ T' \in \text{Thrd}^n_{\text{exe}}, \forall i \in \{0, \ldots, k-1\} : T' \not\in \text{Thrd}^i_{\text{exe}}, \ T' \in \text{Thrd}^k_{\text{exe}} \), \( \xrightarrow{ax} \) is a safe approximation of \( \xrightarrow{ax} \) (Lemma 5.50), TRIM is safe (Lemma 5.28), \text{Thrd}^k_{\text{exe}} \subseteq \text{Thrd}, \forall c' \in \text{Conf} : \forall lck \in \text{Lck} : \text{min}(\text{DLLOCK}(c', lck)) = -\infty \) (Algorithm 5.11), \( \text{DL}(\tilde{I} lck') = \tau_{prg} = \tau_{prg} + \text{TIME}(c', T') \) (Table 4.5) and \( \tau_{prg} \in \gamma_{\text{reg}}(\text{DLLOCK}(c, lck')) \) (Lemma 5.54).

(c) Assume that \( \text{OWN}(\tilde{I}^0 lck') = \bot_{\text{thrd}} \) and \( \text{OWN}(\tilde{I}^0 lck') = T' \) and let \( c \) be such that \( c \xrightarrow{prg} c \land \text{OWN}(\tilde{I} lck') = T' \) and choose \( \tilde{c} \) such that \( \tilde{c} \xrightarrow{prg} \tilde{c} \). Then it is easy to see that \( \text{OWN}(\tilde{I} lck') = \text{OWN}(\tilde{I} lck') = T' \).

First note that since \( \text{OWN}(\tilde{I}^n lck') = \text{OWN}(\tilde{I}^0 lck') = \bot_{\text{thrd}} \land \text{OWN}(\tilde{I}^{k'} lck') = \text{OWN}(\tilde{I}^0 lck') = T' \), it must be that \( \text{STT}(\tilde{I}^n lck') = \).
Chapter 5. Abstractly Interpreting PPL

Lemma 5.58 states that \( \sim_{\text{prog}} \) safely approximates the case that a thread, \( T \in \text{Thrd} \), issuing \text{lock} \( lck \) for some lock, \( lck \in \text{Lck} \), has to wait for an arbitrary number of owner switches on \( lck \) before it acquires \( lck \); i.e., the case that \( T \) is frozen for some period of abstract time before it is assigned \( lck \). Note that the lemma holds if all threads wanting to acquire some lock eventually will be able to do so (which obviously is the case if the concrete transition sequences are finite in length) and if either no thread issues a \text{load}-statement on a global variable or that the thread issuing the \text{load}-statement is the sole thread in \( \text{Thrd}_{\text{exe}} \) in any step of the transition sequence. Note that a variable is considered global if it could transfer data between two or more threads (cf. Algorithm 6.5, defined on page 169).
Lemma 5.58 (Soundness of $\tilde{\rightarrow}_{\text{prg}}$, possibly frozen thread):

If the valid concrete configurations (cf. Definition 4.4), abstract configurations, lock and thread

$$c^0 @ \langle[T, pc^0_T, z^0_T, t^0_T] \rangle_{\text{Thrd}} \in C_{\text{onf}},$$
$$c^{n_1} @ \langle[T, pc^{n_1}_T, z^{n_1}_T, t^{n_1}_T] \rangle_{\text{Thrd}} \in C_{\text{onf}},$$
$$c^{n_2} @ \langle[T, pc^{n_2}_T, z^{n_2}_T, t^{n_2}_T] \rangle_{\text{Thrd}} \in C_{\text{onf}},$$
$$\vdots$$
$$c^{n_m} @ \langle[T, pc^{n}_T, z^{n}_T, t^{n}_T] \rangle_{\text{Thrd}} \in C_{\text{onf}},$$
$$c^0 @ \langle[T, pc^0_T, z^0_T, t^0_T] \rangle_{\text{Thrd}} \in C_{\text{onf}},$$
$$c^{k} @ \langle[T, pc^k_T, z^k_T, t^k_T] \rangle_{\text{Thrd}} \in C_{\text{onf}},$$
$$lck' \in Lck, \text{ and}$$
$$T' \in \text{Thrd}_{c^k},$$

are such that

$$\text{STM}(T', pc^0_{T'}) = (\text{lock lck'}^{pc^0_{T'}},$$
$$0 \leq n_1 \leq n_2 \leq \ldots \leq n_m \leq n,$n
$$c^0 \xrightarrow{\text{prg}} \ldots \xrightarrow{\text{prg}} c^{n_1} \xrightarrow{\text{prg}} \ldots \xrightarrow{\text{prg}} c^{n_2} \xrightarrow{\text{prg}} \ldots \xrightarrow{\text{prg}} c^{n_m} \xrightarrow{\text{prg}} \ldots \xrightarrow{\text{prg}} c^n,$$

$$\tilde{c} \xrightarrow{\text{prg}} \ldots \xrightarrow{\text{prg}} \tilde{c}^k,$$

$$\text{Thrd}_{c^k} \subseteq \text{Thrd}_{c^0} \subseteq \text{Thrd},$$

$$pc^0_T = pc_{\tilde{c}^0_T},$$
$$z^0_T \in \gamma_{\text{reg}}(\tilde{z}^0_T),$$
$$t^0_T \in \gamma_{\text{reg}}(\tilde{t}^0_T),$$

$$\exists x' \in \gamma_{\text{var}}(\tilde{x}^0) : \forall x \in \text{Var} : \forall T \in \text{Thrd} : ((x^0 x) T) \subseteq ((x' x) T),$$
$$\forall lck \in Lck : ((\text{own}(\tilde{lck}) \neq \top_{\text{thrd}}) \Rightarrow (\text{stt}(\tilde{lck}) = \text{stt}(\tilde{lck}) \land \text{own}(\tilde{lck}) = \text{own}(\tilde{lck}) \land \text{dl}(\tilde{lck}) \in \gamma_{\text{dl}}(\tilde{lck}) \land \text{pown}(\tilde{lck}) = \text{pown}(\tilde{lck}) \land \text{rel}(\tilde{lck}) \in \gamma_{\text{rel}}(\text{rel}(\tilde{lck})) \land \text{min}(\gamma_{\text{rel}}(\text{rel}(\tilde{lck}))) = -\infty)))) \land \ldots$$
\[(\text{OWN}(\emptyset lck) = \bot_{\text{thrd}} \Rightarrow ((\text{OWN}(\emptyset lck) = \text{OWN}(\emptyset lck)) \lor (\text{OWN}(\emptyset lck) = T' \land s_{i T'}(\emptyset lck) = \text{unlocked}) \land \langle c^n, T' \rangle \ std(\emptyset lck) \in \gamma_t(\emptyset lck) \land min(\gamma_t(\emptyset lck)) = -\infty) \land \text{POWN}(\emptyset lck) = \text{POWN}(\emptyset lck) \land \text{REL}(\emptyset lck) \in \gamma_t(\text{REL}(\emptyset lck))))),\]

\[
\forall i \in \{0, \ldots, n - 1\} \setminus \{n_1, n_2, \ldots, n_m\} : T' \notin \text{Thread}^i_{\text{exe}},
\forall i \in \{n_1, n_2, \ldots, n_m\} : (T' \in \text{Thread}^i_{\text{exe}} \land \text{OWN}(\emptyset'' lck') \neq T'),
\forall i \in \{0, \ldots, k - 1\} : T' \notin \text{Thread}^i_{\text{exe}},
\text{T' \in Thread}^i_{\text{exe}}, \text{ and }
\forall i \in \{0, \ldots, k\} : (|\text{Thread}^i_{\text{exe}}| \neq 1 \lor \{T \in \text{Thread}^i_{\text{exe}} | \exists r \in \text{Reg}_T : \exists x \in \text{Var}_r :\text{STM(T, pc}^i_{\text{exe}}) = \text{[load r from x]}|pc}^i_{\text{exe}} \rangle = \emptyset),
\]

where for all \(i \in \{0, \ldots, n\}\), \(\text{Thread}^i_{\text{exe}}\) is as defined in Table 4.5, for all \(i \in \{0, \ldots, n\}\), \(\text{Thread}^i_{\text{exe}}\) is as defined in Table 5.13, and \(\text{Var}_r\) contains all \(x \in \text{Var}\) such that \(x\) can be written to by one thread and read from by another thread (i.e., there is a data dependency between the threads), then \(\rightarrow_{\text{cfg}}\) satisfies:

\[
\forall c @ ([T, pc_T, \text{Reg}_T, t^i_T]_{T \in \text{Thread}_k}, \emptyset) \in \text{Conf} : (c^{n} \rightarrow_{\text{cfg}} c \Rightarrow \exists \bar{c} @ ([T, pc^\bar{c}_T, \bar{t}^i_T] \in \text{Thread}_k, \emptyset) \in \text{Conf} : (\bar{c}^{n} \rightarrow_{\text{cfg}} \cdots \rightarrow_{\text{cfg}} \bar{c} \lor pc^i_T = pc^\bar{c}_T \lor \Gamma_T \in \gamma_{\text{reg}}(\bar{t}^i_T) \lor t^i_{T'} \in \gamma_{\text{reg}}(\bar{t}^i_{T'})) \lor \exists \bar{x}' \in \gamma_{\text{var}}(\emptyset) : (\forall x \in \text{Var} : ((\bar{x} x) T') \subseteq ((\bar{x}' x) T')) \land \forall lck \in \text{Lck} : (\text{OWN}(\emptyset lck) = T' \Rightarrow (\text{STT}(\emptyset lck) = s_{i T'}(\emptyset lck) \land \text{OWN}(\emptyset lck) = \text{OWN}(\emptyset lck) \land \text{DL}(\emptyset lck) = \gamma_t(\text{DL}(\emptyset lck)) \land \text{POWN}(\emptyset lck) = \text{POWN}(\emptyset lck) \land \text{REL}(\emptyset lck) \in \gamma_t(\text{REL}(\emptyset lck)) \land min(\gamma_t(\text{REL}(\emptyset lck))) = -\infty)))\]
EXPLANATION OF THE LEMMA. Thread $T'$ executes lock $lck'$ and might have to spin-wait in the concrete case before acquiring $lck'$. The possible spin-waiting is safely taken into account by $\overrightarrow{prg}$.

PROOF. Assume that the valid concrete configurations (cf. Definition 4.4), abstract configurations, lock and thread

\[
\begin{align*}
c^0 @ \langle [T, pc_T^0, z_T^0, t_T^0] \rangle & \in \text{Conf}, \\
c_o1 @ \langle [T, pc_T^1, z_T^1, t_T^1] \rangle & \in \text{Conf}, \\
c_o2 @ \langle [T, pc_T^2, z_T^2, t_T^2] \rangle & \in \text{Conf}, \\
& \vdots \\
c_{on} @ \langle [T, pc_T^{on}, z_T^{on}, t_T^{on}] \rangle & \in \text{Conf}, \\
c^0 @ \langle [T, pc_T^0, z_T^0, t_T^0] \rangle & \in \text{Conf}, \\
c^k @ \langle [T, pc_T^k, z_T^k, t_T^k] \rangle & \in \text{Conf}, \\
\end{align*}
\]

are as assumed in the lemma above.

First note that:

- Since $c^0 \overrightarrow{prg} c^n$, $\forall i \in \{0, \ldots, n-1\}, \langle n_1, n_2, \ldots, n_n \rangle : T' \notin \text{Thrd}_e^i$, $\forall i \in \{n_1, n_2, \ldots, n_m \}$ : ($T' \in \text{Thrd}_e^i \land \text{OWN}(\llbracket n \rrbracket lck') \neq T'$) and $\text{STM}(T', pc_T^0) = [\text{lock } lck'][pc_T^0]$, it must be that $pc_T^{n'} = pc_T^0$, $z_T^{n'} = z_T^0$, $t_T^{n'} = t_T^0 + \text{TIME}(c_{n'}^n, T') + \text{TIME}(c_{n^2}, T') + \ldots + \text{TIME}(c_{n^m}, T')$ and $\forall lck \in \text{Lck} : (\text{OWN}(\llbracket 0 \rrbracket lck) = T' \Rightarrow \llbracket n \rrbracket lck = \llbracket 0 \rrbracket lck)$ (the accumulated execution time for $T'$ is the only concrete thread-local state affected while $T'$ is spin-waiting; the states of the locks assigned to $T'$ are not affected either) (cf. Table 4.5).

- Since $\overrightarrow{prg} \ldots \overrightarrow{prg} c_k$, $\forall i \in \{0, \ldots, k-1\} : T' \notin \text{Thrd}_e^i$, it must be that $pc_T = pc_T^0$, $z_T = z_T^0$, $t_T = t_T^0$ and $\forall lck \in \text{Lck} : (\text{OWN}(\llbracket 0 \rrbracket lck) = T' \Rightarrow (\llbracket k \rrbracket lck = \llbracket 0 \rrbracket lck \land \text{min}(\gamma_i, \text{DL}(\llbracket k \rrbracket lck))) = -\infty)$ (no abstract thread-local state for $T'$ is affected while $T'$ is frozen; the states of the locks assigned to $T'$ are not affected either).
Since $pc_{T^0}^n = pc_{T^0}^0$, $T_{T^0}^n = T_{T^0}^0$, $\forall lck \in Lck : (\text{own}(lck) = T') \Rightarrow lck = lck = lck = lck = lck = lck = lck = lck = lck = \text{min}(\gamma_i(\text{DL}(lck))) = -\infty$, $pc_{T^0}^0 = pc_{T^0}^0$, $T_{T^0}^n \in \gamma_{\text{reg}}(T_{T^0}^0)$ and $\forall lck \in Lck : (\text{own}(lck) = T') \Rightarrow (\text{STT}(lck) = \text{STT}(lck) \land \text{own}(lck) = \text{DL}(lck) \land \gamma_i(\text{DL}(lck)) = \gamma_i(\text{DL}(lck)) \land \text{pown}(lck) = \text{pown}(lck) \land \text{rel}(lck) = \gamma_i(\text{rel}(lck)) \land \text{min}(\gamma_i(\text{DL}(lck))) = -\infty))$, it must be that (all thread-local states for $T'$, except the accumulated execution time, are safely approximated while $T'$ is spin-waiting in the concrete case and frozen in the abstract case; the states of the locks assigned to $T'$ are also safely approximated):

$p_{T^0}^n = pc_{T^0}^k$, $T_{T^0}^n \in \gamma_{\text{reg}}(T_{T^0}^0)$ and $\forall lck \in Lck : (\text{own}(lck) = T') \Rightarrow (\text{STT}(lck) = \text{STT}(lck) \land \text{own}(lck) = \text{DL}(lck) \land \gamma_i(\text{DL}(lck)) = \gamma_i(\text{DL}(lck)) \land \text{pown}(lck) = \text{pown}(lck) \land \text{rel}(lck) = \gamma_i(\text{rel}(lck)) \land \text{min}(\gamma_i(\text{DL}(lck))) = -\infty))$

- Since $\exists x' \in \gamma_{\text{var}}(x^0) : \forall x \in \text{var} : ((x^0 x) T') \subseteq ((x' x) T')$, $c_{\text{prog}}^0 \rightarrow \ldots \rightarrow c_{\text{prog}}^n \rightarrow \ldots \rightarrow c_{\text{prog}}^n \rightarrow \ldots \rightarrow c_{\text{prog}}^n \rightarrow \ldots \rightarrow c_{\text{prog}}^n$, $\forall i \in \{0, \ldots, n - 1\} \setminus \{n_1, n_2, \ldots, n_m\}$ : $T' \notin \text{Thrd}_{\text{exe}}^i$, $\forall i \in \{n_1, n_2, \ldots, n_m\}$ : $\text{own}(lck) = T'$, $\forall i \in \{0, \ldots, k - 1\}$ : $T' \notin \text{Thrd}_{\text{exe}}^i$, $\text{stm}(T', pc_{T^0}^0) = [\text{lock lck}]^{pc_{T^0}^0}$ and TRIM is safe (Lemma 5.28), it must be that $\exists x' \in \gamma_{\text{var}}(x^0) : \forall x \in \text{var} : ((x^0 x) T') \subseteq ((x' x) T')$ (the write histories for $T'$ on the defined program variables are safely approximated while $T'$ is spin-waiting in the concrete case and frozen in the abstract case).

- Since $\forall i \in \{0, \ldots, k\} : (|\text{Thrd}_{\text{exe}}^i| \neq 1 \lor \{T \in \text{Thrd}_{\text{exe}}^i \mid \exists r \in \text{Reg}_T : \exists x \in \text{var}_r : \text{stm}(T, pc_{T^0}^i) = [\text{load r from x}]^{pc_{T^0}^i} = \emptyset\}$, it must be that $\forall i \in \{0, \ldots, k\} : (\{T \in \text{Thrd}_{\text{exe}}^i \mid \exists r \in \text{Reg}_T : \exists x \in \text{var}_r : \text{stm}(T, pc_{T^0}^i) = [\text{load r from x}]^{pc_{T^0}^i} = \emptyset \Rightarrow |\text{Thrd}_{\text{exe}}^i| = 1\}$). This means that if some thread in $\text{Thrd}_{\text{exe}}$, where $i \in \{0, \ldots, k\}$, performs a $\text{load}$-statement, there is only one single thread in $\text{Thrd}_{\text{exe}}$; thus that thread performs the
load-statement. It is then easy to see, from the definition of $\text{Thrd}_{ex}^{lck}$, that there cannot occur any other write than those represented by $\bar{x}^l$ such that it could affect the load-statement of the thread in $\text{Thrd}_{ex}^{lck}$ (cf. Assumption 5.51) – thus, it must be that $\bar{x}^k$ (and also all $\bar{x}^i$, where $i \in \{0, \ldots, k\}$) contains a safe write history (cf. Definition 5.19).

- Since, trivially, $\forall lck \in Lck : \{T \in \text{Thrd}_{ex}^{lck} \cap \text{Thrd}_{ex}^{lck} | \text{STM}(T, pc_T^l) = [\text{lck} lck]^{pc_T^l} \} \subseteq \{T \in \text{Thrd}_{ex}^{lck} | \exists l \in \text{Lbl}_T : \text{STM}(T, l) = [\text{lck} lck]^{l} \}$ (the set of threads executing $\text{lck} lck$ is trivially a subset of the set containing all threads that could execute $\text{lck} lck$ somewhere in the program), it must be that if $T'$ can be assigned a lock in the concrete case, it can also be assigned the lock in the corresponding abstract case.

- Since $\mapsto_{prg}$ over-approximates the lock-owner assignment possible for $\mapsto_{prg}$ and $T' \in \text{Thrd}_{ex}^{lck}$, it must be that $\text{OWN}(\emptyset lck') = T'$ is possible even if $\text{OWN}(\emptyset lck') = \bot_{thrd}$, given that some other thread (i.e., not $T'$) does $\text{lck} lck'$ before $T'$ in the abstract case and that the abstract transition sequence safely represents the concrete transition sequence; the situation is possible since $\text{Time} = \text{Intv}$ (cf. Lemma 5.56). However, if $\text{STM}(T', pc_{T'}^0) = [\text{lck} lck']^{pc_{T'}^0}$, $\text{OWN}(\emptyset lck') = \bot_{thrd}$ and $\text{OWN}(\emptyset lck') = T'$, it must be that $\text{STM}(\emptyset lck') = \text{unlock}$ and $\text{OWN}(\emptyset lck') = T'$, where $\text{stm}^T + \text{time}(c^T, T') \in \gamma_T(\text{DL}(\emptyset lck'))$ and $\min(\gamma_T(\text{DL}(\emptyset lck'))) = -\infty$ (Table 5.13, Algorithm 5.11 and Lemmas 5.54 and 5.56).

- Since $\text{STM}(T', pc_{T'}^k) = [\text{lck} lck']^{pc_{T'}^k}$ and $T' \in \text{Thrd}_{ex}^{lck}$, it must be that $\text{OWN}(\emptyset lck') = T'$ (in the transition(s) from $\bar{x}^k$, $T'$ is no longer frozen and thus it must be the new owner of $lck'$).

The proof will be conducted using induction based on $T'$ having to wait for $j$, where $j \geq 0$, threads to first (acquire and) release $lck'$ before it can successfully acquire $lck'$.

First consider the base case. Therefore, assume that $T'$ is the first thread in a set of competing threads to successfully acquire $lck'$; i.e., $j = 0$. Then it must be that $\{n_1, n_2, \ldots, n_m\} = \emptyset$, and thus, $\forall i \in \{0, \ldots, n-1\} : T' \not\in \text{Thrd}_{ex}^{lck}$. (Note that $c^0$ can be chosen to be the first configuration satisfying $\text{OWN}(\emptyset lck') = \bot_{thrd} \land \text{STM}(T', pc_{T'}^0) = [\text{lck} lck']^{pc_{T'}^0}$ and the rest of the assumptions of the lemma.) It must also be that the case $\text{OWN}(\emptyset lck') = \bot_{thrd}$, $\text{OWN}(\emptyset lck') \in \{\bot_{thrd}, T'\}$ and $\text{OWN}(\emptyset lck') = \text{OWN}(\emptyset lck') = T'$ must be considered.
Chapter 5. Abstractly Interpreting PPL

Since \( T' \) is the first thread to acquire \( lck' \) it must be that \( [lck]_0 = [lck']_0 \) and \( [lck]_k = [lck']_k \). Thus, since \( \text{OWN}([lck]_0) = \bot_{\text{thrd}} \) and \( \text{STM}(T', [p_{c_{T'}}]) = [\text{lock} lck']_{\text{pc}_{T'}} \), it must be that \( \text{OWN}([lck]_0) = \bot_{\text{thrd}} \Rightarrow (\text{STM}(T', [p_{c_{T'}}]) \Rightarrow [\text{lock} lck']_{\text{pc}_{T'}} \Rightarrow ([lck]_0 = [lck']_0 \land [lck]_k = [lck']_k)) \). But, then all the assumptions of Lemma 5.57 are fulfilled, and thus, it must be that:

\[
\forall c \in \{T, p_{c_{T}}, \tau_{T}, \tau_{T}^a\} \in \text{Conf} : \\
(\epsilon_{\text{proj}} \rightarrow c) \Rightarrow \exists c \in \{T, p_{c_{T}}, \tau_{T}, \tau_{T}^a\} \in \text{Conf} : \\
(\epsilon_{\text{proj}}^c \rightarrow \epsilon \land \\
p_{c_{T'}} = p_{c_{T}} \land \\
\tau_{T'} \in \gamma_{\text{reg}}(\tau_{T}) \land \\
t_{T}^a \in \gamma(\tau_{T}) \land \\
\exists x' \in \gamma_{\text{var}}(x) : (\forall x \in \text{Var} : ((x, x) \ T') \subseteq ((x', x) \ T')) \land \\
\forall lck \in \text{Lck} : (\text{OWN}(lck) = T' \Rightarrow \\
(\text{STT}(lck) = \text{STT}(lck) \land \\
\text{OWN}(lck) = \text{OWN}(lck) \land \\
\text{DL}(lck) \in \gamma(\text{DL}(lck)) \land \\
\text{POWN}(lck) = \text{POWN}(lck) \land \\
\text{REL}(lck) \in \gamma(\text{REL}(lck)) \land \\
\text{min}(\gamma(\text{DL}(lck))) = -\infty)))
\]

This concludes the proof of the base case.

Now consider the case that \( T' \) must wait for \( j \) owner switches (i.e., \( \text{lock}:s \) and \( \text{unlock}:s \)) on \( lck' \) before it can acquire \( lck' \) itself; i.e., \( T' \) is owner number \( j + 1 \) among a set of competing threads to successfully acquire \( lck' \) (note that a thread could successfully acquire and release \( lck' \) several times while \( T' \) is waiting to acquire \( lck' \); each time then counts as an owner switch). The induction assumption is that the lemma holds for all \( j \) owners that acquire \( lck' \) while \( T' \) is waiting (i.e., frozen in the abstract case) and for all cases involving other locks.

Assume that \( T' \) must wait for \( j \) owner switches on \( lck' \) before it successfully acquires \( lck' \) itself and that the lemma holds for all \( j \) owners that acquire \( lck' \) while \( T' \) is waiting. Then it must be that \( \{n_1, n_2, \ldots, n_m\} \neq \emptyset \), and thus \( t_{T'}^{n_1} \leq t_{T'}^{n_2} = t_{T'}^{n_1} + \text{TIME}(e_{n_1}, T') + \text{TIME}(e_{n_2}, T') + \ldots + \text{TIME}(e_{n_m}, T') = t_{T'}^{n_m} + \text{TIME}(e_{n_m}, T') \) (cf. Assumption 4.1 and Table 4.5).

Since the lemma holds for all \( j \) owners that acquire \( lck' \) while \( T' \) is waiting, and all other cases involving other locks, and \( \rightarrow_{\text{proj}} \) safely over-approximates the transitions described by \( \rightarrow_{\text{proj}} \) for all other cases (Lemma 5.57), including lock
owner assignments (Lemma 5.56), it must be that there exists an abstract transition trace (starting at $c^0$ and ending at $c^k$) that safely represents the concrete trace from $c^0$ to $c^n$ for all $j$ owners of $lck'$, at least until the point in which they release $lck'$ and do not acquire it again (which is the important part of the trace to consider here), the order in which threads acquire $lck'$ and all states, including the accumulated execution times (cf. Lemmas 5.52 and 5.57 and the induction assumption). Thus, since $T' \in \text{Thrd}_e^{\min} \land \text{OWN}([\beta_\text{m} lck') \neq T'$, $\forall i \in \{n_m + 1, \ldots, n - 1\}$: $T' \not\in \text{Thrd}_e^{i}$ and $T' \in \text{Thrd}_e^{a} \land \text{OWN}([\beta_\text{a} lck') = \bot_{\text{thr}} \land \text{OWN}([\beta_\text{m} lck') = T'$ (since it is assumed that $T'$ acquires $lck'$ in the transition from $c^n$), it must be that $lck'$ is released (by owner number $j$) in a transition to $c^{n'}$, where $c^{n_1} \rightarrow \cdots \rightarrow c^{n'} \rightarrow \cdots \rightarrow c^n$ and $n_m < n' \leq n$. Thus, it must be that $t^{a_m}_{T'} + \text{TIME}(c^{n_1}, T') = t^{a_n}_{T'} \leq \text{REL}(\beta_\text{a} lck') \leq t^{a_m}_{T'} + \text{TIME}(c^n, T')$ (cf. Assumption 4.1 and Table 4.5), where $\text{REL}(\beta_\text{a} lck') = \text{REL}(\beta_\text{a} lck')$ and $\text{REL}(\beta_\text{m} lck') \in \gamma(T_{\text{REL}}(\beta_\text{m} lck'))$ (given the abstract trace from $c^0$ to $c^k$ that safely represents the trace from $c^0$ to $c^{n'}$ for the previous, i.e., $j^{th}$, owner of $lck'$, it is easy to see that this is the result when the $j^{th}$ owner issues $\text{unlock} lck'$).

But, then it is trivially the case that $t^{a_m}_{T'} + \text{TIME}(c^n, T') \in \gamma_\prime(\text{DLOCK}(\beta_\text{k} lck'))$ (Lemma 5.54).

To show that $\text{ACCTIME}$, as defined in Algorithm 5.12 on page 116, is safe for this case, first note that $T' \in \text{Thrd}_e^{\text{k}}$, $\text{STM}(T', pc_k) = [\text{lock} lck') \text{pc}_k^{\text{k}}$ and $\text{STT}(\beta_\text{k} lck') = \text{unlocked}$. Also note that since $t^{a_m}_{T'} + \text{TIME}(c^n, T') \in \gamma_\prime(\text{DLOCK}(\beta_\text{k} lck')) = \text{DL}(\beta_\text{k} lck')$, $\text{DL}(\beta_\text{k} lck') = t^{a_m}_{T'} + \text{TIME}(c^n, T')$ (cf. Tables 4.2 and 4.5 since $T'$ acquires $lck'$ in a transition from $c^n$ and $\text{MIN}(\gamma(T_{\text{DL}}(\beta_\text{k} lck')) \leq t^{a_m}_{T'} + \text{TIME}(c^n, T') \leq \text{MAX}(\gamma(T_{\text{DL}}(\beta_\text{k} lck'))$) (cf. Assumptions 4.1 and 5.51), it must be the case that $\text{DL}(\beta_\text{k} lck') \preceq_{\text{ST}} t^{a_m}_{T'} + \text{TIME}(c^n, T')$, which means that there are three branches of Algorithm 5.12 that must be considered here. Note that this also means that $\text{DL}(\beta_\text{k} lck') \in \gamma(T_{\text{DL}}(\beta_\text{k} lck'))$. For the sake of readability, let $\bar{c}^{\text{k}} = (T, pc_k^{\text{k}}, t^{a_k}_{T'}, \bar{z}_k, \bar{\alpha}_k^{\text{k}})$. Also let $i^{\alpha}_{T'}$ be defined as $i^{\alpha}_{T'}$ in Algorithm 5.12.

1. Since $T'$ has been frozen while waiting to acquire $lck'$, it can be the case that $t^{a_m}_{T'} + \text{TIME}(\bar{c}^{\text{k}}, T') \preceq_{\text{ST}} \text{REL}(\beta_\text{k} lck')$, where $i^{\alpha}_{T'} = i^{\alpha}_{T'}$. (Note that this does not necessarily have to be the case, though.) Let $\bar{c}'$ be any configuration derived before (i.e., $\bar{c}' = \bar{c}^{\text{k}}$) or inside the while-loop in Algorithm 5.12.
First note that it cannot be that \( \text{ABSTIME}(\tilde{c}', T') = [0, 0] \) and \( \tilde{t}_V^w \vdash_t \text{ABSTIME}(\tilde{c}', T') \) \( \preceq_t \text{REL}(\tilde{\ell}^{kn} lck') \) (cf. Assumptions 4.3 and 5.51). This means that the while-loop will eventually terminate. It does so when \( \tilde{t}_V^w \) is the last possible interval in time that safely represents the situation that \( T' \) has not yet acquired \( lck' \); thus, at \( \tilde{t}_V^w \vdash_t \text{ABSTIME}(\tilde{c}', T') \), \( T' \) might have acquired \( lck' \) (i.e., \( \tilde{t}_V^w \preceq_t \text{REL}(\tilde{\ell}^{kn} lck') \) and \( \tilde{t}_V^w \vdash_t \text{ABSTIME}(\tilde{c}', T') \) \( \not\preceq_t \text{REL}(\tilde{\ell}^{kn} lck') \)). In later references within this proof, the \( \tilde{t}_V^w \) obtained at the exit of the while-loop will be referred to as \( \tilde{t}_V^w \).

Since \( \tilde{t}_V^w \preceq_t \text{REL}(\tilde{\ell}^{kn} lck') \) and \( \tilde{t}_V^w \vdash_t \text{ABSTIME}(\tilde{c}', T') \) \( \not\preceq_t \text{REL}(\tilde{\ell}^{kn} lck') \), it is easy to see that this branch will lead to an auxiliary configuration, \( \tilde{c}^{kl} \), such that \( \tilde{c}^k \xrightarrow{\text{reg}} \tilde{c}^{kl} \), for which \( \tilde{\ell}^{kl} lck' = \tilde{\ell}^{kn} lck' \); i.e., \( T' \) has not yet acquired \( lck' \). The only difference for \( T' \) between \( \tilde{c}^k \) and \( \tilde{c}^{kl} \) is that, in the latter, it has an advanced abstract accumulated execution time (cf. Table 5.12). Since \( \tilde{t}_V^w \vdash_t \text{ABSTIME}(\tilde{c}', T') \) \( \not\preceq_t \text{REL}(\tilde{\ell}^{kn} lck') \), it is also easy to see that this branch of Algorithm 5.12 will not be taken when ACCTIME is called based on \( \tilde{c}^{kl} \). Note that it must be that \( |\{n_1, n_2, \ldots, n_m\}| \) is greater than or equal to the number of iterations of the while-loop (Assumption 5.51 and Lemma 5.53). Thus, it is also easy to see that \( \text{DL}(\tilde{\ell}^{kn} lck') \not\preceq_t \tilde{t}_V^w \vdash_t \text{ABSTIME}(\tilde{c}^{kl}, T') \) since \( \text{TIME}(\tilde{c}', T') \in \gamma_i(\text{ABSTIME}(\tilde{c}^{kl}, T')) \), where \( i \in \{n_1, n_2, \ldots, n_m\} \) is the corresponding concrete configuration for which the while-loop terminates (Assumption 5.51). This means that for \( \tilde{c}^{kl} \), one of the two last branches (considered in the next two bullets) of Algorithm 5.12 will apply.

2. First note that it must be that \( \text{PO\WN}(\tilde{\ell}^{kn} lck') \neq T' \) since \( T' \) has been waiting for at least one other thread to release \( lck' \) before it is allowed to acquire it (cf. Tables 5.12 and 5.13 and the induction assumption). If, on the other hand, \( \text{REL}(\tilde{\ell}^{kn} lck') \not\preceq_t \tilde{t}_V^w \vdash_t \text{ABSTIME}(\tilde{c}^{kl'}, T') \), then the proof is equivalent to the corresponding part of the proof for Lemma 5.55 since \( \tilde{t}_V^w + \text{TIME}(\tilde{c}^{kl'}, T') = \text{DL}(\tilde{\ell}^{kn} lck') \), \( \text{DL}(\tilde{\ell}^{kn} lck') \in \gamma_i(\text{DL}(\tilde{\ell}^{kn} lck')) \) and \( \text{REL}(\tilde{\ell}^{kn} lck') \in \gamma_i(\text{REL}(\tilde{\ell}^{kn} lck')) \). Note that this also applies if \( \tilde{t}_V^w = \tilde{t}_V^w \) (i.e., if \( \tilde{c}^{kn} = \tilde{c}^{kl} \)) since it must be that \( \tilde{t}_V^w \in \gamma_i(\tilde{t}_V^w) \), which follows from Assumption 5.51 and Lemma 5.52 based on 1 above.

3. If \( (\tilde{t}_V^w \vdash_t \text{ABSTIME}(\tilde{c}', T')) \cap_t \text{REL}(\tilde{\ell}^{kn} lck') \neq \emptyset \), then let \( \tilde{t}_V^w = \tilde{t}_V^w \vdash_t \text{ABSTIME}(\tilde{c}', T) \) (where \( \tilde{t}_V^w \) is thus defined as in Algorithm 5.12 and \( \tilde{t}_V^w \) is either \( \tilde{t}_V^w \) or \( \tilde{t}_V^w \) and \( \tilde{c}' \) is either \( \tilde{c}^{kn} \) or \( \tilde{c}^{kl} \)), which is obviously a safe
estimation of the first point in time when T' can acquire lex'.

Now, let c' be any configuration derived before (i.e., c' = ek' or c' = ek') or inside the repeat-loop (and the corresponding for i''), which will now be considered. Note that i'' = i is used to exit the loop in case DL(kek' lck') ≺i i'' ⊕i ABSTIME(c', T') or 0 ∈ γi(ABSTIME(c', T')) where the latter case means that a i'' such that REL(kek' lck') ≺i i'' cannot be derived (cf. Assumption 5.51).

(a) If DL(kek' lck') ≺i i'' ⊕i ABSTIME(c', T'), then it must be that i'' is a safe estimation of the last point in time when T' can acquire lex' since t'' + TIME(c'', T') = DL(kek'' lex') ∈ γi(DL(kek' lck')) and T' acquires lex' in a transition from c'' (cf. Assumption 5.51 and Lemma 5.53 which means that the total number of iterations of the repeat-loop, and possibly the while-loop from 1, must be greater than or equal to |{n1, n2,…, nm, n}|). Thus, it must be that t'' + TIME(c'', T') ∈ γi((i'' ⊕i i'') ⊗i DL(kek'' lex') ⊗i (REL(kek'' lex')) ⊗i [∞, ∞]) since REL(kek'' lex') ∈ γi(REL(kek'' lex')) and min(γi(DL(kek'' lex'))) = −∞.

(b) If 0 ∈ γi(ABSTIME(c', T')), then it must obviously be that i ⊕i ABSTIME(c'', T'), where i = (i'' ⊕i i'') ⊗i REL(kek'' lex'), and c'' = (T, p, ckek'', i'', (T = T' ? i : i''') | T ∈ Thre, z) is a safe approximation of the last point in time when T' can (or rather, will) acquire lex' (cf. Assumption 5.51 and Lemma 5.53). Thus, it must be that t'' + TIME(c'', T') ∈ γi((i'' ⊕i i'') ⊗i DL(kek'' lex') ⊗i (REL(kek'' lex')) ⊗i [∞, ∞]) since REL(kek'' lex') ∈ γi(REL(kek'' lex')), DL(kek'' lex') ∈ γi(DL(kek'' lex')), t'' + TIME(c'', T') = DL(kek'' lex'), min(γi(DL(kek'' lex'))) = −∞ and T' acquires lex' in a transition from c''.

(c) If 0 ∉ γi(ABSTIME(c', T')) and also DL(kek' lck') ≺i i'' ⊕i ABSTIME(c', T'), then it must be that, at the end of some iteration of the repeat-loop, REL(kek' lck') ≺i i''. For such a i'', it is easy to see that t'' + TIME(c'', T') ∈ γi((i'' ⊕i i'') ⊗i DL(kek' lck') ⊗i (REL(kek' lck')) ⊗i [∞, ∞]) since REL(kek' lck') ∈ γi(REL(kek' lck')), DL(kek' lck') ∈ γi(DL(kek' lck')), t'' + TIME(c'', T') = DL(kek' lck'), min(γi(DL(kek' lck'))) = −∞ and T' acquires lex' in a transition from c'' (cf. Assumption 5.51 and Lemma 5.53 which means
that the total number of iterations of the repeat-loop, and possibly the while-loop from 1, must be greater than or equal to \(|\{n_1, n_2, \ldots, n_m, n\}|\).

Thus, it has been shown that \(t_{TV}^n + \text{TIME}(c^n, T') \in \gamma_l(\text{ACC}\text{TIME}(\bar{c}' \circ \langle [T, pc_T', \tilde{t}_T', T_T'], \text{Thrd}_{\text{ex}}, \tilde{x}', \tilde{t}' \rangle, \text{Thrd}_e^{\text{ex}}, T'))\), for both the case that \(\bar{c}'\) is \(\check{c}^{k''}\) (if \(\tilde{t}_T^d \vdash \text{ABSTIME}(\check{c}^{k''}, T') \preceq \text{R\text{EL}}(\check{c}^{k''} \text{ lck}')\)) and \(\bar{c}'\) is \(\check{c}^{k'}\) (if \(\tilde{t}_T^d \vdash \text{ABSTIME}(\check{c}^{k''}, T') \preceq \text{R\text{EL}}(\check{c}^{k''} \text{ lck}')\)), where \(\check{c}^{k''}\) and \(\check{c}^{k'}\) are as defined above. If \(\tilde{t}_T^d \vdash \text{ABSTIME}(\check{c}^{k''}, T') \preceq \text{R\text{EL}}(\check{c}^{k''} \text{ lck}')\), it is easy to see that

\[
\begin{align*}
& pc_T^n = pc_T^{\check{c}^{k''}} , \\
& \forall lck \in \text{Lck} : (\text{OWN}(\check{c}^{k''} \text{ lck}) = T') \Rightarrow (\text{STT}(\check{c}^{k''} \text{ lck}) = \text{STT}(\check{c}^{k''} \text{ lck}) ) \land \\
& \text{OWN}(\check{c}^{k''} \text{ lck}) = \text{OWN}(\check{c}^{k''} \text{ lck}) \land \\
& \text{DL}(\check{c}^{k''} \text{ lck}) = \gamma_l(\text{DL}(\check{c}^{k''} \text{ lck}) ) \land \\
& \text{POWN}(\check{c}^{k''} \text{ lck}) = \gamma_l(\text{POWN}(\check{c}^{k''} \text{ lck}) ) \land \\
& \text{REL}(\check{c}^{k''} \text{ lck}) = \gamma_l(\text{REL}(\check{c}^{k''} \text{ lck}) ) \land \\
& \text{min}(\gamma_l(\text{DL}(\check{c}^{k''} \text{ lck})) = -\infty) \\
\end{align*}
\]

since, for \(T'\), the accumulated abstract execution time is the only state affected by the transition \(\check{c}^{k'} \xrightarrow{\text{prog}} \check{c}^{k''}\) (cf. Table 5.12; this means that, for example, \(\check{c}^{k''} \text{ lck}' = \check{c}^{k''} \text{ lck}'\)) and \(\text{TRIM}\) is safe (Lemma 5.28). Thus, for both transition sequences described by \(\check{c}^{k'} \xrightarrow{\text{prog}} \check{c}^{k''} \xrightarrow{\text{prog}} \bar{c}\) and \(\check{c}^{k'} \xrightarrow{\text{prog}} \bar{c}\), where \(\bar{c}@\langle [T, pc_T, \tilde{t}_T, T_T], \text{Thrd}_{\text{ex}}, \tilde{x}, \tilde{t} \rangle \in \text{Conf}\), for the two different cases \(\tilde{t}_T^d \vdash \text{ABSTIME}(\check{c}^{k''}, T') \preceq \text{R\text{EL}}(\check{c}^{k''} \text{ lck}')\) and \(\tilde{t}_T^d \vdash \text{ABSTIME}(\check{c}^{k''}, T') \not\preceq \text{R\text{EL}}(\check{c}^{k''} \text{ lck}')\), respectively, it must be that

\[
\begin{align*}
& pc_T^n = pc_T^{\check{c}'}, \\
& \forall T_T \in \gamma_l(\tilde{t}_T'), \\
& \forall lck \in \text{Lck} : (\text{OWN}(\check{c}^{k''} \text{ lck}) = T') \Rightarrow (\text{STT}(\check{c}^{k''} \text{ lck}) = \text{STT}(\check{c}^{k''} \text{ lck}) ) \land \\
& \text{OWN}(\check{c}^{k''} \text{ lck}) = \text{OWN}(\check{c}^{k''} \text{ lck}) \land \\
& \text{DL}(\check{c}^{k''} \text{ lck}) = \gamma_l(\text{DL}(\check{c}^{k''} \text{ lck}) ) \land \\
& \text{POWN}(\check{c}^{k''} \text{ lck}) = \gamma_l(\text{POWN}(\check{c}^{k''} \text{ lck}) ) \land \\
& \text{REL}(\check{c}^{k''} \text{ lck}) = \gamma_l(\text{REL}(\check{c}^{k''} \text{ lck}) ) \land \\
& \text{min}(\gamma_l(\text{DL}(\check{c}^{k''} \text{ lck})) = -\infty) \\
\end{align*}
\]
where $c^n \xrightarrow{prg} c$ for some $c \in \{ [T, pc_T^0, z_T^0, t_T^0] \}_{T \in \text{Thrd}_e} \in \text{Conf}$, since $\xrightarrow{ax}$ is a safe approximation of $\xrightarrow{ax}$ (Lemma 5.50), $DL(\text{lock}^i) = t_L^n + \text{TIME}(c^n, T')$ (Table 4.2) and TRIM is safe (Lemma 5.28). But then the lemma holds. □

Lemma 5.59 states that $\xrightarrow{prg}$ can be used to safely approximate any finite concrete transition sequence. (It should be obvious that in all finite concrete transition sequences, it must be that any thread wanting to acquire some lock is eventually able to do so since the transition sequence would otherwise be infinite.) Note that the approximation is safe if either no thread issues a load-statement on a global variable or that the thread issuing the load-statement is the sole thread in $\text{Thrd}_{exe}$ in any step of the transition sequence. Note that a variable is considered global if it could transfer data between two or more threads (cf. Algorithm 6.5, defined on page 169).

**Lemma 5.59 (Soundness of $\xrightarrow{prg}$, final state):**

*If the valid concrete configurations (cf. Definition 4.4) $c^0 @ \langle [T, pc_T^0, z_T^0, t_T^0] \rangle \in \text{Conf}$ and $c^n @ \langle [T, pc_T^n, z_T^n, t_T^n] \rangle \in \text{Conf}$ and the abstract configuration $\hat{c}^0 @ \langle [T, pc_T, z_T, t_T] \rangle \in \hat{\text{Conf}}$, are such that $c^0 \in \gamma_{conf}(\hat{c}^0), \forall \text{lock} \in \text{Lck} : \min(\gamma_l(DL(\text{lock}^i))) = -\infty, 0 < n$ and $c^0 \xrightarrow{prg} \ldots \xrightarrow{prg} c^n$, then $\xrightarrow{prg}$ satisfies*

$$(\forall T \in \text{Thrd}_e : \text{STM}(T, pc_T^n) = [\text{halt}]^{pc_T^n})$$

$$\Rightarrow$$

$$\left( \exists \hat{z}^k @ \langle [T, pc_T^k, z_T^k, t_T^k] \rangle \in \hat{\text{Conf}} : \right.$$}

$$
(c^0 \xrightarrow{prg} \ldots \xrightarrow{prg} \hat{c}^k \wedge \\
(\forall i \in \{0, \ldots, k-1\} : [\text{Thrd}_{exe}^i] \neq 1 \lor \\
\{ T \in \text{Thrd}_{exe}^i | \exists r \in \text{Reg}_T : \exists x \in \text{Var}_g : \\
\text{STM}(T, pc_T^i) = [\text{load } r \text{ from } x]^{pc_T^i} = 0 \} )$$

$$\Rightarrow$$
\[(\forall T \in \text{Thrd}_t : (pc_T^n = pc_T^k) \land
\quad t_T^n \in \gamma_{reg}(\tilde{\mathbb{f}}^k) \land
\quad t_T^n \in \gamma(f_T^\iota) \land
\quad \exists x' \in \gamma_{var}(\tilde{\mathbb{f}}^k) : (\forall x \in \text{Var} : ((x^n, x) T) \subseteq ((x', x) T)) \land
\forall \text{lk} \in \text{Lck} : (\text{STT}(\mathbb{f}^n \text{ lck}) = \text{STT}(\tilde{\mathbb{f}}^k \text{ lck}) \land
\quad \text{OWN}(\mathbb{f}^n \text{ lck}) = \text{OWN}(\tilde{\mathbb{f}}^k \text{ lck}) \land
\quad \text{DL}(\mathbb{f}^n \text{ lck}) \in \gamma_i(\text{DL}(\tilde{\mathbb{f}}^k \text{ lck})) \land
\quad \text{POWN}(\mathbb{f}^n \text{ lck}) = \text{POWN}(\tilde{\mathbb{f}}^k \text{ lck}) \land
\quad \text{REL}(\mathbb{f}^n \text{ lck}) \in \gamma_i(\text{REL}(\tilde{\mathbb{f}}^k \text{ lck})) \land
\quad \min(\gamma_i(\text{DL}(\tilde{\mathbb{f}}^k \text{ lck}))) = -\infty))\)

where, for all \(i \in \{0, \ldots, k-1\}\), \(\text{Thrd}_t^i\) is as defined in Table 5.13, and \(\text{Var}_g\) contains all \(x \in \text{Var}\) such that \(x\) can be written to by one thread and read from by another thread (i.e., there is a data dependency between the threads). □

**EXPLANATION OF THE LEMMA.** There is an abstract transition sequence, derived using \(\overset{\sim}{\rightarrow}_{pr_g}\), for each terminating concrete transition sequence, derived using \(\overset{\sim}{\rightarrow}_{pr_g}\), that safely approximates that concrete transition sequence, as long as no unsafe loading of a global variable value occurs on the abstract transition sequence.

**PROOF.** Assume that the valid (cf. Definition 4.4) concrete configurations \(c^0 @ ([T, pc_T^0, z_T^0, t_T^0]_{T \in \text{Thrd}_t}, \mathbb{x}^0, \mathbb{g}^0) \in \text{Conf}\) and \(c^n @ ([T, pc_T^n, z_T^n, t_T^n]_{T \in \text{Thrd}_t}, \mathbb{x}^n, \mathbb{g}^n) \in \text{Conf}\) and the abstract configuration \(\overset{\sim}{c^0} @ ([T, pc_T^0, z_T^0, t_T^0]_{T \in \text{Thrd}_t}, \mathbb{x}^0, \mathbb{g}^0) \in \text{Conf}\) are as assumed in the lemma above.

Note that since \(\forall T \in \text{Thrd}_t : \text{STM}(T, pc_T^0) = \{\text{halt}\}^{pc_T^0}\), it must be that all threads trying to acquire a lock at some point will eventually successfully do so (i.e., there are no deadlocks etc.) and there are no infinite loops. Also note that \(\overset{\sim}{\rightarrow}_{pr_g}\) covers all the possible concrete situations for lock owner assignments, regardless of which thread issues \(\text{lock lck}\) first in the abstract case (Lemma 5.56).

This proof will partly be conducted using induction on how the states of a configuration are changed during transitions, based on one thread at a time. Therefore, consider \(c^f @ ([T, pc_T^f, z_T^f, t_T^f]_{T \in \text{Thrd}_t}, \mathbb{x}^f, \mathbb{g}^f) \in \text{Conf}\), \(c^g @ ([T, pc_T^g, z_T^g, t_T^g]_{T \in \text{Thrd}_t}, \mathbb{x}^g, \mathbb{g}^g) \in \text{Conf}\), \(c^i @ ([T, pc_T^i, z_T^i, t_T^i]_{T \in \text{Thrd}_t}, \mathbb{x}^i, \mathbb{g}^i) \in \text{Conf}\),
\(\mathbb{S}, \mathbb{H}\) \begin{align*}
\mathbb{S}, \mathbb{H} \in \text{Conf} \quad \text{and} \quad T' \in \text{Thrd}_e \quad \text{such that} \quad \\
c^f \xrightarrow{\text{prog}} \cdots \xrightarrow{\text{prog}} c^g \land \\
0 \leq f < g \leq n \land \\
\forall h \in \{f, \ldots, g-2\} : (T' \not\in \text{Thrd}^h_{\text{exe}} \lor \\
\exists \text{lck} \in \text{Lck} : (\text{STM}(T', pc^h_{T'}) = [\text{lck \ lck}]^{pc^h_{T'}} \land \\
\text{OWN}(\mathbb{H}^h+1 \ lck) \neq T')) \land \\
T' \in \text{Thrd}_{\text{exe}}^{g-1} \land \\
\forall \text{lck} \in \text{Lck} : \text{STM}(T', pc^{g-1}_{T'}) = [\text{lck \ lck}]^{pc^{g-1}_{T'}} \Rightarrow \text{OWN}(\mathbb{H}^g \ lck) = T' \land \\
\text{pc}^f_{T'} = pc^g_{T'} \land \\
z^f_{T'} \in \gamma_{\text{reg}}(\mathbb{H}^f_{T'}) \land \\
t^f_{T'} \in \gamma_{\text{reg}}(\mathbb{H}^f_{T'}) \land \\
\exists \mathbb{S}' \in \gamma_{\text{var}}(\mathbb{S}) : \forall x \in \text{Var} : \forall T \in \text{Thrd} : ((\mathbb{S}' x) T) \subseteq ((\mathbb{S} x) T) \land \\
\forall \text{lck} \in \text{Lck} : ((\text{OWN}(\mathbb{H}^f \ lck) \neq \bot_{\text{thrd}} \Rightarrow (\text{STM}(\mathbb{H}^f \ lck) = \text{STM}(\mathbb{H}^f \ lck) \land \\
\text{OWN}(\mathbb{H}^f \ lck) = \text{OWN}(\mathbb{H}^f \ lck) \land \\
\text{DL}(\mathbb{H}^f \ lck) \in \gamma_t(\text{DL}(\mathbb{H}^f \ lck)) \land \\
\text{POWN}(\mathbb{H}^f \ lck) = \text{POWN}(\mathbb{H}^f \ lck) \land \\
\text{REL}(\mathbb{H}^f \ lck) \in \gamma_t(\text{REL}(\mathbb{H}^f \ lck)) \land \\
\text{min}(\gamma_t(\text{DL}(\mathbb{H}^f \ lck))) = -\infty) \land \\
\text{OWN}(\mathbb{H}^f \ lck) = \bot_{\text{thrd}} \Rightarrow ((\text{OWN}(\mathbb{H}^f \ lck) = \text{OWN}(\mathbb{H}^f \ lck) \lor \\
(\text{OWN}(\mathbb{H}^f \ lck) = T' \land \\
\text{STM}(\mathbb{H}^f \ lck) = \text{unlocked} \land \\
t^f_{\mathbb{H}} \in \gamma_t(\text{DL}(\mathbb{H}^f \ lck)) \land \\
\text{min}(\gamma_t(\text{DL}(\mathbb{H}^f \ lck))) = -\infty) \land \\
\text{POWN}(\mathbb{H}^f \ lck) = \text{POWN}(\mathbb{H}^f \ lck) \land \\
\text{REL}(\mathbb{H}^f \ lck) \in \gamma_t(\text{REL}(\mathbb{H}^f \ lck))) \land \\
(|\text{Thrd}_{\text{exe}}^d| \neq 1 \lor \{T \in \text{Thrd}_{\text{exe}}^d \mid \exists r \in \text{Reg}_T : \exists x \in \text{Var}_g : \text{STM}(T, pc^d_{T}) = [\text{load \ r \ from \ x}]^{pc^d_{T}} = \emptyset\})
\end{align*}

where \text{Thrd}_{\text{exe}}^d \text{ is as defined in Table } 4.5 \text{ (on the transition between } c^{g-1} \text{ and } c^g, \text{ T' will execute its active statement and if this statement is } \text{lck \ lck} \text{ for some lock } \text{lck}, \text{ then } T' \text{ successfully acquires } \text{lck}; \text{ if the active statement of } T' \text{ is } \text{lck \ lck}, \text{ then } T' \text{ might have been spin-waiting before being assigned the ownership of } \text{lck}; \text{ no unsafe loading of a global variable value occurs in the abstract case).} \text{ This is the induction assumption. Then it is easy to see that there exists}
a $\tilde{c}^j @ \langle [T, pc^j_T, \tilde{x}^j_T, \tilde{a}^j_T] \mid T \in \text{Thrd}_e, \tilde{x}^j, \tilde{a}^j \rangle \in \text{Conf}$, such that
\[
\tilde{c}^j \xrightarrow{pc^j_T} \ldots \xrightarrow{pc^j_T} \tilde{c}^j \land \\
p c^j_T = pc^j_T \land \\
\tilde{x}^j_T \in \gamma_{\text{reg}}(\tilde{x}^j_T) \land \\
\tilde{a}^j_T \in \gamma_i(\tilde{a}^j_T) \land \\
\exists x' \in \gamma_{\text{var}}(\tilde{x}^j) : (\forall x \in \text{Var} : ((x^{\tilde{c}^j} x) T') \subseteq ((x' x) T')) \land \\
\forall lck \in \text{Lck} : ((\text{Own}(\tilde{c}^j lck) = T') \lor \\
\text{Own}(\tilde{c}^j lck) = T') \Rightarrow (\text{Stt}(\tilde{c}^j lck) = \text{Stt}(\tilde{c}^j lck) \land \\
\text{Own}(\tilde{c}^j lck) = \text{Own}(\tilde{c}^j lck) \land \\
\text{Dl}(\tilde{c}^j lck) \in \gamma_i(\tilde{Dl}(\tilde{c}^j lck)) \land \\
\text{Pown}(\tilde{c}^j lck) = \text{Pown}(\tilde{c}^j lck) \land \\
\text{Rel}(\tilde{c}^j lck) \in \gamma_i(\tilde{Rel}(\tilde{c}^j lck)) \land \\
\text{Min}(\gamma_i(\tilde{Dl}(\tilde{c}^j lck))) = -\infty))
\]
as long as no unsafe loading of a global variable value occurs on the trace between $\tilde{c}^j$ and $\tilde{c}^j$ (Lemmas 5.56, 5.57 and 5.58). Note that even if for some lock, $lck \in \text{Lck}$, $T'$ issues lock $lck$ but $lck$ is assigned to some other thread, $T'$ will eventually be assigned $lck$ so that it can acquire it (since all threads that want to acquire a lock eventually will be able to do so). For such cases, $T'$ is the owner of lock in $c^g$ and $\tilde{c}^j$ (cf. Lemmas 5.57 and 5.58).

Now consider the base case for the induction part of the proof. Since $c^0 \in \gamma_{\text{conf}}(c^0)$, $\forall lck \in \text{Lck} : \text{Min}(\gamma_i(\tilde{Dl}(\tilde{c}^0 lck))) = -\infty$ and $c^0$ is valid, it is easy to see that
\[
\forall T \in \text{Thrd}_e : (pc^0_T = pc^0_T \land \\
\tilde{x}^0_T \in \gamma_{\text{reg}}(\tilde{x}^0_T) \land \\
\tilde{a}^0_T \in \gamma_i(\tilde{a}^0_T) \land \\
\exists x' \in \gamma_{\text{var}}(\tilde{x}^0) : (\forall x \in \text{Var} : ((x^0 x) T) \subseteq ((x' x) T)) \land \\
\forall lck \in \text{Lck} : (\text{Stt}(\tilde{c}^0 lck) = \text{Stt}(\tilde{c}^0 lck) \land \\
\text{Own}(\tilde{c}^0 lck) = \text{Own}(\tilde{c}^0 lck) \land \\
\text{Dl}(\tilde{c}^0 lck) \in \gamma_i(\tilde{Dl}(\tilde{c}^0 lck)) \land \\
\text{Pown}(\tilde{c}^0 lck) = \text{Pown}(\tilde{c}^0 lck) \land \\
\text{Rel}(\tilde{c}^0 lck) \in \gamma_i(\tilde{Rel}(\tilde{c}^0 lck)) \land \\
\text{Min}(\gamma_i(\tilde{Dl}(\tilde{c}^0 lck))) = -\infty)
\]
which means that as long as $\forall h \in \{0, \ldots, k - 1\} : (|\text{Thrd}_e|^{c^h} \neq 1 \lor \{T \in \text{Thrd}_e \mid \exists r \in \text{Reg}_T : \exists x \in \text{Var}_g : \text{Stm}(T, pc^h_T) = [\text{load } r \text{ from } x]|pc^h_T \} = \emptyset)$
(no unsafe loading of a global variable value occurs on the transition sequence between $\tilde{c}^0$ and $\tilde{c}^k$) for the resulting $\tilde{c}^k$, such that $c^n \in \gamma_{\text{conf}}(\tilde{c}^k)$, the induction holds for all threads in $\text{Thrd}_c$.

Note that by definition, $\tilde{lck}^n = \tilde{lck}^0$ (cf. Tables 4.2 and 4.5) and $\tilde{lck}^k = \tilde{lck}^0$ (cf. Tables 5.12 and 5.13) if $lck$ is never acquired by any thread, or never released by its initially owning thread (i.e., the owner of $lck$ in $c^0$ and $c^0$, respectively).

This concludes the proof.

Because of the unsafe nature of $\xrightarrow{\text{prg}}$ (i.e., it cannot safely approximate all concrete transition sequences), it cannot be directly used to derive a safe set of possible final configurations (i.e., configurations such that all threads are issuing $\text{halt}$). It must instead be encapsulated by an algorithm that uses it in a safe manner and handles the unsafe situations explicitly. Such an algorithm is defined in the next chapter.
Chapter 6

Safe Execution Time Analysis by Abstract Execution

In this chapter, an algorithm for deriving safe timing bounds of PPL programs will be defined. The analysis will be based on the abstraction of the PPL semantics presented in Chapter 5. Examples where the presented analysis is used are given in Chapter 7.

NOTE.

A summary of the notation and nomenclature used in this thesis can be found in Appendix A.

6.1 Abstract Execution

The abstract execution function, \( \text{ABS \ EXE :} \ (P(C \ \text{Conf}) \times \text{Time}) \rightarrow (P(C \ \text{Conf}) \times P(C \ \text{Conf}) \times P(C \ \text{Conf})) \), defined in Algorithm 6.1, is a worklist algorithm that encapsulates \( \rightarrow \) _prg_ and explicitly handles the problems discussed in the previous chapter. The input to the algorithm, \( \widetilde{C} \in P(C \ \text{Conf}) \) and \( \widetilde{tto} \in \text{Time} \), is a set of abstract initial configurations (i.e., states) and a timeout value. A configuration, fetched from the worklist, \( \widetilde{C_w} \), will not be further considered if the timeout value is exceeded by the accumulated execution times of all threads in 163.
Chapter 6

Safe Execution Time Analysis by Abstract Execution

In this chapter, an algorithm for deriving safe timing bounds of PPL programs will be defined. The analysis will be based on the abstraction of the PPL semantics presented in Chapter 5. Examples where the presented analysis is used are given in Chapter 7.

NOTE. A summary of the notation and nomenclature used in this thesis can be found in Appendix A.

6.1 Abstract Execution

The abstract execution function, \(\text{ABS\ EXE} : (\mathcal{P}(\text{Conf}) \times \text{Time}) \rightarrow (\mathcal{P}(\text{Conf}) \times \mathcal{P}(\text{Conf}) \times \mathcal{P}(\text{Conf}))\), defined in Algorithm 6.1, is a worklist algorithm that encapsulates \(\rightarrow_{\text{prg}}\) and explicitly handles the problems discussed in the previous chapter. The input to the algorithm, \(\tilde{C} \in \mathcal{P}(\text{Conf})\) and \(\tilde{t}_{\text{to}} \in \text{Time}\), is a set of abstract initial configurations (i.e., states) and a timeout value. A configuration, fetched from the worklist, \(\tilde{C}^w\), will not be further considered if the timeout value is exceeded by the accumulated execution times of all threads in
that configuration. The timeout will be further discussed below.

The overall strategy of the algorithm is depicted in Figure 6.1. Given some safely approximated (by \( \tilde{c}_0 \in \text{CoIf} \)) concrete configuration, \( c_0 \in \text{Conf} \), there is an abstract transition sequence (which is safe for each thread individually) for each possible concrete transition sequence starting from \( c_0 \). If the concrete sequence reaches a final state configuration (i.e., a state where the active statement of each thread is halt as further discussed below), \( c_q \in \text{Conf} \), then so will the corresponding abstract sequence and the concrete final state configuration will be safely approximated (considering all threads) by the abstract final state configuration, \( \tilde{c}_p \in \text{CoIf} \). Note that \( c_1, c_2, \ldots, c_{q-1} \in \text{Conf} \) might not be safely approximated to their entirety by any of the abstract configurations \( \tilde{c}_1, \tilde{c}_2, \ldots, \tilde{c}_{p-1} \in \text{CoIf} \) because of problem 1, defined in Chapter 5 on page 118. Although, it should be noted that for each thread individually, there are abstract configurations among these that safely approximate all the concrete states of that thread on the given concrete transition sequence.

For each thread that issues a load-statement on some global variable while not being the sole thread in Thrd$_{exe}$, given some considered configuration from the worklist, \( \tilde{c}_0^w \), ABS$\text{EXE}$ removes that thread from the configuration and calls itself recursively (with an adapted timeout value) to derive all the possible values that could be loaded by the thread. Note that this is possible since the state for variables is a mapping from variables and threads to a set of time-stamped values (cf. Section 5.5) and since no trimming of the variable state is performed in this case (cf. Table 5.13). When the possible values have been derived, they are merged and put in the target register for the thread that issues the load-statement. Next, a configuration in which the load-statements have been performed is added to the worklist. If no load-statement on some global variable is issued in any thread, or a thread issuing such a load-statement is the sole thread that will execute on a transition, \( \tilde{c}_0 \) is used to derive a set of succeeding configurations, which are then added to the worklist. This strategy addresses problem 2, defined on page 118.

Problem 3, defined on page 118, is partly addressed in the definition of \( \tilde{c}_0 \).
Algorithm 6.1 Abstract execution

1: function ABS_EXE($\tilde{C}, \tilde{t}_0$)
2:   $\tilde{C}^w \leftarrow \tilde{C}$; $\tilde{C}^f \leftarrow \emptyset$; $\tilde{C}^d \leftarrow \emptyset$; $\tilde{C}^r \leftarrow \emptyset$
3:   while $\tilde{C}^w \neq \emptyset$ do
4:      $\tilde{c} @ \langle [T, pc_T, \tilde{t}_T, \tilde{t}_f^T]_{T \in \text{Thrd}_T}, \tilde{x}, \tilde{y} \rangle \leftarrow \text{CHOOSE}(\tilde{C}^w)$
5:      $\tilde{C}^w \leftarrow \tilde{C}^w \setminus \{\tilde{c}\}$
6:      if $\text{ISFINAL}(\tilde{c})$ then
7:         $\tilde{C}^f \leftarrow \tilde{C}^f \cup \{\tilde{c}\}$
8:      else if $\text{IS_DEADLOCK}(\tilde{c})$ then
9:         $\tilde{C}^d \leftarrow \tilde{C}^d \cup \{\tilde{c}\}$
10:     else if $\text{IS_TIMEOUT}(\tilde{c}, \tilde{t}_0)$ then
11:        $\tilde{C}^r \leftarrow \tilde{C}^r \cup \{\tilde{c}\}$
12:     else if $\text{IS_VALID}(\tilde{c}, \tilde{t}_0)$ then
13:        $\text{Thrd}^\text{load} \leftarrow \text{EXE_LOAD_THRD}(\tilde{c})$
14:        if $\text{Thrd}^\text{load} \neq \emptyset$ then
15:           $(\hat{[\tilde{t}_0^T])_{T \in \text{Thrd}^\text{load}}} \leftarrow \langle \hat{[\tilde{t}_0^T + \text{ABSTIME}(\tilde{c}, T)]_{T \in \text{Thrd}^\text{load}}} \rangle$
16:           for all $T \in \text{Thrd}^\text{load}$ do
17:              $x \leftarrow \text{GET_VAR_LOAD}(\text{STM}(T, pc_T))$
18:              $r \leftarrow \text{GET_REG_LOAD}(\text{STM}(T, pc_T))$
19:              $\tilde{c}' \leftarrow \langle [T', pc_T, \tilde{t}_T', \tilde{t}_f^T]_{T \in \text{Thrd}_T \setminus \{T\}, \tilde{x}', \tilde{y}'] \rangle$
20:              $(\tilde{C}'_f, \tilde{C}'_d, \tilde{C}'_r) \leftarrow \text{ABS_EXE}(\{\tilde{c}\'}, \tilde{t}_f^T \tilde{f}_r (\tilde{t}_0 \cup_T \alpha_t(\{-\infty\})))$
21:              $\tilde{\text{STM}} \leftarrow \tilde{\text{STM}}[r \mapsto \tilde{x}']$
22:              for all $\langle T, \tilde{x}', \tilde{y}' \rangle \in (\tilde{C}'_f \cup \tilde{C}'_d \cup \tilde{C}'_r \cup \{\tilde{c}\})$ do
23:                 $\tilde{\text{STM}}[r \mapsto (\tilde{\text{STM}}[r] \cup \tilde{x}' \cup \tilde{y}')]$
24:          end for
25:     end if
26:     end if
27:     end for
28:     $(\hat{[pc_T])_{T \in \text{Thrd}^\text{load}}} \leftarrow \langle [pc_T + 1]_{T \in \text{Thrd}^\text{load}} \rangle$
29:     $(\hat{[\tilde{t}_0^T])_{T \in \text{Thrd}^\text{load}}} \leftarrow \langle [\tilde{t}_0^T]_{T \in \text{Thrd}^\text{load}} \rangle$
30:     $\tilde{C}^w \leftarrow \tilde{C}^w \cup \{([T, pc_T, \tilde{t}_T, \tilde{t}_f^T]_{T \in \text{Thrd}_T}, \tilde{x}, \tilde{y})\}$
31:     else
32:        $\tilde{C}^w \leftarrow \tilde{C}^w \cup \{\tilde{c} \in \text{Conf} \mid \tilde{c} \xrightarrow{\text{prog}} \tilde{c}'\}$
33:     end if
34: end while
35: return $(\tilde{C}^f, \tilde{C}^d, \tilde{C}^r)$
36: end function
(cf. Table 5.13), as discussed in the previous chapter. \( \text{ABSExe} \) fully addresses
the problem by collecting all the possible transitions (i.e., resulting configura-
tions) and adding them to the worklist. Problem 4, defined on page 119, is
addressed by identifying deadlocked configurations and aborting their transi-
tions.

It should be noted that \( \text{ABSExe} \), as defined here, might not terminate for all
possible inputs. This matter will be further discussed in Chapter 8 and Section
10.3.

A configuration is said to be in the final state if all threads are issuing the
\texttt{halt}-statement. A configuration is said to be deadlocked if it cannot possibly
reach the final state according to the semantic transition rules. A configuration
is said to be timed-out if the final state cannot possibly be reached before a
given point in time as represented by the timeout, \( \tilde{t}_{to} \), according to the semantic
transition rules. A configuration is said to have valid concrete counterparts if it
represents at least one concrete configuration that can semantically occur. Two
cases for which a configuration lacks concrete counterparts are when a dead-
lock involves a non-acquired lock and when the owner of a non-acquired lock
misses to acquire it before the expiration of the owner assignment deadline.
Such configurations are discontinued. Note that a configuration representing a
lock owner assignment where the owner of some lock has not yet acquired the
lock, and the owner’s accumulated execution time has not passed the owner as-
signment deadline, reaches a configuration with possibly valid concrete counter-
parts if the owner issues a \texttt{lock}-statement on (i.e., acquires) the lock before the
expiration of the deadline. See Chapter 7 for examples of the described
cases.

If a concrete configuration that is abstracted by some configuration in \( \tilde{C} \)
could be executed into a final configuration, \( \text{ABSExe}(\tilde{C}, \tilde{t}_{to}) \) will either find
a final abstract configuration that safely represents the timing behavior of the
concrete final configuration or reach a timeout due to the value of \( \tilde{t}_{to} \) for the
corresponding abstract transition sequence (whenever the algorithm actually
terminates), and thus, \( \tilde{C}' \neq \emptyset \) (cf. Theorem 6.8 on page 177).

If a concrete configuration that is abstracted by some configuration in \( \tilde{C} \)
could be executed into a state from which a final configuration cannot be
reached, \( \text{ABSExe}(\tilde{C}, \tilde{t}_{to}) \) will either find the corresponding abstract situation
(i.e., \( \tilde{C}' \neq \emptyset \)) or reach a timeout for the corresponding abstract transition se-
quence (whenever the algorithm actually terminates), and thus, \( \tilde{C}' \neq \emptyset \) (cf.
Theorem 6.8).

It is also the case that if \( \tilde{C}' \cup \tilde{C}' = \emptyset \) (whenever the algorithm actually termi-
nates), then all concrete configurations represented by the abstract configura-
tions in $\tilde{C}$ are guaranteed to, along all possible paths, reach a state in which all threads issue the `halt'-statement; i.e., reach the final state, or in other words, terminate (cf. Theorem 6.8).

$\text{ABSEXE}(\tilde{C}, \tilde{t_0})$ hence safely approximates the timing behavior of all threads in any valid concrete configuration represented by an abstract configuration in the input set, $\tilde{C}$, up until $\tilde{t_0}$, which corresponds to the concrete time-point $t_0 = \max(\gamma_l(\tilde{t_0}))$, whenever the algorithm terminates (cf. Theorem 6.8). It should be noted that if a transition sequence is aborted before a final state configuration is reached (e.g., because a deadlocked or timed-out configuration is identified), then an infinite WCET must be assumed for that transition sequence (cf. the algorithm defined in Section 6.2).

The auxiliary functions used in the definition of $\text{ABSEXE}$ are further discussed here. $\text{CHOOSE}(S)$, defined in Algorithm 6.2, gives a deterministically chosen element from the set $S$.

**Algorithm 6.2** Choose an element

```plaintext
1: function CHOOSE(S)
2: Require: $S \neq \emptyset$
3: return a deterministically chosen elements of S
4: end function
```

Given a graph, $(V, E)$, $\text{CYCLE}(V, E)$, as defined in Algorithm 6.3, means that $(V, E)$ contains at least one cycle (Lemma 6.1); cf. Tarjan’s algorithm for finding strongly connected components in a graph [121] which could be slightly modified for increased efficiency and then used as a substitute for the presented algorithm.

**Lemma 6.1 (Soundness of CYCLE):**
If the graph $(V, E)$, where $V$ is a set of vertices and $E$ is a set of edges (i.e., pairs of vertices) connecting the vertices, then $\text{CYCLE}(V, E)$ iff $(V, E)$ contains at least one cycle. □

**Proof.** Assume that $(V, E)$ is a graph, where $V$ is a set of vertices and $E$ is a set of edges connecting the vertices. By definition, a cycle involving vertices $v_1, v_2, v_3, \ldots, v_n$ is described by the edges $(v_1, v_2), (v_2, v_3), \ldots, (v_n, v_1)$, where $n \geq 2$. Thus it is easy to see that a vertex, $v \in V$, cannot be part of a cycle in $(V, E)$ if $\neg \exists v' \in V : (v', v) \in E$; i.e., if $v$ has no incoming edges. It is also easy to see that the graph $(V', E')$, where $V' = V \setminus \{v \in V \mid \neg \exists v' \in V : (v', v) \in E\}$ (i.e., all vertices without incoming edges are removed) and $E' = E \setminus \{(v, v') \in E \mid v \in \{v'' \in V \mid \neg \exists v''' \in V : (v''', v'') \in E\} \land v' \in V\}$, (i.e., all edges going out
Algorithm 6.3 Determine if graph has cycles

1: function CYCLE($V, E$
2: $V' \leftarrow V$
3: $E' \leftarrow E$
4: while $V' \neq \emptyset$ do
5:     $V'' \leftarrow \{ v \in V' \mid \exists v' \in V' : (v', v) \in E' \}$
6:     if $V'' = \emptyset$ then
7:         return true
8:     else
9:         $E' \leftarrow E' \setminus \{(v, v') \in E' \mid v \in V'' \land v' \in V'\}$
10:     $V' \leftarrow V' \setminus V''$
11: end if
12: end while
13: return false
14: end function

from a vertex without incoming edges are removed) contains exactly as many cycles as $(V, E)$.

Thus it must be that if this procedure can be repeated until an empty graph is reached, there are no cycles in the initial graph. Likewise, if it is not possible to reduce the initial graph to an empty graph, there must be at least one cycle in the initial graph. If there is no cycle in the initial graph, it is easy to see that the graph can be reduced to the empty graph by the above procedure. Likewise it is easy to see that if there is a cycle in the initial graph, the graph cannot be reduced to the empty graph by the above procedure. But then it must be that $\text{CYCLE}(V, E)$ iff the graph $(V, E)$ contains at least one cycle.

Given a configuration, $\tilde{c} \in \text{Conf}$, $\text{EXETHRD}(\tilde{c})$, as defined in Algorithm 6.4, is an over-approximation of $\text{Thrde}_\text{exe}$ as defined in Table 5.13 (Lemma 6.2).

Lemma 6.2 (Soundness of EXETHRD):

Given $\tilde{c} \in \text{Conf}$, $\text{Thrde}_\text{exe} \subseteq \text{EXETHRD}(\tilde{c})$, where $\text{Thrde}_\text{exe}$ is as defined in Table 5.13. \qed

Proof. Based on $\tilde{c} @ ([T, p_{CT}, \tilde{z}_T, \tilde{y}_T]_{T \in \text{Thrde}_\text{exe}}: \tilde{x}, \tilde{I}) \in \text{Conf}$, assume that $\tilde{t}$ is as defined in Algorithm 6.4 and that $\tilde{r}$ is defined as $\tilde{t}$ given by Table 5.13. It is easy to see that $\text{Thrde}_{\text{hold}}$ as given by Algorithm 6.4 is a superset of $\text{Thrde}_{\text{hold}}$ as given by Table 5.13 since in the latter case, a lock might have been assigned...
Algorithm 6.4 Threads to execute in an abstract configuration

1: function EXETHRD(\(c\)@ \([T, pcT, lck]|T \in \text{Thrd}_c\))
2: \(\text{Thrd}_{\text{hold}} \leftarrow \{ T \in \text{Thrd}_c \mid \text{STM}(T, pcT) = [\text{halt}]^{pcT} \lor \exists lck \in \text{Lck} :\)
   \((\text{STM}(T, pcT) = [\text{lock} lck]^{pcT} \land \text{OWN}(\| lck) \neq T)\} \)
3: \(\langle [\text{lck}] T \in \text{Thrd}_c \rangle \leftarrow \langle [\text{ABSTIME}(\text{c}, T)] T \in \text{Thrd}_c \rangle\)
4: \(t_{\text{min}} \leftarrow \min \{ \min(\gamma_T(\text{lck}^a \| lck) : T \in \text{Thrd}_c \backslash \text{Thrd}_{\text{hold}})\}\}
5: \(t_{\text{max}} \leftarrow \min \{ \max(\gamma_T(\text{lck}^a \| lck) : T \in \text{Thrd}_c \backslash \text{Thrd}_{\text{hold}})\}\}
6: \(\bar{t} \leftarrow \alpha_t(t_{\text{min}}, t_{\text{max}})\)
7: return \(\{ T \in \text{Thrd}_c \mid \text{Thrd}_{\text{hold}} \mid \exists lck \in \text{Lck} :\)
   \((\text{STM}(T, pcT) = [\text{lock} lck]^{pcT} \land \text{OWN}(\| lck) = \bot_{\text{thrd}})\}\)∪
   \(\{ T \in \text{Thrd}_{\text{hold}} \mid \exists lck \in \text{Lck} :\)
   \((\text{STM}(T, pcT) = [\text{lock} lck]^{pcT} \land \text{OWN}(\| lck) = \bot_{\text{thrd}})\}\)
8: end function

to some thread, which will exclude that thread from \(\text{Thrd}_{\text{hold}}\). Thus it must be that \(\min(\gamma_T(\bar{t})) \leq \min(\gamma_T(\bar{t}))\) and \(\max(\gamma_T(\bar{t})) \leq \max(\gamma_T(\bar{t}))\) (since \(\bar{t}\) is derived based on a superset of the threads used to derive \(\bar{t}\)); note that if it is a true superset, it must be that all the extra threads issue \(\text{lock} lck\) for some locks, \(lck \in \text{Lck}\), and have been assigned the ownership of \(lck\), and that for those locks \(\text{OWN}(\| lck) = \bot_{\text{thrd}}\). Thus it must be that \(\text{Thrd}_{\text{exe}}^c \subseteq \text{EXETHRD}(\text{c})\), where \(\text{Thrd}_{\text{exe}}^c\) is as defined in Table 5.13, since \(\text{EXETHRD}(\text{c})\) is derived based on \(\bar{t}\) but also includes all threads issuing \(\text{lock} lck\) where \(lck \in \text{Lck}\) and \(\text{OWN}(\| lck) = \bot_{\text{thrd}}\).

Given a set of threads, \(\text{Thrd}_c \subseteq \text{Thrd}\), \(\text{GLOBALVAR}(\text{Thrd}_c)\), as defined in Algorithm 6.5, is the set of variables that could transfer data between some of the threads in \(\text{Thrd}_c\) (Lemma 6.3); i.e., the variables in the set are such that they could be read by some thread and also written to by at least one other thread.

Algorithm 6.5 Global variables in an abstract configuration

1: function GLOBALVAR(\(\text{Thrd}_c\))
2: \(\langle \{x \in \text{Var} \mid \exists r \in \text{Reg}_T : \exists l \in \text{Lbl}_T : \text{STM}(T, l) = [\text{load} r \text{ from } x] l\} T \in \text{Thrd}_c\rangle \leftarrow\)
3: \(\langle \{x \in \text{Var} \mid \exists r \in \text{Reg}_T : \exists l \in \text{Lbl}_T : \text{STM}(T, l) = [\text{store} r \text{ to } x] l\} T \in \text{Thrd}_c\rangle \leftarrow\)
4: return \(\{x \in \text{Var} \mid \exists T, T' \in \text{Thrd}_c : (T \neq T' \land x \in \text{Var}^\text{load}_T \land x \in \text{Var}^\text{store}_T)\}\)
5: end function
Lemma 6.3 (Soundness of \texttt{GLOBALVAR}): 

\texttt{GLOBALVAR}(\texttt{Thrd}_\ddagger) is the set of variables (called global variables) for which a data dependency between two or more threads can occur in the program described by \texttt{Thrd}_\ddagger.

PROOF. Assume that $x \in \texttt{Var}$. First note that

- if $\{T \in \texttt{Thrd}_\ddagger \mid \exists l \in \texttt{Lbl}_T : \exists r \in \texttt{Reg}_T : [\texttt{store} \ r \ \texttt{to} \ x]^l\} = \emptyset$ (i.e., no thread ever writes to $x$), then it must be that $x$ can be considered a constant (since $x \in \texttt{Var}$, there must be some thread reading from it),

- if $\{T \in \texttt{Thrd}_\ddagger \mid \exists l \in \texttt{Lbl}_T : \exists r \in \texttt{Reg}_T : [\texttt{load} \ r \ \texttt{from} \ x]^l\} = \emptyset$ (i.e., no thread ever reads from $x$), then it must be that $x$ can be considered a trash variable (since $x \in \texttt{Var}$, there must be some thread writing to it),

- if, for some thread, $T' \in \texttt{Thrd}_\ddagger$, $\{T \in \texttt{Thrd}_\ddagger \mid \exists l \in \texttt{Lbl}_T : \exists r \in \texttt{Reg}_T : [\texttt{store} \ r \ \texttt{to} \ x]^l\} = \{T'\}$ and $\{T \in \texttt{Thrd}_\ddagger \mid \exists l \in \texttt{Lbl}_T : \exists r \in \texttt{Reg}_T : [\texttt{load} \ r \ \texttt{from} \ x]^l\} = \{T'\}$, then it must be that $x$ is only read from and written to by $T'$ (thus there cannot be any data dependency on $x$ between two threads), and

- for a data dependency to occur on $x$ for two threads, $T', T'' \in \texttt{Thrd}_\ddagger$, it must be that $T' \in \{T \in \texttt{Thrd}_\ddagger \mid \exists l \in \texttt{Lbl}_T : \exists r \in \texttt{Reg}_T : [\texttt{store} \ r \ \texttt{to} \ x]^l\}$, $T'' \in \{T \in \texttt{Thrd}_\ddagger \mid \exists l \in \texttt{Lbl}_T : \exists r \in \texttt{Reg}_T : [\texttt{load} \ r \ \texttt{from} \ x]^l\}$ and $T' \neq T''$ (there must be at least one thread, $T'$, that writes to $x$ and at least one other thread, $T''$, that reads from $x$ somewhere in the program for $T''$ to be data dependent on $T'$ via $x$).

Thus, since for each $T \in \texttt{Thrd}_\ddagger$, the set $\texttt{Var}^\text{load}_T$ contains all variables that $T$ might read from and the set $\texttt{Var}^\text{store}_T$ contains all variables that $T$ might write to, it must be that $\{x \in \texttt{Var} \mid \exists T', T'' \in \texttt{Thrd}_\ddagger : (T' \neq T'' \land x \in \texttt{Var}^\text{load}_T \land x \in \texttt{Var}^\text{store}_T)\}$ is the set of variables for which data dependencies occur between at least two threads.

Given a configuration, $\ddagger \in \texttt{Conf}$, \texttt{EXELOADTHRD}(\ddagger), as defined in Algorithm 6.6, is a set of threads that might issue a load-statement on a global variable in a transition from \ddagger (Lemma 6.4).

Lemma 6.4 (Soundness of \texttt{EXELOADTHRD}): 

Given a configuration $\ddagger@([T,pc_T,\ddagger_T,l_T]_{T \in \texttt{Thrd}_\ddagger},\ddagger,T)$ \in \texttt{Conf}, $\{T \in \texttt{Thrd}_\ddagger_{\text{exe}} \mid \exists r \in \texttt{Reg}_T : \exists x \in \texttt{GLOBALVAR}(\texttt{Thrd}_\ddagger) : \texttt{stm}(T,pc_T) = [\texttt{load} \ r \ \texttt{from} \ x]^r_T\} \subseteq \texttt{EXELOADTHRD}(\ddagger)$, where $\texttt{Thrd}_\ddagger_{\text{exe}}$ is defined as in Table 5.13.

\hfill $\square$
Lemma 6.4 (Soundness of \textsc{Exe Load Thrd}): variable in a transition from

Thus, since for each \( T \)

The proof follows directly from the fact that \( \text{Thrd}_{\text{exe}} \subseteq \text{ExeThrd}(\bar{c}) \) (Lemma 6.2).

Given an abstract configuration, \( \bar{c} \in \text{Conf} \), ISF\( \text{INAL}(\bar{c}) \), as defined in Algorithm 6.7, means that \( \bar{c} \) is in the final state; i.e., all threads issue the \texttt{halt}-

Given an abstract configuration, \( \bar{c} \in \text{Conf} \), IS\( \text{EADLOCK}(\bar{c}) \), as defined in Algorithm 6.8, means that \( \bar{c} \) cannot reach a final state according to the abstract semantic rules (Lemma 6.5). Note that IS\( \text{EADLOCK} \) is not guaranteed to identify all such cases, though.

Algorithm 6.6: Threads executing a possibly unsafe \texttt{load}-

Algorithm 6.7: Final abstract configuration

Algorithm 6.8: Deadlocked abstract configuration

Require: \( \neg\text{ISF\( \text{INAL}(\bar{c}) \))

\( \text{Thrd}_{\text{lock}} \leftarrow \{T \in \text{Thrd}_{\bar{c}} \mid \exists \text{\texttt{lock}} \in \text{Lck} : (\text{STM}(T, p_{CT}) = [\texttt{lock \texttt{lock}}]_{p_{CT}}^{p_{CT}} \wedge \right.

\( \text{Thrd}_{\text{lock}} \leftarrow \{T \in \text{Thrd}_{\bar{c}} \mid \exists \text{\texttt{lock}} \in \text{Lck} : (\text{STM}(T, p_{CT}) = [\texttt{lock \texttt{lock}}]_{p_{CT}}^{p_{CT}} \wedge \right.

\( \text{E} \leftarrow \{\langle T, T' \rangle \in \text{Thrd}_{\text{lock}} \times \text{Thrd}_{\text{lock}} \mid \exists \text{\texttt{lock}} \in \text{Lck} : (\text{STM}(T, p_{CT}) = [\texttt{lock \texttt{lock}}]_{p_{CT}}^{p_{CT}} \wedge \right.

\( \text{return} \text{Thrd}_{\bar{c}} = \text{Thrd} \land (\text{CYCLE}(\text{Thrd}_{\text{lock}}, E) \lor \exists T \in \text{Thrd}_{\bar{c}} : \exists \text{\texttt{lock}} \in \text{Lck} : (\text{STM}(T, p_{CT}) = [\texttt{lock \texttt{lock}}]_{p_{CT}}^{p_{CT}} \wedge \right.

\( \text{end function} \)
Lemma 6.5 (Soundness of isDeadlock):

Given a configuration $\bar{c} @ ([T, pc_T, \bar{x}_T, \bar{r}_T]_{T \in \text{Thrd}_{\bar{c}}}, \bar{x}, \bar{l}) \in \text{Conf}$, such that $\exists T \in \text{Thrd}_{\bar{c}} : \text{STM}(T, pc_T) \neq [\text{halt}]^{pc_T}$, $\text{isDeadlock}(\bar{c}) \Rightarrow \forall c \in \gamma_{\text{conf}}(\bar{c}) : ¬\exists c'@([T, pc'_T, \bar{x}'_T, \bar{r}'_T]_{T \in \text{Thrd}_{\bar{c}}}, \bar{x}', \bar{l}') \in \text{Conf} : (c \xrightarrow{\text{prg}} \ldots \xrightarrow{\text{prg}} c' \land \forall T \in \text{Thrd}_{\bar{c}} : \text{STM}(T, pc'_T) = [\text{halt}]^{pc'_T}$, where $c$ and $c'$ are valid concrete configurations (cf. Definition 4.4); i.e., if $\text{isDeadlock}(\bar{c})$, then $\bar{c}$ does not represent any concrete configuration that can possibly reach a final state.

PROOF. Assume that $\bar{c} @ ([T, pc_T, \bar{x}_T, \bar{r}_T]_{T \in \text{Thrd}_{\bar{c}}}, \bar{x}, \bar{l}) \in \text{Conf}$, such that $\exists T \in \text{Thrd}_{\bar{c}} : \text{STM}(T, pc_T) \neq [\text{halt}]^{pc_T}$ (note that this assumption fulfills $¬\text{ISFINAL}(\bar{c})$) and that $\text{isDeadlock}(\bar{c})$. Note that it must be that $\text{Thrd}_{\bar{c}} = \text{Thrd}$ (otherwise, $¬\text{isDeadlock}(\bar{c})$).

Since $\text{Thrd}_{\text{lock}} = \{T \in \text{Thrd} \mid \exists lck \in \text{Lck} : (\text{STM}(T, pc_T) = [\text{lock} lck]^{pc_T} \land \text{OWN}(\bar{l} lck) \notin \{\bot_{\text{thrd}}, T\} \land \text{S\texttt{TT}}(\bar{l} lck) = \text{locked})\}$ and $E = \{(T, T') \mid T, T' \in \text{Thrd}_{\text{lock}} \land \exists lck \in \text{Lck} : (\text{STM}(T, pc_T) = [\text{lock} lck]^{pc_T} \land \text{OWN}(\bar{l} lck) = T')\}$, it is easy to see that $(\text{Thrd}_{\text{lock}}, E)$ is a graph where the vertices (in $\text{Thrd}_{\text{lock}}$) represent threads that are waiting to acquire a lock that is currently acquired (i.e., owned and locked) by some other thread, and each edge, $(T, T') \in E$, describes a dependency (i.e., $T$ is waiting to acquire a lock that is currently acquired by $T'$). But then it must be that if $\text{cycle}(\text{Thrd}_{\text{lock}}, E)$, then for all $c \in \gamma_{\text{conf}}(\bar{c})$ (such that $c$ is valid) there exists a deadlock in $c$ (since $(\text{Thrd}_{\text{lock}}, E)$ contains at least one cycle; Lemma 6.1), and thus, $\forall c \in \gamma_{\text{conf}}(\bar{c}) : ¬\exists c'@([T, pc'_T, \bar{x}'_T, \bar{r}'_T]_{T \in \text{Thrd}_{\bar{c}}}, \bar{x}', \bar{l}') \in \text{Conf} : (c \xrightarrow{\text{prg}} \ldots \xrightarrow{\text{prg}} c' \land \forall T \in \text{Thrd}_{\bar{c}} : \text{STM}(T, pc'_T) = [\text{halt}]^{pc'_T}$) (a final configuration, $c'$, cannot be reached from $c$). (Note that if for some thread, $T \in \text{Thrd}_{\bar{c}}$, and lock, $lck \in \text{Lck}$, $\text{STM}(T, pc_T) = [\text{lock} lck]^{pc_T}$, $\text{OWN}(\bar{l} lck) \notin \{\bot_{\text{thrd}}, T\}$ and $\text{S\texttt{TT}}(\bar{l} lck) = \text{unlocked}$, then $T \notin \text{Thrd}_{\text{lock}}$ since $\text{OWN}(\bar{l} lck)$ has not yet acquired $lck$; deadlock-cycles involving this case are further considered in Algorithm 6.10.)

If $\exists T \in \text{Thrd}_{\bar{c}} : \exists lck \in \text{Lck} : (\text{STM}(T, pc_T) = [\text{lock} lck]^{pc_T} \land \text{S\texttt{TT}}(\bar{l} lck) = \text{locked} \land \text{OWN}(\bar{l} lck) \neq \bot_{\text{thrd}} \land \text{STM}(\text{OWN}(\bar{l} lck), pc_{\text{OWN}(\bar{l} lck)}) = [\text{halt}]^{pc_{\text{OWN}(\bar{l} lck)}}}$ (T is waiting for a lock owned by a terminated thread to be released), it is easy to see that $\text{OWN}(\bar{l} lck)$ will never issue $\text{unlock} lck$ (cf. Tables 5.12 and 5.13) and thus $\forall c \in \gamma_{\text{conf}}(\bar{c}) : ¬\exists c'@([T, pc'_T, \bar{x}'_T, \bar{r}'_T]_{T \in \text{Thrd}_{\bar{c}}}, \bar{x}', \bar{l}') \in \text{Conf} : (c \xrightarrow{\text{prg}} \ldots \xrightarrow{\text{prg}} c' \land \forall T \in \text{Thrd}_{\bar{c}} : \text{STM}(T, pc'_T) = [\text{halt}]^{pc'_T}$) (a final configuration, $c'$, cannot be reached from $c$).

This concludes the proof. □
Given an abstract configuration, \( \varepsilon \in \text{Conf} \), and a timeout, \( \tilde{t}_{\text{to}} \in \text{Tim}_e \), \( \text{isTimeout}(\varepsilon, \tilde{t}_{\text{to}}) \), as defined in Algorithm 6.9, means that a final state cannot be reached from \( \varepsilon \) before \( \tilde{t}_{\text{to}} \) has passed according to the abstract semantic rules (Lemma 6.6). Note that \( \text{isTimeout} \) might not identify all possible such cases.

**Algorithm 6.9** Timed-out abstract configuration

1. **function** \( \text{isTimeout}(\varepsilon@\langle[T,pc_T,\tilde{z}_T,\tilde{t}_{T}^a]_{T\in\text{Thrd}_e},\tilde{x},\tilde{t}\rangle,\tilde{t}_{\text{to}}) \)

2. **Require:** \( \neg\text{isFinal}(\varepsilon) \land \neg\text{isDeadlock}(\varepsilon) \)

3. **return** \( \forall T \in \text{Thrd}_e : (\text{STM}(T,pc_T) \neq \text{halt}[^{pc_T}] \Rightarrow (\tilde{t}_{\text{to}} \leq T \text{halt}(\varepsilon, T) \lor (\text{Thrd}_e \subset \text{Thrd} \land \exists lck \in Lck : (\text{STM}(T,pc_T) = \text{lock lck}[^{pc_T}] \land \text{OWN}(\varepsilon, lck) \not\in \{\bot_{\text{thread}}, T\})))) \)

**Lemma 6.6 (Soundness of \( \text{isTimeout} \)):**

Given a configuration, \( \varepsilon@\langle[T,pc_T,\tilde{z}_T,\tilde{t}_{T}^a]_{T\in\text{Thrd}_e},\tilde{x},\tilde{t}\rangle \in \text{Conf} \), and timeout, \( \tilde{t}_{\text{to}} \in \text{Tim}_e \), such that \( \exists T \in \text{Thrd}_e : (\text{STM}(T,pc_T) \neq \text{halt}[^{pc_T}] \land \neg\text{isDeadlock}(\varepsilon) \land \text{isTimeout}(\varepsilon, \tilde{t}_{\text{to}}) \Rightarrow \forall c \in \gamma_{\text{conf}}(\varepsilon) : \neg\exists c'@\langle[T,pc_T',\tilde{z}_T',\tilde{t}_{T}^a']_{T\in\text{Thrd}_e},\tilde{x}',\tilde{t}'\rangle \in \text{Conf} : (c_{\text{prg}} \rightarrow \ldots \rightarrow c'_{\text{prg}} \land \forall T \in \text{Thrd}_e : (\text{STM}(T,pc_T') = \text{halt}[^{pc_T}] \land \tilde{t}_{T}^a' \leq \max(\gamma_{\text{to}}(\tilde{t}_{\text{to}})))) \), where \( c \) and \( c' \) are valid concrete configurations (cf. Definition 4.4); i.e., if \( \text{isTimeout}(\varepsilon, \tilde{t}_{\text{to}}) \), then \( \varepsilon \) does not represent any concrete configuration that can possibly reach a final state before the given timeout (i.e., before max(\( \gamma_{\text{to}}(\tilde{t}_{\text{to}}) \))).

---

**Explanation of the Lemma.** A configuration is timed-out (a final state cannot be reached from the configuration within the given timeout value) if all non-terminated threads are such that their accumulated execution times exceed the given timeout value, or a safe value for some global variable is being derived (i.e., \( \text{Thrd}_e \subset \text{Thrd} \)) and the thread is waiting to acquire a lock which it does not own.

**Proof.** Assume that \( \varepsilon@\langle[T,pc_T,\tilde{z}_T,\tilde{t}_{T}^a]_{T\in\text{Thrd}_e},\tilde{x},\tilde{t}\rangle \in \text{Conf} \) and \( \tilde{t}_{\text{to}} \in \text{Tim}_e \) are such that \( \exists T \in \text{Thrd}_e : (\text{STM}(T,pc_T) \neq \text{halt}[^{pc_T}] \land \neg\text{isDeadlock}(\varepsilon) \land \text{isTimeout}(\varepsilon, \tilde{t}_{\text{to}}) \). Since \( \text{isTimeout}(\varepsilon, \tilde{t}_{\text{to}}) \), it must be that \( \forall T \in \text{Thrd}_e : (\text{STM}(T,pc_T) \neq \text{halt}[^{pc_T}] \Rightarrow (\tilde{t}_{\text{to}} \leq T \text{halt}(\varepsilon, T) \lor (\text{Thrd}_e \subset \text{Thrd} \land \exists lck \in Lck : (\text{STM}(T,pc_T) = \text{lock lck}[^{pc_T}] \land \text{OWN}(\varepsilon, lck) \not\in \{\bot_{\text{thread}}, T\})))) \) (all non-terminated threads are such that their accumulated execution times exceed the
given timeout value, or a safe value for some global variable is being derived (\(\text{Thrd}_c \subset \text{Thrd}\)) and the thread is waiting to acquire a lock which it does not own. For all threads, \(T \in \text{Thrd}_c\), such that \(\text{stm}(T, pc_T) \neq [\text{halt}]^{pc_T} \land \bar{t}_{to} \not< t\) (the threads that have not terminated and whose accumulated execution times exceed the given timeout value), it is trivially the case that they cannot terminate before their accumulated execution times exceed the timeout value; i.e., \(\neg\exists c' \in \langle [T, pc_T, t', t'] \mid T \in \text{Thrd}_c, \bar{x}, \bar{l}' \rangle \in \text{Conf}: (c \stackrel{pq} \rightarrow \ldots \stackrel{pq} \rightarrow c' \land \text{stm}(T, pc_T) = [\text{halt}]^{pc_T} \land \bar{t}_{to}' \not< \max(\gamma(T, \bar{t}_{to}))\) (cf. Assumptions 4.1 and 5.51). Thus, for all other threads, \(T \in \text{Thrd}_c\), such that \(\exists lck \in \text{Lck} : (\text{stm}(T, pc_T) = [\text{lock } lck]^{pc_T} \land \text{OWN}(\bar{l} lck) \not\in \{\bot_{thrd}, T\})\) (the threads that are waiting to acquire a lock which is currently owned by some other thread), it must be that \(\neg\exists c' \in \langle [T, pc_T, t', t'] \mid T \in \text{Thrd}_c, \bar{x}, \bar{l}' \rangle \in \text{Conf}: (c \stackrel{pq} \rightarrow \ldots \stackrel{pq} \rightarrow c' \land \text{stm}(T, pc_T) = [\text{halt}]^{pc_T} \land \bar{t}_{to}' \not< \max(\gamma(T, \bar{t}_{to}))\) (a final configuration, \(c'\), cannot be reached from \(c\) before the timeout is exceeded) since the respective locks cannot possibly be released at any time, \(t\), such that \(t \leq \max(\gamma(T, \bar{t}_{to}))\) (cf. Assumptions 4.1 and 5.51).

This concludes the proof.

Given an abstract configuration, \(\bar{c} \in \text{Conf}\), and a timeout, \(\bar{t}_{to} \in \text{Time}\), \(\neg\text{isVALID}(\bar{c}, \bar{t}_{to})\), where \(\text{isVALID}(\bar{c}, \bar{t}_{to})\) is as defined in Algorithm 6.10, means that \(\bar{c}\) cannot reach a configuration that could represent at least one valid (cf. Definition 4.4) concrete configuration (Lemma 6.7). Note that \(\text{isVALID}\) might not identify all possible such cases, though.

**Algorithm 6.10 Valid abstract configuration**

1. **function** \(\text{isVALID}(\bar{c} \circ \langle [T, pc_T, t_T, t_T'] \mid T \in \text{Thrd}_c, \bar{x}, \bar{l}\rangle, \bar{t}_{to})\)

   **Require:** \(\neg\text{isFINAL}(\bar{c}) \land \neg\text{isDEADLOCK}(\bar{c}) \land \neg\text{isTIMEOUT}(\bar{c}, \bar{t}_{to})\)

2. \(\text{Thrd}_{\text{lock}} \leftarrow \{T \in \text{Thrd}_c \mid \exists lck \in \text{Lck} : (\text{stm}(T, pc_T) = [\text{lock } lck]^{pc_T} \land \text{OWN}(\bar{l} lck) \not\in \{\bot_{thrd}, T\})\}\)

3. \(E \leftarrow \{(T, T') \in \text{Thrd}_{\text{lock}} \times \text{Thrd}_{\text{lock}} \mid \exists lck \in \text{Lck} : (\text{stm}(T, pc_T) = [\text{lock } lck]^{pc_T} \land \text{OWN}(\bar{l} lck) = T')\}\)

4. **return** (\(\text{Thrd}_c = \text{Thrd} \Rightarrow \neg\text{CYCLE}(\text{Thrd}_{\text{lock}}, E)\)) \(\land \forall lck \in \text{Lck} : \forall T \in \text{Thrd}_c : ((\text{OWN}(\bar{l} lck) = T \land \text{STM}(lck) = \text{unlocked}) \Rightarrow (\text{stm}(T, pc_T) \neq [\text{halt}]^{pc_T} \land\)

   \(\text{DL}(\bar{l} lck) \not< (\bar{t}_T + \bar{t}_{ABSTRACT}(\bar{c}, T)))\))

5. **end function**
Lemma 6.7 (Soundness of ISVALID):
Given a configuration $\tilde{c} @ ([T, pc_T, \tilde{\tau}_T, \tilde{\tau}_T])_{\tilde{\tau}_T} \in \tilde{\text{Thrd}}$, and an abstract time, $\tilde{t}_{to}$, such that $\exists T \in \tilde{\text{Thrd}}_{\tilde{c}} : \text{STM}(T, pc_T) \neq [\text{halt}]^{pc_T}$, $\neg \text{ISDEADLOCK}(\tilde{c})$, and $\neg \text{ISTIMEOUT}(\tilde{c}, \tilde{t}_{to})$, then $\neg \text{ISVALID}(\tilde{c}, \tilde{t}_{to}) \Rightarrow \neg \exists \tilde{c}' \in \tilde{\text{Conf}} : \tilde{c}' \rightarrow \tilde{c}' \land \exists c @ ([T, pc_T, \tau_T, \tau_T])_{\tau_T} \in \text{Conf} (\tilde{c}') : (\exists \tilde{c}' \in \text{Conf} : \tilde{c}' \rightarrow \tilde{c}' \land \forall \text{lck} \in \text{Lck} : (\text{STT}(\tilde{lck}) = \text{unlocked} \Rightarrow \text{OWN}(\tilde{lck}) = \bot_{thrd})$; i.e., $\tilde{c}$ can never lead to a configuration that could represent at least one valid concrete configuration (cf. Definition 4.4).

**EXPLANATION OF THE LEMA.** A configuration is valid if it is not final, deadlocked, or timed-out, and there are no cycles in the lock-dependency graph involving assigned, but not yet acquired, locks and each thread that is assigned some lock that it has not yet acquired has not terminated its execution and the accumulated execution time of the thread has not succeeded the deadline for acquiring the lock. An abstract configuration is thus not valid in case there is no configuration that can be reached from it such that that configuration corresponds to at least one concrete configuration in which all unlocked locks are unassigned; i.e., a valid concrete configuration cannot be reached (cf. Definition 4.4).

**PROOF.** Assume that $\tilde{c} @ ([T, pc_T, \tilde{\tau}_T, \tilde{\tau}_T])_{\tilde{\tau}_T} \in \tilde{\text{Conf}}$ and $\tilde{t}_{to} \in \tilde{\text{Time}}$ are such that $\exists T \in \tilde{\text{Thrd}}_{\tilde{c}} : \text{STM}(T, pc_T) \neq [\text{halt}]^{pc_T}$, $\neg \text{ISDEADLOCK}(\tilde{c})$, and $\neg \text{ISTIMEOUT}(\tilde{c}, \tilde{t}_{to})$. Then it must be that

1. $\neg (\text{Thrd}_{\tilde{c}} = \text{Thrd} \Rightarrow \neg \text{CYCLE}(\text{Thrd}_{\text{lock}}, E))$ (i.e., $\text{Thrd}_{\tilde{c}} = \text{Thrd} \land \text{CYCLE}(\text{Thrd}_{\text{lock}}, E)$), where $\text{Thrd}_{\text{lock}} = \{T \in \text{Thrd}_{\tilde{c}} | \exists \text{lck} \in \text{Lck} : (\text{STM}(T, pc_T) = [\text{lock lck}]^{pc_T} \land \text{OWN}(\tilde{lck}) \neq \{\bot_{thrd}, T\})\}$

and $E = \{(T, T') | T, T' \in \text{Thrd}_{\text{lock}} \land \exists \text{lck} \in \text{Lck} : (\text{STM}(T, pc_T) = [\text{lock lck}]^{pc_T} \land \text{OWN}(\tilde{lck}) = T')\}$ (there is a cycle in the lock-dependency graph involving a non-acquired lock since $\neg \text{ISDEADLOCK}(\tilde{c})$), or

2. $\neg \forall \text{lck} \in \text{Lck} : \forall T \in \text{Thrd}_{\tilde{c}} : ((\text{OWN}(\tilde{lck}) = T \land \text{STT}(\tilde{lck}) = \text{unlocked}) \Rightarrow (\text{STM}(T, pc_T) \neq [\text{halt}]^{pc_T} \land \text{DL}(\tilde{lck}) \tilde{\tau}_T \tilde{\tau}_T, \text{ABSTIME}(\tilde{c}, T)))$ (there is some thread that is assigned some lock that it has not yet acquired, and the thread has terminated its execution or is such that its accumulated execution time succeeds the deadline for acquiring the lock).

If $\text{Thrd}_{\tilde{c}} = \text{Thrd} \land \text{CYCLE}(\text{Thrd}_{\text{lock}}, E)$, then it must be that there is a cycle in the dependency graph, $(\text{Thrd}_{\text{lock}}, E)$, for threads waiting to acquire
some lock (Lemma 6.1). Since ¬isDEADLOCK(\(\overline{c}\)), it must be that this cycle involves at least one lock, \(lck \in \text{Lck}\), such that for some thread, \(T \in \text{Thrd}\), \(\text{OWN}(\overline{\text{Lck}}) \not\in \{\bot_{\text{thrd}}, T\}\) and \(\text{STT}(\overline{\text{Lck}}) = \text{unlocked}\) (cf. Algorithm 6.8). But then it is easy to see that \(\overline{c}\) can never lead to a configuration that could represent at least one valid concrete configuration \(\neg\exists c' \in \text{Conf} : (\overline{c} \xrightarrow{p_{\text{prg}}} \ldots \xrightarrow{p_{\text{prg}}} c') \land \exists c @ ([T, p_{\text{prg}}, x_{\text{prg}}, t_{\text{prg}}]_{T \in \text{Thrd}}, x, \overline{\emptyset}) \in \gamma_{\text{conf}}(c') : (\exists c' \in \text{Conf} : c' \xrightarrow{p_{\text{prg}}} \ldots \xrightarrow{p_{\text{prg}}} c) \land \forall lck \in \text{Lck} : (\text{STT}(\overline{lck}) = \text{unlocked} \Rightarrow \text{OWN}(\overline{lck}) = \bot_{\text{thrd}}))\) (cf. Tables 5.12 and 5.13 and Lemma 4.5).

Note that if \(\neg\forall lck \in \text{Lck} : \forall T \in \text{Thrd}_T : ((\text{OWN}(\overline{\text{Lck}}) = T \land \text{STT}(\overline{lck}) = \text{unlocked}) \Rightarrow (\text{STM}(T, p_{\text{prg}}) \not= [\text{halt}]^{p_{\text{prg}}} \land \text{DL}(\overline{lck}) \not\in T_i (i^u + t, \text{ABSTIME}(\overline{\text{Lck}}, T)))\), then it must logically be that \(\exists lck \in \text{Lck} : \exists T \in \text{Thrd}_T : (\text{OWN}(\overline{\text{Lck}}) = T \land \text{STT}(\overline{lck}) = \text{unlocked} \land (\text{STM}(T, p_{\text{prg}}) = [\text{halt}]^{p_{\text{prg}}} \lor \text{DL}(\overline{lck}) \not\in T_i (i^u + t, \text{ABSTIME}(\overline{\text{Lck}}, T))))\).

If \(\exists lck \in \text{Lck} : \exists T \in \text{Thrd}_T : (\text{OWN}(\overline{\text{Lck}}) = T \land \text{STT}(\overline{lck}) = \text{unlocked} \land \text{STM}(T, p_{\text{prg}}) = [\text{halt}]^{p_{\text{prg}}})\), then it is easy to see that \(\neg\exists c' \in \text{Conf} : (\overline{c} \xrightarrow{p_{\text{prg}}} \ldots \xrightarrow{p_{\text{prg}}} c') \land \exists c @ ([T, p_{\text{prg}}, x_{\text{prg}}, t_{\text{prg}}]_{T \in \text{Thrd}_T}, x, \overline{\emptyset}) \in \gamma_{\text{conf}}(c') : (\exists c' \in \text{Conf} : c' \xrightarrow{p_{\text{prg}}} \ldots \xrightarrow{p_{\text{prg}}} c) \land \forall lck \in \text{Lck} : (\text{STT}(\overline{lck}) = \text{unlocked} \Rightarrow \text{OWN}(\overline{lck}) = \bot_{\text{thrd}})))\) (\(\overline{c}\) can never lead to a configuration that could represent at least one valid concrete configuration) since, for the given lock, the owner will be \(T\) but the state will remain \text{unlocked} for all configurations following \(\overline{c}\) (cf. Tables 5.12 and 5.13).

If \(\exists lck \in \text{Lck} : \exists T \in \text{Thrd}_T : (\text{OWN}(\overline{\text{Lck}}) = T \land \text{STT}(\overline{lck}) = \text{unlocked} \land \text{DL}(\overline{lck}) \not\in T_i (i^u + t, \text{ABSTIME}(\overline{\text{Lck}}, T)))\), then it is easy to see that \(\neg\exists c' \in \text{Conf} : (\overline{c} \xrightarrow{p_{\text{prg}}} \ldots \xrightarrow{p_{\text{prg}}} c') \land \exists c @ ([T, p_{\text{prg}}, x_{\text{prg}}, t_{\text{prg}}]_{T \in \text{Thrd}_T}, x, \overline{\emptyset}) \in \gamma_{\text{conf}}(c') : (\exists c' \in \text{Conf} : c' \xrightarrow{p_{\text{prg}}} \ldots \xrightarrow{p_{\text{prg}}} c) \land \forall lck \in \text{Lck} : (\text{STT}(\overline{lck}) = \text{unlocked} \Rightarrow \text{OWN}(\overline{lck}) = \bot_{\text{thrd}})))\) (\(\overline{c}\) can never lead to a configuration that could represent at least one valid concrete configuration) since \(\text{OWN}(\overline{\text{Lck}})\) must be one of the threads that issue \text{lock} \(lck\) (and thus determines \text{DL}(\overline{lck})) in the concrete case for \(\forall lck \in \text{Lck} : (\text{STT}(\overline{lck}) = \text{unlocked} \Rightarrow \text{OWN}(\overline{lck}) = \bot_{\text{thrd}})\) to hold. But since \(\text{DL}(\overline{lck}) \stackrel{?}{=} T_i (i^u + t, \text{ABSTIME}(\overline{\text{Lck}}, T))\) and \(\text{DL}(\overline{lck}) \in \gamma_i (\text{DL}(\overline{lck}))\) for any given transitional sequence (cf. Lemma 5.59), there cannot be any \(c \in \gamma_{\text{conf}}(c')\) such that \(\text{OWN}(\overline{lck}) = T\) (cf. Assumptions 4.1 and 5.51 and Tables 4.2, 4.5, 5.12 and 5.13).

This concludes the proof. \(\blacksquare\)

Given a \text{load}-statement, \(s\), \text{GETVARLOAD}(s)\), as defined in Algorithm 6.11, is the variable, and \text{GETREGLOAD}(s)\), as defined in Algorithm 6.12, is the register, defined by the statement.
6.1 Abstract Execution

Algorithm 6.11 Get variable in load-statement

1: function GETVARLOAD([load r from x])
2:  return x
3: end function

Algorithm 6.12 Get register in load-statement

1: function GETREGLOAD([load r from x])
2:  return r
3: end function

Theorem 6.8 states that, as discussed above, $\text{ABSEX} \tilde{C}, \tilde{t}_{io}$ (as defined in Algorithm 6.1 on page 165) gives a sound approximation of the timing behavior of all configurations of the concrete initial configurations corresponding to the abstract configurations found in $\tilde{C}$ up until $t_{io} = \max(\gamma(\tilde{t}_{io}))$ whenever it terminates. Assuming that the algorithm terminates, the theorem also states some important properties of the output sets; $\tilde{C}^f$, $\tilde{C}^d$ and $\tilde{C}^i$. If $\tilde{C}^d$ and $\tilde{C}^i$ are empty, then the analyzed program is guaranteed to terminate given the considered initial states and safe approximations of all corresponding concrete final states are found in $\tilde{C}^f$. If $\tilde{C}^d$ is not empty, then it could be that the program deadlocks for some of the initial states. And, if $\tilde{C}^i$ is not empty, then it could be that the program does not terminate also due to some other reason (at least the program might not terminate before $t_{io}$).

Theorem 6.8 (Soundness of $\text{ABSEX}$):
If the sets of valid configurations $C \in \mathcal{P}(\text{Conf})$ (cf. Definition 4.4) and $\tilde{C} \in \mathcal{P}(\tilde{\text{Conf}})$, are such that $\forall c @ \langle [T,p\tilde{c}_T,\tilde{z}_T,\tilde{t}_{io}]\rangle \in \tilde{C} : (\forall [T,p\tilde{c}_T,\tilde{z}_T,\tilde{t}_{io}]\rangle \in \tilde{C} : (\text{Thrd}_1 = \text{Thrd}_2 = \text{Thrd}) ) \wedge \exists \tilde{c} \in \tilde{C} : c \in \gamma_{conf}(\tilde{c})) \wedge |\text{Thrd}| < \infty \wedge \forall c @ \langle [T,p\tilde{c}_T,\tilde{z}_T,\tilde{t}_{io}]\rangle \in \tilde{C} : \forall l \in \text{Lck} : \min(\gamma(\text{DL}(l \text{ lck}) = -\infty))$, then given $\tilde{t}_{io} \in \text{Time}$, $(\tilde{C}^f,\tilde{C}^d,\tilde{C}^i) \in  

s\in \mathcal{S}(\tilde{C}^i)$ and when $\tilde{C}^d$ is non-empty, then it could be that the program deadlocks for some of the initial states. And, if $\tilde{C}^i$ is non-empty, then it could be that the program does not terminate also due to some other reason (at least the program might not terminate before $t_{io}$).

Theorem 6.8 (Soundness of $\text{ABSEX}$):
If the sets of valid configurations $C \in \mathcal{P}(\text{Conf})$ (cf. Definition 4.4) and $\tilde{C} \in \mathcal{P}(\tilde{\text{Conf}})$, are such that $\forall c @ \langle [T,p\tilde{c}_T,\tilde{z}_T,\tilde{t}_{io}]\rangle \in \tilde{C} : (\forall [T,p\tilde{c}_T,\tilde{z}_T,\tilde{t}_{io}]\rangle \in \tilde{C} : (\text{Thrd}_1 = \text{Thrd}_2 = \text{Thrd}) ) \wedge \exists \tilde{c} \in \tilde{C} : c \in \gamma_{conf}(\tilde{c})) \wedge |\text{Thrd}| < \infty \wedge \forall c @ \langle [T,p\tilde{c}_T,\tilde{z}_T,\tilde{t}_{io}]\rangle \in \tilde{C} : \forall l \in \text{Lck} : \min(\gamma(\text{DL}(l \text{ lck}) = -\infty))$, then given $\tilde{t}_{io} \in \text{Time}$, $(\tilde{C}^f,\tilde{C}^d,\tilde{C}^i) \in  

\( \text{ABS} \text{EXE}(\vec{C}, t_{io}) \) is such that

\[ \forall c \in C : \forall c' @ (\{T, pc_T', z_T', t_{T'}\}) \in \text{Conf} : \]

\[ ((c \overset{\text{prg}}{\longrightarrow} \ldots \overset{\text{prg}}{\longrightarrow} c') \land \forall T \in \text{Thrd} : \text{STM}(T, pc_T') = [\text{halt}]^{pc_T'} \Rightarrow \]

\[ (\vec{c}' \neq \emptyset \lor \exists \vec{c} @ (\{T, pc_T', z_T', t_{T'}\}) \in \text{Thrd} : \vec{x}', \vec{l}' \in \vec{C} : \forall T \in \text{Thrd} : \]

\[ (pc_T = pc_T' \land t_{T'} \in \gamma_t(\vec{l}_T')) \land \]

\[ \forall c \in C : \forall c' @ (\{T, pc_T', z_T', t_{T'}\}) \in \text{Conf} : \]

\[ ((c \overset{\text{prg}}{\longrightarrow} \ldots \overset{\text{prg}}{\longrightarrow} c') \land (\text{CYCLE}(\text{Thrd'}_{\text{lock}}, E')) \land \]

\[ \exists T \in \text{Thrd} : \exists \text{lck} \in \text{Lck} : \]

\[ (\text{STM}(T, pc_T') = [\text{lock lck}]^{pc_T'} \land \]

\[ \text{OWN}(\vec{l}' \text{lck}) \notin \{\perp_{\text{thrd}}, T\} \land \]

\[ \text{STM}(\text{OWN}(\vec{l}' \text{lck}), pc_{\text{OWN}(\vec{l}' \text{lck}})) = \]

\[ \text{halt}_{\text{OWN}(\vec{l}' \text{lck})})] \Rightarrow \]

\[ (\vec{c}' \neq \emptyset \lor \vec{c} \neq \emptyset) \]

where \( \text{Thrd'}_{\text{lock}} = \{T \in \text{Thrd} | \exists \text{lck} \in \text{Lck} : (\text{STM}(T, pc_T') = [\text{lock lck}]^{pc_T'} \land \]

\[ \text{OWN}(\vec{l}' \text{lck}) \notin \{\perp_{\text{thrd}}, T\})\} \) and \( E' = \{(T, T') | T, T' \in \text{Thrd'}_{\text{lock}} \land \exists \text{lck} \in \text{Lck} : \]

\[ (\text{STM}(T, pc_T') = [\text{lock lck}]^{pc_T'} \land \text{OWN}(\vec{l}' \text{lck}) = T')\}, \) whenever it terminates.

Furthermore, if \( \vec{C} \cup \vec{c} = \emptyset \), then:

\[ \forall c \in C : \forall c' @ (\{T, pc_T', z_T', t_{T'}\}) \in \text{Thrd} : \vec{x}', \vec{l}' \in \text{Conf} : \]

\[ ((c \overset{\text{prg}}{\longrightarrow} \ldots \overset{\text{prg}}{\longrightarrow} c') \land \forall T \in \text{Thrd} : \text{STM}(T, pc_T') = [\text{halt}]^{pc_T'} \Rightarrow \]

\[ \exists \vec{c} @ (\{T, pc_T', z_T', t_{T'}\}) \in \text{Thrd} : \vec{x}', \vec{l}' \in \vec{C} : (pc_T = pc_T' \land t_{T'} \in \gamma_t(\vec{l}_T')) \]

\[ \square \]

**Explanation of the Theorem.** For each valid concrete configuration, \( c \), from which final configurations can be reached, there is either an abstract configuration in \( \vec{C}' \) (as derived by \text{ABS} \text{EXE}) that safely approximate the timing behaviors of those concrete final configurations, or timed-out configurations that are derived somewhere along the abstract transition sequences (and thus \( \vec{C}' \), as derived by \text{ABS} \text{EXE}, is non-empty). Also, for each valid concrete configuration from which a deadlocked configuration can be reached, there is either a corresponding deadlocked abstract configuration in \( \vec{C} \), as derived by \text{ABS} \text{EXE}, or a timed-out configuration that is derived somewhere along the abstract transition sequence (and thus \( \vec{C}' \), as derived by \text{ABS} \text{EXE}, is non-empty).

Furthermore, if no deadlocked or timed-out abstract configurations are derived based on the given input set of initial abstract configurations,
\( \tilde{C} \), then all possible transition sequences from the configurations in the concrete input set, \( C \), will terminate and the timing behaviors of all final concrete configurations derived from the configurations in \( C \) are safely approximated by some abstract final configurations in \( \tilde{C}^i \).

**Proof.** Given \( \tilde{t}_{i_0} \in \text{Time} \), assume that the sets of valid configurations \( C \in \mathcal{P}(Conf) \) and \( \tilde{C} \in \mathcal{P}(\tilde{Conf}) \) are as assumed in the lemma above and that \( (\tilde{C}^i, \tilde{C}^d, \tilde{C}^t) = \text{ABS EXE}(\tilde{C}, \tilde{t}_{i_0}) \).

This proof will partly be conducted using induction on the considered level of recursion, where level 0 is the base level (i.e., the level where for any considered \( \tilde{c} \), \( \text{Thrd}_i = \text{Thrd} \)) and level \( n \geq 0 \) is the bottom level (i.e., the level from which no more recursion occurs, which is also referred to as the maximum level of recursion), while assuming that all sequentially preceding load-statements in all threads for any considered configuration on any level of recursion have been safely approximated. Before beginning the induction part of the proof, first note that:

- The overall structure of the algorithm is of the worklist type; i.e., given an item (abstract configuration in this case) that is extracted from a worklist, new items are generated, based on some rules, and are either added to the worklist (and will thus eventually be extracted themselves) or saved as output items if some condition is fulfilled. When the worklist is empty, the algorithm terminates.

- Since \( (\tilde{C}^i, \tilde{C}^d, \tilde{C}^t) = \text{ABS EXE}(\tilde{C}, \tilde{t}_{i_0}) \), i.e., \( \text{ABS EXE}(\tilde{C}, \tilde{t}_{i_0}) \) results in a tuple of three sets, it must be that the algorithm terminates for the particular input (i.e., \( \tilde{C} \) and \( \tilde{t}_{i_0} \)).

- The structure of the algorithm is such that, for any \( \tilde{c} \in \tilde{Conf} \) and \( \tilde{t}_{i_0} \in \text{Time} \), \( \text{ISDEADLOCK}(\tilde{c}) \) is only issued when \( \neg \text{ISFINAL}(\tilde{c}) \), \( \text{ISTIMEOUT}(\tilde{c}, \tilde{t}_{i_0}) \) is only issued when \( \neg \text{ISDEADLOCK}(\tilde{c}) \), and \( \text{ISVALID}(\tilde{c}, \tilde{t}_{i_0}) \) is only issued when \( \neg \text{ISTIMEOUT}(\tilde{c}, \tilde{t}_{i_0}) \). This means that the requirements of Algorithms 6.8, 6.9 and 6.10 are fulfilled.

- The timing behaviors of the threads included on any recursion level are safely given by \( \text{ABSTIME} \) (Assumption 5.51).

- When the considered level of recursion, \( i \), is greater than 0, it is easy to see that \( \text{Thrd}_i \subset \text{Thrd} \), where \( \text{Thrd}_i \) is the set of threads included in any configuration on recursion level \( i \) for the considered recursion pattern. Note that \( \text{Thrd}_0 = \text{Thrd} \).
• The maximum level (i.e., depth) of any recursion pattern is $|\text{Thrd}| - 1$ since $|\text{EXE}\text{Thrd}(\bar{c})| > 1$ for recursion to occur and $|\text{Thrd}| \geq |\text{EXE}\text{Thrd}(\bar{c})|$ for any $\bar{c} \in \text{Conf}$ (cf. Algorithm 6.4). Since $|\text{Thrd}| < \infty$, it must thus be that $0 \leq n \leq |\text{Thrd}| - 1 < \infty$. But then, since the recursion depth is of a finite size, it must be that the recursion eventually stops for any considered case.

• The timeout, $\tilde{t}_{to}$, for recursion level $i > 0$ is such that $\max(\gamma_t(\tilde{t}_{to})) \leq \max(\gamma_t(\tilde{t}_{to}^{-1}))$ since $\tilde{t}_{to}^i = \tilde{t}_{to}^{i-1} \cap_\tau \text{EXE}\text{Thrd} \cap_\tau \text{ABS}\text{Time}(\bar{c}^{-1}, T_{i-1})$. Figure 6.2 illustrates a case where $n = 4$, $\tilde{t}_{to}^0$ is the timeout at the base level (i.e., recursion level 0) and for all $i \in \{1, 2, 3, 4\}$, $\tilde{t}_{to}^i$ is the timeout at recursion level $i$, $T_{i-1}$ is the thread not included in configurations at recursion level $i$ and $\tilde{t}_{to}^i = \tilde{t}_{to}^{i-1} \cap_\tau \text{ABS}\text{Time}(\bar{c}^{-1}, T_{i-1})$.

• Assume that a thread, $T \in \text{Thrd}$, issues a potentially unsafe load-statement at some recursion level $i - 2$, where $i \geq 2$, and has hence been removed from all configurations at recursion level $i - 1$ and beyond for the given recursion pattern, and that no events occurring after $\tilde{t}_{to}^{i-1}$ can affect the loaded value. If some other thread, $T' \in \text{Thrd}$, issues a possibly unsafe load-statement at recursion level $i - 1$, then a new recursion level, $i$, will be created to determine a safe write history before the load in $T'$ is evaluated. Then it is easy to see that any event occurring after $\tilde{t}_{to}^i$ cannot affect the value loaded by $T'$. But then it is easy to see that the value loaded by $T$ at recursion level $i - 2$ cannot be affected by any event occurring after $\tilde{t}_{to}^i = \tilde{t}_{to}^{i-1} \cap_\tau \text{EXE}\text{Thrd} \cap_\tau \text{ABS}\text{Time}(\bar{c}^{-1}, T_{i-1})$ for the considered recursion instance at level $i$. Thus, for all recursion levels $i \in \{1, \ldots, n\}$, the timeout for recursion level $i$, as determined by the algorithm, is safe since the accumulated time for a thread cannot decrease (cf. Assumption 5.51).

• The structure of the algorithm (i.e., on a recursion level, one new recursion-instance is created for each thread that is executing a possibly unsafe load-statement) gives that all possible cases, for in which order load-statements in different threads can be issued, are considered.

Assume that, given some configuration, $\bar{c}@([T, pc_T, \tilde{t}_T, \tilde{t}_{to}^i]_{T \in \text{Thrd}}, \bar{x}, \bar{I})$, and timeout, $\tilde{t}_{to}$, on recursion level $i$, where $0 \leq i < n$, some thread, $T_i \in \text{Thrd}, i$,
Figure 6.2: Illustration of how the timeout, $\tilde{t}_{to}$, for a new level of recursion in ABSEXE is determined.
issues a possibly unsafe load-statement (which means that a deeper recursion level, \(i + 1\), will exist) that cannot be affected by any event occurring after \(\tilde{t}_o\). Further assume that the local thread states (i.e., program counters, register values and accumulated execution times) and write history for all variables as given by \(\tilde{c}\) safely approximate the possible concrete thread states and variable values given the considered program point and the corresponding concrete transition sequences (if any), and that all load-statements on recursion levels \(i + 1\) to \(n\) are safely approximated. This comprises the induction assumption.

Now consider the induction step. Since the local thread states and write history for all variables as given by \(\tilde{c}\) safely approximate the possible concrete thread states and variable values given the configuration at the end of the corresponding concrete transition sequences and all load-statements on recursion levels \(i + 1\) to \(n\) are safely approximated, it must be that all \(\tilde{c}' \in \text{Conf}\), such that \(\tilde{c} \rightarrow_{\text{prg}} \ldots \rightarrow_{\text{prg}} \tilde{c}'\), safely approximate the possible concrete thread states and variable values given the considered program point and the corresponding concrete transition sequences since \(\rightarrow_{\text{prg}}\) is used to approximate the execution of all statements except load-statements that are possibly unsafe (cf. Lemmas 5.56, 5.57 and 5.58). Thus, it must be that

\[
(\tilde{C}_{i+1}^f, \tilde{C}_{i+1}^d, \tilde{C}_{i+1}) \rightarrow_{\text{ABS EXE}} \{([T, p c_T^\tilde{c}^f, \tilde{x}_T, \tilde{x}_T^d]_{T \in \text{Thrds}} \setminus \{\tilde{T}_i\}, \tilde{\xi}, \tilde{\eta})\}, (\tilde{T}_i', \tilde{\xi}, \tilde{\eta}), \text{ABSTIME}(\tilde{c}, T_i), \tilde{\alpha}_i(\{-\infty\}))
\]

is such that \(\tilde{\alpha}_i(\{-\infty\})\) safely approximates all possible concrete values that could be read by the load-statement in \(T_i\) for the corresponding concrete transition sequence, since (the \text{ABS EXE} instance mentioned above, corresponding to recursion level \(i + 1\) is considered)

1. \(\{\tilde{c}'' \in \text{Conf} \mid \tilde{c}' \rightarrow_{\text{prg}} \tilde{c}''\}\) safely collects all transition possibilities for any given configuration, \(\tilde{c}' \in \text{Conf}\), or rather, thread, for which no possibly unsafe load-statements are approximated by the transition (cf. Lemmas 5.56, 5.57 and 5.58),

2. \text{TRIM} is not used to remove old writes from the write history since \(i + 1 > 0\) (cf. Table 5.13),

3. (note that \(\text{Thrds}_{i+1} \subset \text{Thrds}\)) \(\forall \tilde{c}' \rightarrow_{\text{ABS EXE}} \{([T, p c_T', \tilde{x}_T', \tilde{x}_T'^d]_{T \in \text{Thrds}}', \tilde{\xi}', \tilde{\eta}') \in \text{Conf} : (\text{Thrds}' \subset \text{Thrds} \Rightarrow \neg \text{IS DEADLOCK}(\tilde{c}')\); i.e., even if a deadlock exists in \(\tilde{c}'\), it is further evaluated just in case there are threads that are not part of the deadlock and thus could affect the value of the variable which is read on the lower recursion level (cf. Algorithm 6.8),
4. for any $c' \in \text{Conf}$ and $\tilde{r}_{t_0} \in \text{Time}$, $\text{ISTIMEOUT}(c', \tilde{r}_{t_0}) \Rightarrow \forall c \in \\
\gamma_{\text{conf}}(c') : \neg\exists c' @ \langle [T, pc', r'_T, T'], \tilde{r}'_{\text{Thrd,}}, \tilde{x}', \tilde{l}' \rangle \in \text{Conf} : (c \xrightarrow{pc} \cdots \xrightarrow{pc}) \\
c' \land \forall T \in \text{Thrd,} : (\text{STM}(T, pc_T') = [\text{halt}]^{pc}_T \land r''_T \leq \max(\gamma_{c}(\tilde{r}_{t_0}))))$, where $c$ and $c'$ are valid concrete configurations (cf. Definition 4.4); i.e., if $\text{ISTIMEOUT}(c', \tilde{r}_{t_0})$, then $c'$ does not represent any concrete configuration that can possibly reach a final state before the given timeout (Lemma 6.6), or in other words, no thread in $c'$ can affect the system state so that the effects are visible at or before $\tilde{r}_{t_0}$ (cf. Algorithm 6.9 and Assumption 5.51).

5. (note that $\text{Thrd}_{i+1} \subset \text{Thrd}$) $\forall c' @ \langle [T, pc', r'_T, T'], \tilde{r}'_{\text{Thrd,}}, \tilde{x}', \tilde{l}' \rangle \in \text{Conf} : ((\text{Thrd} \subset \text{Thrd} \land \neg\text{VALID}(c')) \Rightarrow \forall \text{lck} \in \text{Lck} : \forall T \in \text{Thrd}' : ((\text{OWN}(\tilde{l'} \text{lck}) = T \land \text{STM}(\tilde{l'} \text{lck}) = \text{unlocked}) \Rightarrow (\text{STM}(T, pc_T') \neq [\text{halt}]^{pc}_T \land \tilde{DL}(\tilde{l'} \text{lck}) \neq \alpha_t \langle \tilde{r}_T', +_t \text{ABSTIME}(c, T) \rangle))$, which follows directly from Algorithm 6.10 and means that there is no possibility that $c'$ has any (or could lead to a configuration that has a) valid concrete counterpart (cf. Definition 4.4 and the proof of Lemma 6.7).

It is important to notice that $\text{ISTIMEOUT}$ captures all configurations such that all threads have either executed beyond the timeout or are waiting to acquire a lock that is currently owned by a thread that has executed beyond the timeout or is also waiting to acquire some lock (cf. Algorithm 6.9), which means that the first mentioned thread cannot possibly acquire the lock before the timeout has passed (cf. Tables 5.12 and 5.13 and Assumption 5.51). This means that $\text{ISTIMEOUT}$ captures all deadlocked configurations, since $\text{ISDEADLOCK}$ does not capture any configuration at all when the considered recursion level is greater than 0 (cf. Algorithm 6.8), and also all configurations allowed by $\text{ISVALID}$, although they lack valid concrete counterparts (cf. Algorithm 6.10).

Since $\tilde{r}'_{T_i} = \tilde{r}_T' +_t \text{ABSTIME}(c', T_i)$, $pc_{T_i}' = pc_{T_i} + 1$ and $\tilde{f}'_{T_i}[r \mapsto \mu_{\text{val}}(\text{READ}(\tilde{x}', x, T_i, \tilde{r}_i') +_t \text{ABSTIME}(c', T_i)) | (T, \tilde{x}', \tilde{l}') \in \tilde{C}_{i+1}'] \cup \tilde{C}_{i+1}'] \cup \tilde{C}_{i+1}' \cup \{c'\}]$ (assuming that the possibly unsafe $\text{load}$-statement issued by $T_i$ is $\text{load r from } x[pc_{T_i}$ for some $r \in \text{Reg}_{T_i}$ and $x \in \text{Var}$), it must thus be that the $\text{load}$-statement in thread $T_i$ on recursion level $i$ is safely approximated and that the new configuration, which is added to the worklist on line 28, therefore safely approximates the local thread states (i.e., program counters, register values and accumulated execution times) for all threads and the write history for all variables as given by the possible concrete thread states and variable values in the considered program point and the corresponding concrete transition sequences (if any). But this means that all possibly unsafe $\text{load}$-statements on
recursion level \(i\) are safely approximated. This concludes the induction step part of the proof.

Now consider recursion level \(n\) (i.e., the level from which no more recursion will occur for a given recursion pattern, which is the base case for the induction part of the proof) for the first ever occurring recursion pattern for a given transition sequence, such that no potentially unsafe \(\text{load}\)-statement has yet been approximated. Since no potentially unsafe \(\text{load}\)-statement has yet been approximated and \(\forall c \in C : \exists \tilde{c} \in \tilde{C} : c \in \gamma_{\text{conf}}(\tilde{c})\), it must be that any concrete state for all threads individually, and the write history for each variable, must be safely approximated up until the considered point of the considered transition sequence (since \(\tilde{c} \rightarrow \tilde{c}'\) has been safely used for all transitions and \(\{\tilde{c}' \in \text{Conf} \mid \tilde{c} \rightarrow \tilde{c}'\}\) collects all abstract transition possibilities for any given configuration, \(\tilde{c} \in \text{Conf}\), or rather, thread; cf. Lemmas 5.56, 5.57 and 5.58). Since no more (i.e., deeper) recursion will occur, it must be that for any considered configuration, \(\tilde{c} @ (\{T, pc_T, \tilde{r}_T, \tilde{\theta}_T \mid T \in \text{Thrd}_n\}, \tilde{x}, \tilde{I}) \in \text{Conf}\), at level \(n\), \(|\text{EXETHRD}(\tilde{c})| > 1\) or \(\text{EXELoadTHRD}(\tilde{c}) = \emptyset\). But since for any given configuration \(\tilde{c} @ (\{T, pc_T, \tilde{r}_T, \tilde{\theta}_T \mid T \in \text{Thrd}_n\}, \tilde{x}, \tilde{I}) \in \text{Conf}, \text{Thrd}_n \subseteq \text{EXETHRD}(\tilde{c})\) (Lemma 6.2) and \(\{T \in \text{Thrd}_n \mid \exists r \in \text{Reg}_T : \exists x \in \text{GLOBALVAR}((\text{Thrd}_n) : \text{STM}(T, pc_T) = |\text{load } r \text{ from } x|^{pc_T}\} \subseteq \text{EXELoadTHRD}(\tilde{c})\) (Lemma 6.4), where \(\text{Thrd}_n\) is as defined in Table 5.13, it must thus be that \(|\text{Thrd}_n| \neq 1 \lor \{T \in \text{Thrd}_n \mid \exists r \in \text{Reg}_T : \exists x \in \text{VAR}_x : \text{STM}(T, pc_T) = |\text{load } r \text{ from } x|^{pc_T}\} = \emptyset\) for all \(\tilde{c} \in \text{Conf}\) at recursion level \(n\). Thus, it must be that \(\{\tilde{c}' \in \text{Conf} \mid \tilde{c} \rightarrow \tilde{c}'\}\) will be safely used to collect all the possible transitions on recursion level \(n\) until all threads either reach the final state (i.e., issue halt-statements) or execute beyond the timeout (cf. Lemmas 5.56, 5.57, 5.58, 5.59, 6.5, 6.6 and 6.7). But, then it must be that all the possible concrete transition sequences for each thread are safely approximated up until the timeout point (if ever reached, and if reached before the final state) since \(\forall c \in C : \exists \tilde{c} \in \tilde{C} : c \in \gamma_{\text{conf}}(\tilde{c})\). This concludes the induction part of the proof.

Now consider the different ways the algorithm stops evaluating a given, valid transition sequence and hence the way \(\tilde{C}'\), \(\tilde{C}''\) and \(\tilde{C}'\) are created. Given \(c \in C\) and \(c' @ (\{T, pc'_T, \tilde{r}'_T, \tilde{\theta}'_T \mid T \in \text{Thrd}'\}, \tilde{x}', \tilde{I}') \in \text{Conf}\), the following concrete cases (corresponding to a terminating program, a program reaching a deadlocked state and a more general case of a nonterminating program, respectively) must be considered.

1. Assume that \(c \rightarrow pc_T \ldots \rightarrow c' \land \forall T \in \text{Thrd} : \text{STM}(T, pc_T) = |\text{halt}|^{pc_T}\) (ter-
mining transition sequence). Note that since all possible concrete transition sequences for each thread individually are safely approximated up until the timeout point and a final configuration is reached in the concrete case, there must be an abstract trace of transitions such that all configurations, \( \tilde{c} \in \text{Conf} \), on that trace are such that \( \neg \text{isDEADLOCK}(\tilde{c}) \) (cf. Algorithm 6.8 and Lemma 6.5) and \( \text{isVALID}(\tilde{c}, \tilde{I}_0) \) (cf. Algorithm 6.10 and Lemma 6.7). It must also be that, eventually, a configuration, \( \tilde{c} @ \langle [T, pc_T^{\tilde{c}}, \tilde{x}_T, \tilde{\iota}_T] \rangle \in \text{Conf} \), for which either \( \forall T \in \text{Thrd} : \text{STM}(T, pc_T^{\tilde{c}}) = [\text{halt}]^{pc_T^{\tilde{c}}} \) or \( \forall T \in \text{Thrd} : (\text{STM}(T, pc_T^{\tilde{c}}) \neq [\text{halt}]^{pc_T^{\tilde{c}}} \Rightarrow \tilde{I}_0 \preceq T \\tilde{\iota} T \preceq T, \text{ABSTIME}(\tilde{c}, T)) \) is derived along the corresponding (over-approximating) abstract trace of transitions.

If \( \forall T \in \text{Thrd} : \text{STM}(T, pc_T^{\tilde{c}}) = [\text{halt}]^{pc_T^{\tilde{c}}} \), then it is easy to see that \( \text{isFINAL}(\tilde{c}) \) (cf. Algorithm 6.7), which means that \( \tilde{c} \in \tilde{C} \). Thus, it must be that \( \exists \tilde{c} @ \langle [T, pc_T^{\tilde{c}}, \tilde{x}_T, \tilde{\iota}_T] \rangle \in \tilde{C} \forall T \in \text{Thrd} : (pc_T^{\tilde{c}} = pc_T^{\tilde{c}} \land \tilde{\iota}_T \in \gamma_T(\tilde{\iota}_T)) \).

If \( \exists T \in \text{Thrd} : \text{STM}(T, pc_T^{\tilde{c}}) \neq [\text{halt}]^{pc_T^{\tilde{c}}} \land \forall T \in \text{Thrd} : (\text{STM}(T, pc_T^{\tilde{c}}) \neq [\text{halt}]^{pc_T^{\tilde{c}}} \Rightarrow \tilde{I}_0 \preceq T \\tilde{\iota} T \preceq T, \text{ABSTIME}(\tilde{c}, T)) \), then it is easy to see that \( \neg \text{isFINAL}(\tilde{c}) \), \( \neg \text{isDEADLOCK}(\tilde{c}) \) (since the program terminates in the concrete case) and \( \text{isTIMEOUT}(\tilde{c}, \tilde{I}_0) \) (cf. Algorithms 6.7 and 6.9), which means that \( \tilde{c} \in \tilde{C} \). Thus, it must be that \( \tilde{C} \neq \emptyset \).

2. Assume that \( c \xrightarrow{prg} \ldots \xrightarrow{prg} c' \) and \( (\text{CYCLE}^{\text{Thrd}_{lock}}') \vee \exists T \in \text{Thrd} : \exists lck \in \text{Lck} : (\text{STM}(T, pc_T') = [\text{lck} lck]^{pc_T'} \land \text{OWN}(lck') \not\in \{ \bot, T \}) \land \text{STM(OWN}(lck'), pc_{\text{OWN}(lck')}') = [\text{halt}]^{pc_{\text{OWN}(lck')}'} \) (a transition sequence reaching a deadlock state), where \( \text{Thrd}_{lock}' = \{ T \in \text{Thrd} \mid \exists lck \in \text{Lck} : \langle (\text{STM}(T, pc_T') = [\text{lck} lck]^{pc_T'} \land \text{OWN}(lck') \not\in \{ \bot, T \}) \rangle \} \) and \( E' = \{ (T, T') \mid T, T' \in \text{Thrd}_{lock}' \land \exists lck \in \text{Lck} : \langle (\text{STM}(T, pc_T') = [\text{lck} lck]^{pc_T'} \land \text{OWN}(lck') = T') \rangle \) (remember that \( \text{OWN}(lck') \neq \bot \Rightarrow \text{STT}(lck') = \text{locked} \) since \( c \) is valid and \( \xrightarrow{prg} \) preserves validity; cf. Definition 4.4 and Lemma 4.5). Note that since all possible concrete transition sequences for each thread individually are safely approximated up until the timeout point and a deadlocked configuration is reached in the concrete case, there must be an abstract trace of transitions such that all configurations, \( \tilde{c} \in \text{Conf} \), on that trace are such that \( \neg \text{isFINAL}(\tilde{c}) \) (cf. Algorithm 6.7) and \( \text{isVALID}(\tilde{c}, \tilde{I}_0) \) (cf. Algorithm 6.10 and Lemma 6.7). It must also be that, eventually, a con-
configuration, \( \tilde{c} @ \langle [T, pc^\tilde{c}], \tilde{x}, \tilde{t}^a_T] \rangle \in \text{Conf} \), will be derived (along the corresponding, over-approximating abstract trace of transitions) for which either 
\((\text{CYCLE}(\text{Thrd}^E, E)) \vee \exists T \in \text{Thrd} : \exists \text{lck} \in \text{Lck} : (\text{STM}(T, pc^T) = [\text{locked} \wedge \text{OWN}(\tilde{I} \text{lck}) \neq \perp] \wedge \text{STM}(\tilde{I} \text{lck}, pc^\tilde{c}_\text{OWN}(\tilde{I} \text{lck})) = [\text{halt}]) \neq \emptyset \) (\( \tilde{c} \) is timed-out but not deadlocked), where \( \text{Thrd}^E \) is the set of all concrete configurations, \( \text{STM} = (\text{STM}(T, pc^T) = [\text{locked} \wedge \text{OWN}(\tilde{I} \text{lck}) \neq \perp] \wedge \text{STM}(\tilde{I} \text{lck}, pc^\tilde{c}_\text{OWN}(\tilde{I} \text{lck})) = [\text{halt}]) \), then it is easy to see that \( \neg \text{ISFINAL}(\tilde{c}) \) and \( \neg \text{ISDEADLOCK}(\tilde{c}) \) (cf. Algorithm 6.8 and Lemma 6.5), which means that \( \tilde{c} \neq \emptyset \).

If \( \forall T \in \text{Thrd} : (\text{STM}(T, pc^T) = [\text{halt}]) \) \( \neq \emptyset \), then it is easy to see that \( \neg \text{ISFINAL}(\tilde{c}) \), \( \neg \text{ISDEADLOCK}(\tilde{c}) \) and \( \text{ISTIMEOUT}(\tilde{c}, I_\text{io}) \) (cf. Algorithm 6.9 and Lemma 6.6), which means that \( \tilde{c} \neq \emptyset \).

To finalize the proof, all possible terminating concrete transition sequences will be considered. Therefore, assume that \( \tilde{c} \) is not deadlocked. Since \( \forall [T, pc^T, \tilde{x}, \tilde{t}^a_T] \rangle \in \tilde{c} : \forall T \in \text{Thrd} : \text{STM}(T, pc^T) = [\text{halt}] \neq \emptyset \) (cf. Algorithm 6.7) and \( \neg \text{ISVALID}(\tilde{c}, I_\text{io}) \) only if \( \tilde{c} \) can never lead to a configuration that might have a valid concrete counterpart (Lemma 6.7), it is easy to see that all concrete executions of the configurations in \( C \) will terminate since all possible concrete transition sequences are safely approximated. Further assume that \( c \in C \) and \( c' \) such that \( c \xrightarrow{\text{prog}} \cdots \xrightarrow{\text{prog}} c' \wedge \forall T \in \text{Thrd} : \text{STM}(T, pc^T) = [\text{halt}] \). Since \( \tilde{c} \neq \emptyset \) and
all possible concrete transition sequences are safely approximated. Further
that might have a valid concrete counterpart (Lemma 6.7), it is easy to see

This concludes the proof.

NOTE. absExe has not been proven to terminate for all inputs. However,
when it does terminate, it safely approximates the transition se-
duences for the corresponding concrete input set.

One case for which absExe will not terminate is when some thread
could execute an infinite amount of statements in zero amount of time;
cf. an infinite loop where all the statements of the loop could be executed
without any progression of time.

6.2 Execution Time Analysis

The BCET and WCET of a program (Definition 6.9) given some initial set
of system states, are safely approximated by analysis, which is defined in
Algorithm 6.13, whenever it terminates (Theorem 6.11). (Note that for a con-
crete and final configuration, \( c@⟨[T, pc_T, c_T, 0_T]_{T∈ Thrd}, x, \bar{x}⟩ ∈ Conf \), the execu-
tion time of the program is \( \max(\{t_T | T ∈ Thrd_c\}) \), which is the same as the
BCET and WCET of the program if only considering this single configuration.)
The algorithm simply derives a safe approximation of the timing behavior of
the concrete collecting semantics of a given set of initial configurations using
absExe (defined in Algorithm 6.1 on page 165). Then the smallest BCET
approximation and the largest WCET approximation among the resulting con-
figurations are found. Note that choose was defined in Algorithm 6.2 on
page 167 and that it gives a deterministically chosen element from the con-
sidered set. Also note that the BCET and WCET approximations given by
a single final abstract configuration, \( \bar{c} ∈ Conf \), are defined in Definition 6.10. It
is straightforward to see that these approximations safely bound the concrete
program execution time given by any concrete configuration, \( c ∈ Conf \), such
that \( c ∈ γ_{conf}(\bar{c}) \).

Definition 6.9 (BCET and WCET):
The Best-Case Execution Time, BCET, and the Worst-Case Execution Time,
WCET, given a set of final concrete configurations, \( C ∈ \mathcal{P}(Conf) \) are defined
as:
\[
\begin{align*}
    BCET &= \min\{\max\{t_T^a \mid T \in \text{Thrd}_c\} \mid 
    c@([T, pc_T, \exists_T, t_T^a]_{T \in \text{Thrd}_c}, \varnothing, \varnothing) \in C\} \\
    WCET &= \max\{\min\{t_T^a \mid T \in \text{Thrd}_c\} \mid 
    c@([T, pc_T, \exists_T, t_T^a]_{T \in \text{Thrd}_c}, \varnothing, \varnothing) \in C\}
\end{align*}
\]

Definition 6.10 (BCET and WCET approximations):
The Best-Case Execution Time approximation, \(aBCET_{\tilde{c}}\), and the Worst-Case Execution Time approximation, \(aWCET_{\tilde{c}}\), of a single given final abstract configuration, \(\tilde{c}@([T, pc_T, \exists_T, t_T^a]_{T \in \text{Thrd}_c}, \tilde{x}, \tilde{l}) \in \text{Conf}\) are defined as:
\[
\begin{align*}
    aBCET_{\tilde{c}} &= \max\{\min\{\gamma_T(t_T^a) \mid T \in \text{Thrd}_c\}\} \\
    aWCET_{\tilde{c}} &= \max\{\max\{\gamma_T(t_T^a) \mid T \in \text{Thrd}_c\}\}
\end{align*}
\]

Algorithm 6.13 BCET/WCET analysis

1: function ANALYSIS(\(\tilde{C}, \tilde{t}_{to}\))
2: \((\tilde{C}^f, \tilde{C}^d, \tilde{C}'^d) \leftarrow \text{ABSEXEC}(\tilde{C}, \tilde{t}_{to})\)
3: if \(\tilde{C}^d \cup \tilde{C}'^d \neq \emptyset\) then
4: \(aBCET \leftarrow \min\{\min\{\gamma_T(t_T^a) \mid T \in \text{Thrd}\} \wedge \{(T, pc_T, \exists_T, t_T^a)_{T \in \text{Thrd}}, \tilde{x}, \tilde{l}\} \in \tilde{C}\}\)
5: return \((aBCET, \infty)\)
6: end if
7: \(aBCET \leftarrow \infty\)
8: \(aWCET \leftarrow -\infty\)
9: while \(\tilde{C}^f \neq \emptyset\) do
10: \(\tilde{c}@([T, pc_T, \exists_T, t_T^a]_{T \in \text{Thrd}}, \tilde{x}, \tilde{l}) \leftarrow \text{CHOOSE}(\tilde{C}^f)\)
11: \(\tilde{C}^f \leftarrow \tilde{C}^f \setminus \{\tilde{c}\}\)
12: \(aBCET_{\tilde{c}} \leftarrow \max\{\min\{\gamma_T(t_T^a) \mid T \in \text{Thrd}\}\}\)
13: \(aWCET_{\tilde{c}} \leftarrow \max\{\max\{\gamma_T(t_T^a) \mid T \in \text{Thrd}\}\}\)
14: if \(aBCET > aBCET_{\tilde{c}}\) then
15: \(aBCET \leftarrow aBCET_{\tilde{c}}\)
16: end if
17: if \(aWCET < aWCET_{\tilde{c}}\) then
18: \(aWCET \leftarrow aWCET_{\tilde{c}}\)
19: end if
20: end while
21: return \((aBCET, aWCET)\)
22: end function
Theorem 6.11 (Soundness of ANALYSIS):
If the sets of valid concrete configurations $C \in \mathcal{P}(\text{Conf})$ (cf. Definition 4.4) and abstract configurations $\tilde{C} \in \mathcal{P}(\text{Conf})$, are such that $\forall c \in C : \exists \tilde{c} : (\forall t \in [T, pc_T, rt_T, it_T]_{\text{Thr}} \in \text{Thr} : t \leq \text{aWCET})$ and $\forall \tilde{c} : \exists c : (\forall t \in [T, pc_T, rt_T, it_T]_{\text{Thr}} \in \text{Thr} : t \leq \text{aWCET})$, then given $\tilde{t}_0 \in \tilde{\text{Time}}$, $(\text{aBCET}, \text{aWCET}) \oplus \text{ANALYSIS}(\tilde{C}, \tilde{t}_0)$ is such that

$$
\forall c \in C : \forall c' @ (\forall t \in [T, pc_T, rt_T, it_T]_{\text{Thr}} \in \text{Thr} : t \leq \text{aWCET})
$$

given that the algorithm terminates.

Proof. Given $\tilde{t}_0 \in \tilde{\text{Time}}$, assume that the sets of valid concrete configurations $C \in \mathcal{P}(\text{Conf})$ (cf. Definition 4.4) and abstract configurations $\tilde{C} \in \mathcal{P}(\text{Conf})$ are as assumed in the theorem above and that $(\text{aBCET}, \text{aWCET}) = \text{ANALYSIS}(\tilde{C}, \tilde{t}_0)$.

Since $(\text{aBCET}, \text{aWCET}) = \text{ANALYSIS}(\tilde{C}, \tilde{t}_0)$, it must be that $(\tilde{C}, \tilde{C}' \oplus \tilde{C}^d) \oplus$
Abbreviation (Algorithm 6.1) terminates at some point and that

\[ \forall c \in C : \forall c' @ ([T, pc_T'], x'_T, T])_{T \in Thrd}, x', [\beta] \in Conf : \]

\[ ((c \xrightarrow{\text{prog}} \ldots \xrightarrow{\text{prog}} c') \land \forall T \in Thrd : STM(T, pc_T') = [\text{halt}]^{pc_T'}) \Rightarrow \]

\[ (\tilde{C'} \neq \emptyset \lor \exists c @ ([T, pc_T', x'_T, T])_{T \in Thrd}, x, [\beta] \in \tilde{C}' : \forall T \in Thrd : \]

\[ (pc_T' = pc_T' \land r''_T \in \gamma_T([\beta])) \land \]

\[ \forall c \in C : \forall c' @ ([T, pc_T', x'_T, T])_{T \in Thrd}, x', [\beta] \in Conf : \]

\[ ((c \xrightarrow{\text{prog}} \ldots \xrightarrow{\text{prog}} c') \land (\text{CYCLE}(\text{Thrd}^c_\text{lock}, E^{c'})) \lor \]

\[ \exists T \in Thrd : \exists lck \in Lck : \]

\[ (STM(T, pc_T') = [\text{lock} lck]^{pc_T'} \land \]

\[ \text{OWN}([\beta] lck) \notin \{\text{T-thrd}, T\} \land \]

\[ STM(\text{OWN}([\beta] lck), pc_{\text{OWN}}([\beta] lck)) = \]

\[ [\text{halt}]^{pc_{\text{OWN}}([\beta] lck)}) \Rightarrow \]

\[ (\tilde{C'} \neq \emptyset \lor \tilde{C}' \neq \emptyset) \]

(6.8.1). It is thus apparent that if \( \tilde{C}' \cup \tilde{C}' \neq \emptyset \) (deadlocked and/or timed-out configurations have been derived), there might exist an infinite transition sequence in the concrete case. However, it is easy to see that \( \min\{\min(\gamma_T([\beta])) | T' \in Thrd \land ([T, pc_T, x'_T, T])_{T \in Thrd}, \tilde{x}, [\beta] \in \tilde{C} \} \) is a safe approximation of the BCET since time only moves forward (Lemma 4.2 and Assumption 5.51) and that \( \infty \) is a safe approximation of the WCET for all such (and all other) cases.

If \( \tilde{C}' \cup \tilde{C}' = \emptyset \) (no deadlocked and no timed-out configurations have been derived), then all concrete transition sequences are of finite length and \( \forall c \in C : \forall c' @ ([T, pc_T', x'_T, T])_{T \in Thrd}, x', [\beta] \in Conf : ((c \xrightarrow{\text{prog}} \ldots \xrightarrow{\text{prog}} c') \land \forall T \in Thrd : STM(T, pc_T') = [\text{halt}]^{pc_T'} \Rightarrow \exists ([T, pc_T', x'_T, T])_{T \in Thrd}, \tilde{x}, [\beta] \in \tilde{C}' : (pc_T = pc_T' \land r''_T \in \gamma_T([\beta])) \) (the execution times of all derivable final concrete configurations, given the initial configurations, are safely approximated) (Theorem 6.8.1). Thus, since the structure of the algorithm trivially gives that the smallest possible estimation of the execution time, aBCET, and the largest possible estimation of the execution time, aWCET, among the derived final abstract configurations in \( \tilde{C}' \) are found (cf. Definition 6.10), it must be that \( \forall c \in C : \forall c' @ ([T, pc_T', x'_T, T])_{T \in Thrd}, x', [\beta] \in Conf : ((c \xrightarrow{\text{prog}} \ldots \xrightarrow{\text{prog}} c') \land \forall T \in Thrd : STM(T, pc_T') = [\text{halt}]^{pc_T'} \Rightarrow \forall T \in Thrd : aBCET \leq r''_T \leq aWCET \) (the execution time of each final concrete configuration, given the initial configurations, is safely bounded). But, then it must also
be that $\forall c \in C : \forall c' @ ([T, p c'_T, z'_T, t'^e_T]_{T \in \text{Thrd}}, x'_T, y'_T) \in \text{Conf} : (c \rightarrow \ldots \rightarrow c') \Rightarrow \forall T \in \text{Thrd} : t'^d_T \leq a \text{WCET}$ (the execution time of any derivable concrete configuration, given the initial configurations, is safely upper bounded) since time only moves forward (Lemma 4.2 and Assumption 5.51), which concludes the proof.

\[\square\]
Chapter 7

Examples

To clarify and explain the analysis defined in Chapters 5 and 6, this chapter instantiates it for some example PPL programs and timing models.

7.1 Communication

This case shows the recursive behavior of ABS EXE; i.e., how it peeks into the future to derive safe write histories for load-statements acting on global variables.

For the program, $\text{Thrd} = \{T1, T2, T3\}$, defined in Table 7.1, it is easy to see that $\text{Reg}_{T1} = \{r\}$, $\text{Reg}_{T2} = \{r\}$, $\text{Reg}_{T3} = \{r\}$, $\text{Var} = \{x, y, z\}$ and $\text{Lck} = /0$.

Table 7.1: Communication – Program.

<table>
<thead>
<tr>
<th>Thread</th>
<th>Program</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{Thrd}_1$</td>
<td>$(T1, \text{load } r \text{ from } x, \text{store } r \text{ to } y, \text{halt})$</td>
</tr>
<tr>
<td>$\text{Thrd}_2$</td>
<td>$(T2, \text{load } r \text{ from } y, \text{store } r \text{ to } z, \text{halt})$</td>
</tr>
<tr>
<td>$\text{Thrd}_3$</td>
<td>$(T3, \text{if } r \leq 3 \text{ goto } 4, \text{store } r \text{ to } x, \text{skip}, \text{halt})$</td>
</tr>
</tbody>
</table>
Chapter 7

Examples

To clarify and explain the analysis defined in Chapters 5 and 6, this chapter instantiates it for some example PPL programs and timing models.

7.1 Communication

This case shows the recursive behavior of ABSEXE; i.e., how it peeks into the future to derive safe write histories for load-statements acting on global variables.

For the program, \textbf{Thrd} = \{T_1, T_2, T_3\}, defined in Table 7.1, it is easy to see that \textbf{Reg}_{T_1} = \{r\}, \textbf{Reg}_{T_2} = \{r\}, \textbf{Reg}_{T_3} = \{r\}, \textbf{Var} = \{x, y, z\} and \textbf{Lck} = \emptyset. Note that \(r\) represents local memory within each thread; i.e., the register-name \(r\) can refer to three different memory locations—which location it refers to

| T_1@T_1, [load r from x]^1; [store r to y]^2; [halt]^3 |
| T_2@T_2, [load r from y]^1; [store r to z]^2; [halt]^3 |
| T_3@T_3, [if r <= 3 goto 4]^1; [store r to x]^2; [skip]^3; [halt]^4 |

Table 7.1: Communication – Program.
Table 7.2: Communication – Timing model.

<table>
<thead>
<tr>
<th>$pc_T$ (T ∈ Thrd)</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{ABSTIME}(\tilde{c}, T_1):$</td>
<td>$[1, 5]$</td>
<td>$[1, 3]$</td>
<td>$-$</td>
</tr>
<tr>
<td>$\text{ABSTIME}(\tilde{c}, T_2):$</td>
<td>$[2, 6]$</td>
<td>$[2, 3]$</td>
<td>$-$</td>
</tr>
<tr>
<td>$\text{ABSTIME}(\tilde{c}, T_3):$</td>
<td>$[1, 4]$</td>
<td>$[3, 4]$</td>
<td>$[3, 3]$</td>
</tr>
</tbody>
</table>

depends on which thread is considered.

Assume that $\text{ABSTIME}(\tilde{c}, T)$, i.e., the abstracted timing model, where $\tilde{c} @ \langle [T, pc_T, \tilde{T}_T, \bar{T}_T]_{T \in \text{Thrd}_\tilde{c}}, \tilde{x}, \tilde{y} \rangle \in \text{Conf}$ and $T \in \text{Thrd}_\tilde{c}$, is such that for any $\tilde{c}$, it assumes the values described by Table 7.2. A ‘$-$’ indicates that the entry is not applicable to the considered thread.

Also assume that the initial configuration, $\tilde{c}_0^0 @ \langle [T, pc_T, \tilde{T}_T, \bar{T}_T]_{T \in \text{Thrd}_\tilde{c}}, \tilde{x}, \tilde{y} \rangle$, is as described in Table 7.3. (Due to the semantics of the program, the parts of the states that are left out from the table are of no interest for this case study.)

Tables 7.3 and 7.4 collect all the configurations derived by $\text{ABSEXET}(\{\tilde{c}_0^0\}, [-\infty, \infty])$ during the analysis described by $\text{ANALYSIS}(\{\tilde{c}_0^0\}, [-\infty, \infty])$. A ‘$-$’ indicates that the entry is not applicable to (i.e., not included in) the configuration. Figure 7.5 shows the relation between the derived configurations. In the figure, final configurations are circled and timed-out configurations are circled and marked with a ‘$'$'. To see how new recursive instances of $\text{ABSEXET}$ are created, note that when $\text{Thrd}_\tilde{c} = \{T_1, T_2, T_3\}$, then $\text{Var}_g = \{x, y\}$; when $\text{Thrd}_\tilde{c} = \{T_1, T_3\}$, then $\text{Var}_g = \{x\}$; and when $\text{Thrd}_\tilde{c} = \{T_2, T_3\}$, then $\text{Var}_g = \emptyset$.

It is apparent that $\text{ABSEXET}(\{\tilde{c}_0^0\}, [-\infty, \infty]) = (\{\tilde{c}_{11}^0, \tilde{c}_{23}^0\}, 0, 0)$; i.e., $\tilde{c}_{11}^0$ and $\tilde{c}_{23}^0$ are final-state configurations and there are no deadlocked or timed-out configurations. Note that $\tilde{c}_{12}^1$, $\tilde{c}_{11}^2$, $\tilde{c}_{22}^1$, $\tilde{c}_{11}^2$ and $\tilde{c}_{22}^2$ only exist within the recursively called $\text{ABSEXET}$-instances. According to Algorithm 6.13, it is thus easy to see that the estimated timing bounds are:

\[
\begin{align*}
\text{aBCET} &= \min(\{\max(\{\min(\gamma_i(\bar{T}_T^i)) | T \in \text{Thrd}\}) | \langle [T, pc_T, \tilde{T}_T, \bar{T}_T]_{T \in \text{Thrd}_\tilde{c}}, \tilde{x}, \tilde{y} \rangle \in \{\tilde{c}_{11}^0, \tilde{c}_{23}^0\}\}) = 4 \\
\text{aWCET} &= \max(\{\max(\{\min(\gamma_i(\bar{T}_T^i)) | T \in \text{Thrd}\}) | \langle [T, pc_T, \tilde{T}_T, \bar{T}_T]_{T \in \text{Thrd}_\tilde{c}}, \tilde{x}, \tilde{y} \rangle \in \{\tilde{c}_{11}^0, \tilde{c}_{23}^0\}\}) = 11
\end{align*}
\]
Table 7.3: Communication – Configurations (first half).

<table>
<thead>
<tr>
<th>$\tilde{c}$</th>
<th>$pc_{T_1}$</th>
<th>$pc_{T_2}$</th>
<th>$pc_{T_3}$</th>
<th>$\tilde{x}_{T_1}$</th>
<th>$\tilde{x}_{T_2}$</th>
<th>$\tilde{x}_{T_3}$</th>
<th>$\tilde{a}_{T_1}$</th>
<th>$\tilde{a}_{T_2}$</th>
<th>$\tilde{a}_{T_3}$</th>
<th>$(\tilde{x}, x)$ $T_3$</th>
<th>$(\tilde{x}, y)$ $T_1$</th>
<th>$(\tilde{x}, z)$ $T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c^0_{11}$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>[0, 0]</td>
<td>[0, 0]</td>
<td>[2, 4]</td>
<td>[0, 0]</td>
<td>[0, 0]</td>
<td>[0, 0]</td>
<td>${(1, 1), [0, 0]}$</td>
<td>${(5, 5), [0, 0]}$</td>
<td>${(1, 1), [1, 1]}$</td>
</tr>
<tr>
<td>$c^0_{10}$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>[0, 0]</td>
<td>[2, 4]</td>
<td>[0, 0]</td>
<td>[0, 0]</td>
<td>[0, 0]</td>
<td>[0, 0]</td>
<td>${(1, 1), [0, 0]}$</td>
<td>${(5, 5), [0, 0]}$</td>
<td>${(1, 1), [1, 1]}$</td>
</tr>
<tr>
<td>$c^0_{11}$</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>[5, 5]</td>
<td>[2, 3]</td>
<td>[2, 6]</td>
<td>[1, 4]</td>
<td>[0, 0]</td>
<td>[0, 0]</td>
<td>${(1, 1), [0, 0]}$</td>
<td>${(5, 5), [0, 0]}$</td>
<td>${(1, 1), [1, 1]}$</td>
</tr>
<tr>
<td>$c^0_{11}$</td>
<td>3</td>
<td>4</td>
<td>2</td>
<td>[5, 5]</td>
<td>[2, 3]</td>
<td>[4, 9]</td>
<td>[1, 4]</td>
<td>[0, 0]</td>
<td>[0, 0]</td>
<td>${(1, 1), [0, 0]}$</td>
<td>${(5, 5), [0, 0]}$</td>
<td>${(1, 1), [1, 1]}$</td>
</tr>
<tr>
<td>$c^0_{12}$</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>[5, 5]</td>
<td>[4, 4]</td>
<td>[2, 6]</td>
<td>[1, 4]</td>
<td>[0, 0]</td>
<td>[0, 0]</td>
<td>${(1, 1), [0, 0]}$</td>
<td>${(5, 5), [0, 0]}$</td>
<td>${(1, 1), [1, 1]}$</td>
</tr>
<tr>
<td>$c^0_{12}$</td>
<td>3</td>
<td>4</td>
<td>2</td>
<td>[5, 5]</td>
<td>[2, 3]</td>
<td>[4, 9]</td>
<td>[1, 4]</td>
<td>[0, 0]</td>
<td>[0, 0]</td>
<td>${(1, 1), [0, 0]}$</td>
<td>${(5, 5), [0, 0]}$</td>
<td>${(1, 1), [1, 1]}$</td>
</tr>
<tr>
<td>$c^0_{12}$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>[0, 0]</td>
<td>[2, 4]</td>
<td>[0, 0]</td>
<td>[0, 0]</td>
<td>[0, 0]</td>
<td>[0, 0]</td>
<td>${(1, 1), [0, 0]}$</td>
<td>${(5, 5), [0, 0]}$</td>
<td>${(1, 1), [1, 1]}$</td>
</tr>
</tbody>
</table>
Table 7.4: Communication – Configurations (second half).

<table>
<thead>
<tr>
<th>( \bar{c} )</th>
<th>( pc_{T_1} )</th>
<th>( pc_{T_2} )</th>
<th>( pc_{T_3} )</th>
<th>( \bar{\tau}_{T_1} )</th>
<th>( \bar{\tau}_{T_2} )</th>
<th>( \bar{\tau}_{T_3} )</th>
<th>( \bar{\tau}_{T_1}^a )</th>
<th>( \bar{\tau}_{T_2}^a )</th>
<th>( \bar{\tau}_{T_3}^a )</th>
<th>( (\bar{\tau} _x) _T_3 )</th>
<th>( (\bar{\tau} _y) _T_1 )</th>
<th>( (\bar{\tau} _z) _T_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \bar{c}_{1}^{12} )</td>
<td>2</td>
<td>–</td>
<td>1</td>
<td>[1, 4]</td>
<td>–</td>
<td>[2, 4]</td>
<td>[1, 5]</td>
<td>–</td>
<td>[0, 0]</td>
<td>{[1, 1], [0, 0]}</td>
<td>{[5, 5], [0, 0]}</td>
<td>{\bar{I}_{val}, \bar{I}_r}</td>
</tr>
<tr>
<td>( \bar{c}_{11}^{12} )</td>
<td>3</td>
<td>–</td>
<td>4</td>
<td>[1, 4]</td>
<td>–</td>
<td>[2, 3]</td>
<td>[2, 8]</td>
<td>–</td>
<td>[1, 4]</td>
<td>{[1, 1], [0, 0]}</td>
<td>{[5, 5], [0, 0]}</td>
<td>{\bar{I}_{val}, \bar{I}_r}</td>
</tr>
<tr>
<td>( \bar{c}_{21}^{12} )</td>
<td>3</td>
<td>–</td>
<td>2</td>
<td>[1, 4]</td>
<td>–</td>
<td>[4, 4]</td>
<td>[2, 8]</td>
<td>–</td>
<td>[1, 4]</td>
<td>{[1, 1], [0, 0]}</td>
<td>{[5, 5], [0, 0]}</td>
<td>{\bar{I}_{val}, \bar{I}_r}</td>
</tr>
<tr>
<td>( \bar{c}_{22}^{12} )</td>
<td>3</td>
<td>–</td>
<td>3</td>
<td>[1, 4]</td>
<td>–</td>
<td>[4, 4]</td>
<td>[2, 8]</td>
<td>–</td>
<td>[4, 8]</td>
<td>{[1, 1], [0, 0]}, {[4, 4], [4, 8]}</td>
<td>{[5, 5], [0, 0]}, {[1, 4], [2, 8]}</td>
<td>{\bar{I}_{val}, \bar{I}_r}</td>
</tr>
<tr>
<td>( \bar{c}_{1}^{0} )</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>[1, 4]</td>
<td>[1, 5]</td>
<td>[2, 4]</td>
<td>[1, 5]</td>
<td>[2, 6]</td>
<td>[0, 0]</td>
<td>{[1, 1], [0, 0]}</td>
<td>{[5, 5], [0, 0]}</td>
<td>{\bar{I}_{val}, \bar{I}_r}</td>
</tr>
<tr>
<td>( \bar{c}_{11}^{0} )</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>[1, 4]</td>
<td>[1, 5]</td>
<td>[2, 3]</td>
<td>[2, 8]</td>
<td>[4, 9]</td>
<td>[1, 4]</td>
<td>{[1, 1], [0, 0]}</td>
<td>{[5, 5], [0, 0]}, {[1, 4], [2, 8]}</td>
<td>{\bar{I}_{val}, \bar{I}_r}, {[1, 5], [4, 9]}</td>
</tr>
<tr>
<td>( \bar{c}_{21}^{0} )</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>[1, 4]</td>
<td>[1, 5]</td>
<td>[4, 4]</td>
<td>[2, 8]</td>
<td>[4, 9]</td>
<td>[1, 4]</td>
<td>{[1, 1], [0, 0]}</td>
<td>{[5, 5], [0, 0]}, {[1, 4], [2, 8]}</td>
<td>{\bar{I}_{val}, \bar{I}_r}, {[1, 5], [4, 9]}</td>
</tr>
<tr>
<td>( \bar{c}_{22}^{0} )</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>[1, 4]</td>
<td>[1, 5]</td>
<td>[4, 4]</td>
<td>[2, 8]</td>
<td>[4, 9]</td>
<td>[4, 8]</td>
<td>{[1, 1], [0, 0]}, {[4, 4], [4, 8]}</td>
<td>{[5, 5], [0, 0]}, {[1, 4], [2, 8]}</td>
<td>{\bar{I}_{val}, \bar{I}_r}, {[1, 5], [4, 9]}</td>
</tr>
<tr>
<td>( \bar{c}_{23}^{0} )</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>[1, 4]</td>
<td>[1, 5]</td>
<td>[4, 4]</td>
<td>[2, 8]</td>
<td>[4, 9]</td>
<td>[7, 11]</td>
<td>{[1, 1], [0, 0]}, {[4, 4], [4, 8]}</td>
<td>{[5, 5], [0, 0]}, {[1, 4], [2, 8]}</td>
<td>{\bar{I}_{val}, \bar{I}_r}, {[1, 5], [4, 9]}</td>
</tr>
</tbody>
</table>
Figure 7.5: Communication – Configuration relations.
Table 7.6: Synchronization (Deadlock) – Program.

\[
\begin{align*}
T_1 & @ (T_1, [\text{lock la}]^1; [\text{lock lb}]^2; [\text{unlock la}]^3; [\text{unlock lb}]^4; [\text{halt}]^5) \\
T_2 & @ (T_2, [\text{lock la}]^1; [\text{lock lb}]^2; [\text{halt}]^3)
\end{align*}
\]

Table 7.7: Synchronization (Deadlock) – Timing model.

<table>
<thead>
<tr>
<th>pc(T) ((T \in \text{Thrd}))</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABSTIME((\tilde{c}, T_1)) : ([2, 2])</td>
<td>[1, 2]</td>
<td>[1, 1]</td>
<td>[1, 1]</td>
<td></td>
</tr>
<tr>
<td>ABSTIME((\tilde{c}, T_2)) : [1, 2]</td>
<td>[1, 2]</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
</tbody>
</table>

7.2 Synchronization – Deadlock

This case shows how \textsc{absexe} identifies deadlocked configurations and how it discontinues deadlocked configurations that lack concrete counterparts.

For the program, \(\text{Thrd} = \{T_1, T_2\}\), defined in Table 7.6, it is easy to see that \(\text{Reg}_{T_1} = \emptyset\), \(\text{Reg}_{T_2} = \emptyset\), \(\text{Var} = \emptyset\) and \(\text{Lck} = \{\text{la}, \text{lb}\}\).

Assume that the abstracted timing model, \(\text{ABSTIME}(\tilde{c}, T)\), where \(\tilde{c} \in \text{Reg} \cap \text{Dom}(\text{Reg})\), is such that for any \(\tilde{c}\), it assumes the values described by Table 7.7. A ‘-’ indicates that the entry is not applicable to the considered thread.

Also assume that the initial configuration, \(\tilde{c}_0^0 @ \langle [T, pc_T, \tilde{\tilde{\tau}}_T, \tilde{\tilde{\tau}}_T]_{T \in \text{Thrd}}; \tilde{x}, \tilde{\tilde{\ell}} \rangle \in \text{Conf}\) and \(T \in \text{Thrd}\), is such that for any \(\tilde{c}\), it assumes the values described by Table 7.7. Figure 7.9 shows the relation between the derived configurations. In the figure, final configurations are circled, deadlocked configurations are circled and marked with a ‘\(d\)’ and discontinued configurations are crossed out. Note that \(\tilde{c}_2^4\) occurs since \(T_2\) has been waiting to acquire la and is now assigned it; \(T_2\)’s accumulated abstract execution time is updated to account for the concrete spin-waiting (cf. the proof of Lemma 5.58).

It is apparent that \(\text{ABSEXE}(\{\tilde{c}_0^0\}, [\infty, \infty]) = (\{\tilde{c}_2^2\}, \{\tilde{c}_4^2\}, \emptyset)\); i.e., \(\tilde{c}_2^2\) is a final-state configuration, \(\tilde{c}_4^2\) is a deadlocked configuration, and there are no timed-out configurations.
Table 7.8: Synchronization (Deadlock) – Configurations.

<table>
<thead>
<tr>
<th>$\tilde{c}$</th>
<th>$pc_{T_1}$</th>
<th>$pc_{T_2}$</th>
<th>$\tilde{r}_{T_1}^a$</th>
<th>$\tilde{r}_{T_2}^a$</th>
<th>$\tilde{l}a$</th>
<th>$\tilde{lb}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\tilde{c}_0$</td>
<td>1</td>
<td>1</td>
<td>[0,0]</td>
<td>[0,0]</td>
<td>(unlocked, $\bot_{thrd}$, $\bot_T$, $\bot_{thrd}$)</td>
<td>(unlocked, $\bot_{thrd}$, $\bot_T$, $\bot_{thrd}$)</td>
</tr>
<tr>
<td>$\tilde{c}_1$</td>
<td>2</td>
<td>1</td>
<td>[2,2]</td>
<td>[0,0]</td>
<td>(locked, $T_1$, $[-\infty, 2]$, $\bot_{thrd}$, $\bot_T$)</td>
<td>(unlocked, $\bot_{thrd}$, $\bot_T$, $\bot_{thrd}$)</td>
</tr>
<tr>
<td>$\tilde{c}_2$</td>
<td>2</td>
<td>1</td>
<td>[2,2]</td>
<td>[0,0]</td>
<td>(locked, $T_1$, $[-\infty, 2]$, $\bot_{thrd}$, $\bot_T$)</td>
<td>(unlocked, $T_2$, $[-\infty, 4]$, $\bot_{thrd}$, $\bot_T$)</td>
</tr>
<tr>
<td>$\tilde{c}_3$</td>
<td>3</td>
<td>1</td>
<td>[3,4]</td>
<td>[0,0]</td>
<td>(locked, $T_1$, $[-\infty, 2]$, $\bot_{thrd}$, $\bot_T$)</td>
<td>(locked, $T_1$, $[-\infty, 4]$, $\bot_{thrd}$, $\bot_T$)</td>
</tr>
<tr>
<td>$\tilde{c}_4$</td>
<td>4</td>
<td>1</td>
<td>[4,5]</td>
<td>[0,0]</td>
<td>(unlocked, $\bot_{thrd}$, $[-\infty, 2]$, $T_1$, $[4,5]$)</td>
<td>(locked, $T_1$, $[-\infty, 4]$, $\bot_{thrd}$, $\bot_T$)</td>
</tr>
<tr>
<td>$\tilde{c}_5$</td>
<td>5</td>
<td>1</td>
<td>[5,6]</td>
<td>[0,0]</td>
<td>(unlocked, $T_1$, $[-\infty, 12]$, $T_1$, $[4,5]$)</td>
<td>(unlocked, $\bot_{thrd}$, $[-\infty, 4]$, $T_1$, $[5,6]$)</td>
</tr>
<tr>
<td>$\tilde{c}_6$</td>
<td>4</td>
<td>1</td>
<td>[4,5]</td>
<td>[1,2]</td>
<td>(unlocked, $T_2$, $[-\infty, 12]$, $T_1$, $[4,5]$)</td>
<td>(locked, $T_1$, $[-\infty, 4]$, $\bot_{thrd}$, $\bot_T$)</td>
</tr>
<tr>
<td>$\tilde{c}_7$</td>
<td>4</td>
<td>2</td>
<td>[4,5]</td>
<td>[4,12]</td>
<td>(locked, $T_2$, $[-\infty, 12]$, $T_1$, $[4,5]$)</td>
<td>(locked, $T_1$, $[-\infty, 4]$, $\bot_{thrd}$, $\bot_T$)</td>
</tr>
<tr>
<td>$\tilde{c}_8$</td>
<td>5</td>
<td>2</td>
<td>[5,6]</td>
<td>[4,12]</td>
<td>(locked, $T_2$, $[-\infty, 12]$, $T_1$, $[4,5]$)</td>
<td>(unlocked, $\bot_{thrd}$, $[-\infty, 4]$, $T_1$, $[5,6]$)</td>
</tr>
<tr>
<td>$\tilde{c}_9$</td>
<td>5</td>
<td>2</td>
<td>[5,6]</td>
<td>[4,12]</td>
<td>(locked, $T_2$, $[-\infty, 12]$, $T_1$, $[4,5]$)</td>
<td>(unlocked, $T_1$, $[-\infty, 18]$, $T_1$, $[5,6]$)</td>
</tr>
<tr>
<td>$\tilde{c}_{10}$</td>
<td>5</td>
<td>3</td>
<td>[5,6]</td>
<td>[5,18]</td>
<td>(locked, $T_2$, $[-\infty, 12]$, $T_1$, $[4,5]$)</td>
<td>(locked, $T_2$, $[-\infty, 18]$, $T_1$, $[5,6]$)</td>
</tr>
<tr>
<td>$\tilde{c}_{11}$</td>
<td>1</td>
<td>2</td>
<td>[0,0]</td>
<td>[1,2]</td>
<td>(locked, $T_2$, $[-\infty, 2]$, $\bot_{thrd}$, $\bot_T$)</td>
<td>(unlocked, $\bot_{thrd}$, $\bot_T$, $\bot_{thrd}$)</td>
</tr>
<tr>
<td>$\tilde{c}_{12}$</td>
<td>1</td>
<td>2</td>
<td>[0,0]</td>
<td>[1,2]</td>
<td>(locked, $T_2$, $[-\infty, 2]$, $\bot_{thrd}$, $\bot_T$)</td>
<td>(unlocked, $T_1$, $[-\infty, 4]$, $\bot_{thrd}$, $\bot_T$)</td>
</tr>
<tr>
<td>$\tilde{c}_{13}$</td>
<td>1</td>
<td>3</td>
<td>[0,0]</td>
<td>[2,4]</td>
<td>(locked, $T_2$, $[-\infty, 2]$, $\bot_{thrd}$, $\bot_T$)</td>
<td>(locked, $T_2$, $[-\infty, 4]$, $\bot_{thrd}$, $\bot_T$)</td>
</tr>
</tbody>
</table>
According to Algorithm 6.13, it is thus easy to see that the estimated timing bounds are:

\[
\begin{align*}
\text{aBCET} &= 0 \\
\text{aWCET} &= \infty
\end{align*}
\]

This case illustrates how the analysis discontinues configurations for which an assigned lock owner does not acquire the lock in time. It also illustrates how the analysis detects deadlocks.

For the program, \( \text{Thrd} = \{T_1, T_2\} \), defined in Table 7.10, it is easy to see that

\[
\begin{align*}
\text{Reg}_{T_1} &= 0 \\
\text{Reg}_{T_2} &= 0 \\
\text{Var} &= 0 \\
\text{Lck} &= \{l\}
\end{align*}
\]

Assume that \( \text{ABS \ TIME}(c_{0}, T) \), i.e., the abstracted timing model, where \( \tilde{c} \in \tilde{\text{confs}} \) and \( T \in \text{Thrd} \), is such that for any \( \tilde{c} \), it assumes the values described by Table 7.11.

Also assume that the initial configuration, \( \tilde{c}_{0} \), is as described in Table 7.12. (Due to the semantics of the program, the parts of the states that are left out from the table are of no interest for this case study.)

Table 7.12 collects all the configurations derived by \( \text{ABS \ EXE}(\{c_{0}\}, [-\infty, \infty]) \) during the analysis described by \( \text{ANALYSIS}(\{c_{0}\}, [-\infty, \infty]) \). Figure 7.9 shows the relation between the derived configurations. In the figure, deadlocked con-
7.3 Synchronization – Deadline Miss

Table 7.10: Synchronization (Deadline miss) – Program.

<table>
<thead>
<tr>
<th>T1 @(T1,[lock l]1;[halt]2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>T2 @(T2,[lock l]1;[halt]2)</td>
</tr>
</tbody>
</table>

Table 7.11: Synchronization (Deadline miss) – Timing model.

<table>
<thead>
<tr>
<th>pcT (T ∈ Thrd)</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABSTIME(∅,T1)</td>
<td>[5, 5]</td>
</tr>
<tr>
<td>ABSTIME(∅,T2)</td>
<td>[10, 10]</td>
</tr>
</tbody>
</table>

According to Algorithm 6.13, it is thus easy to see that the estimated timing bounds are:

\[
\begin{align*}
& aBCET = 0 \\
& aWCET = \infty
\end{align*}
\]

7.3 Synchronization – Deadline Miss

This case illustrates how the analysis discontinues configurations for which an assigned lock owner does not acquire the lock in time. It also illustrates how the analysis detects deadlocks.

For the program, Thrd = \{T1, T2\}, defined in Table 7.10, it is easy to see that RegT1 = ∅, RegT2 = ∅, Var = ∅ and Lck = \{l\}.

Assume that ABSTIME(∅, T), i.e., the abstracted timing model, where \(\hat{\ell}@\langle [T,pcT,\tilde{\varepsilon}_T,\tilde{l}^{|T|}_T]_{T∈Thrd};\tilde{x},\tilde{y}\rangle \in Conf\) and \(T ∈ Thrd\), is such that for any \(\hat{\ell}\), it assumes the values described by Table 7.11.

Also assume that the initial configuration, \(\hat{c}_0@\langle [T,pcT,\tilde{\varepsilon}_T,\tilde{l}^{|T|}_T]_{T∈Thrd};\tilde{x},\tilde{y}\rangle\), is as described in Table 7.12. (Due to the semantics of the program, the parts of the states that are left out from the table are of no interest for this case study.)

Table 7.12 collects all the configurations derived by ABSEXE(\{c0\}, [−∞, ∞]) during the analysis described by ANALYSIS(\{c0\}, [−∞, ∞]). Figure 7.13 shows the relation between the derived configurations. In the figure, deadlocked con-
Table 7.12: Synchronization (Deadline miss) – Configurations.

<table>
<thead>
<tr>
<th>( \tilde{c} )</th>
<th>( pc_{T_1} )</th>
<th>( pc_{T_2} )</th>
<th>( \tilde{r}^c_{T_1} )</th>
<th>( \tilde{r}^c_{T_2} )</th>
<th>( \hat{l} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \tilde{c}_0 )</td>
<td>1</td>
<td>1</td>
<td>[0, 0]</td>
<td>[0, 0]</td>
<td>(unlocked, ( \perp_{thrd} ), ( \perp_{thrd} ), ( \perp_t ))</td>
</tr>
<tr>
<td>( \tilde{c}_1 )</td>
<td>2</td>
<td>1</td>
<td>[5, 5]</td>
<td>[0, 0]</td>
<td>(locked, ( T_1 ), [( -\infty, 5 )], ( \perp_{thrd} ), ( \perp_t ))</td>
</tr>
<tr>
<td>( \tilde{c}_2 )</td>
<td>1</td>
<td>1</td>
<td>[0, 0]</td>
<td>[10, 10]</td>
<td>(unlocked, ( T_2 ), [( -\infty, 5 )], ( \perp_{thrd} ), ( \perp_t ))</td>
</tr>
</tbody>
</table>

\[
\text{ABSExe}(\{\tilde{c}_0\}, [\infty, \infty])
\]

\[
\xymatrix{ \tilde{c}_0 \\
\vdots \\
\tilde{c}_1 \quad \tilde{c}_2 }
\]

Figure 7.13: Synchronization (Deadline miss) – Configuration relations.

Figurations are circled and marked with a ‘d’ and discontinued configurations are crossed out. It is apparent that \( \text{ABSExe}(\{\tilde{c}_0\}, [\infty, \infty]) = (\emptyset, \{\tilde{c}_1\}, \emptyset) \); i.e., there are no final-state or timed-out configurations, and \( \tilde{c}_1 \) is a deadlocked configuration. According to Algorithm 6.13, it is thus easy to see that the estimated timing bounds are:

\[
\begin{align*}
\text{aBCET} &= 0 \\
\text{aWCET} &= \infty
\end{align*}
\]

### 7.4 Data Parallel Loop

The purpose of the program in Table 7.14 is to increment the value of the variable \( x \) with \( \sum_{i=1}^{4} (2i + 3) \). The task of calculating the sum is equally divided onto two threads, \( T_1 \) and \( T_2 \). It is easy to see that \( \text{Thrd} = \{T_1, T_2\} \), \( \text{Reg}_{T_1} = \{p, r\} \), \( \text{Reg}_{T_2} = \{p, r\} \), \( \text{Var} = \{x\} \) and \( \text{Lck} = \{1\} \). Note that \( p \) (and \( r \)) represent local memory within each thread; i.e., the register-name \( p \) (and \( r \)) can refer to two different memory locations – what location it refers to depends...
The purpose of the program in Table 7.14 is to increment the value of the variable \(x\).

Table 7.14: Data parallel loop – Program.

\[
\begin{align*}
T_1 &: (T1, \{ p := p + 1 \}) \quad ; [r := r + 2 \times p + 3 \}; [\text{if } p < 2 \text{ goto } 1 \}; [\text{lock } l \}; [\text{load } p \text{ from } x \}; [p := p + r \}; [\text{store } p \text{ to } x \}; [\text{unlock } l \}; [\text{halt}]) \\
T_2 &: (T2, \{ p := p + 1 \}) \quad ; [r := r + 2 \times p + 3 \}; [\text{if } p < 4 \text{ goto } 1 \}; [\text{lock } l \}; [\text{load } p \text{ from } x \}; [p := p + r \}; [\text{store } p \text{ to } x \}; [\text{unlock } l \}; [\text{halt}])
\end{align*}
\]

Table 7.15: Data parallel loop – Timing model.

<table>
<thead>
<tr>
<th>(pc_T (T \in \text{Thrd}))</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{ABSTIME}(\tilde{c}, T_1)</td>
<td>2, 2</td>
<td>1, 1</td>
<td>1, 2</td>
<td>1, 2</td>
<td>2, 3</td>
<td>1, 1</td>
<td>2, 3</td>
<td>2, 3</td>
</tr>
<tr>
<td>\text{ABSTIME}(\tilde{c}, T_2)</td>
<td>2, 2</td>
<td>1, 1</td>
<td>4, 5</td>
<td>5, 6</td>
<td>2, 5</td>
<td>2, 2</td>
<td>2, 4</td>
<td>2, 3</td>
</tr>
</tbody>
</table>

on which thread is considered. It is easy to see that \(x\) is a global variable when \(\text{Thrd}_c = \{T_1, T_2\}\) and that there are no global variables when \(\text{Thrd}_c = \{T_1\}\) or \(\text{Thrd}_c = \{T_2\}\).

For the sake of simplicity, the timing model (i.e., \text{ABSTIME}) as described in Table 7.15 gives that each statement within a thread has constant timing bounds.

Assume that the initial configuration, \(\tilde{c}_0 @ ([T, pc_T, \tilde{\kappa}_T, \tilde{\eta}_T]_{T \in \text{Thrd}}, \tilde{x}, \tilde{u})\), is as described in Tables 7.16, 7.17 and 7.18. Note that \(p\) and \(r\) for \(T_1\), and \(r\) for \(T_2\), are initialized to \([0, 0]\), and that \(p\) for \(T_2\) is initialized to \([2, 2]\). The tables also collect all the configurations derived by \text{ABSEX}\(E\)(\{\(\tilde{c}_0\)}, [\(-\infty, \infty]\)). A ‘−’ indicates that the entry is not included in the configuration. Figure 7.19 shows the relation between the derived configurations. In the figure, final configurations are circled, timed-out configurations are circled and marked ‘\(\tau\)’, and discontinued (invalid) configurations are crossed out. \(\tilde{c}_1\) is discontinued since the timing constraints given by \(\tilde{\eta}_T \uparrow_T\), \text{ABSTIME}(\tilde{c}_1, T_2) = [10, 11] \uparrow_T\, [4, 5] = [14, 16] and the lock owner assignment deadline, \([\infty, 12]\), give that \(T_2\) cannot acquire 1 before \(T_1\). \(\tilde{c}_{12}\) is discontinued since \(T_1\) cannot acquire 1 after
reaching a halt-statement. Given \( \tilde{c}_7^2 \), a store to \( x \) in \( T_2 \) could affect the value loaded by \( T_1 \); however, the value loaded by \( T_1 \) cannot be affected after \( \tilde{r}_T^a \), \( \tilde{r}_T \) = [9, 12] \( \tilde{r}_T \) = [2, 3] = [11, 15].

It is apparent that \( \text{ABSEXE}(\{c_0\}, [\infty, \infty]) = (\{c_{16}\}, 0, 0) \); i.e., \( c_{16} \) is a final-state configuration and there are no deadlocked or timed-out configurations. Note that \( c_{32}^2 \) only exists within the recursively called \( \text{ABSEXE} \)-instance. According to Algorithm 6.13, it is thus easy to see that the estimated timing bounds are:

\[
\begin{align*}
\text{aBCET} &= \min\left(\left\{ \max\left(\left\{ \min(\gamma_T(\tilde{r}_T^2)) \mid T \in \text{Thrd} \right\} \mid \langle [T, p_{CT}, \tilde{r}_T, \tilde{r}_T]_{T \in \text{Thrd}, \tilde{x}, \tilde{I}} \rangle \in \{c_{16}\} \right\} \right\} = 27 \\
\text{aWCET} &= \max\left(\left\{ \max\left(\left\{ \min(\gamma_T(\tilde{r}_T^2)) \mid T \in \text{Thrd} \right\} \mid \langle [T, p_{CT}, \tilde{r}_T, \tilde{r}_T]_{T \in \text{Thrd}, \tilde{x}, \tilde{I}} \rangle \in \{c_{16}\} \right\} \right\} = 42
\end{align*}
\]
### 7.4 Data Parallel Loop

Table 7.16: Data parallel loop – Configurations (thread-local states).

<table>
<thead>
<tr>
<th>$\tilde{c}$</th>
<th>$pc_{T_1}$</th>
<th>$pc_{T_2}$</th>
<th>$\tilde{T}_1$ p</th>
<th>$\tilde{T}_1$ r</th>
<th>$\tilde{T}_2$ p</th>
<th>$\tilde{T}_2$ r</th>
<th>$\tilde{i}^a_{T_1}$</th>
<th>$\tilde{i}^a_{T_2}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\tilde{c}_0$</td>
<td>1</td>
<td>1</td>
<td>[0,0]</td>
<td>[0,0]</td>
<td>[2,2]</td>
<td>[0,0]</td>
<td>[0,0]</td>
<td>[0,0]</td>
</tr>
<tr>
<td>$\tilde{c}_1$</td>
<td>2</td>
<td>2</td>
<td>[1,1]</td>
<td>[0,0]</td>
<td>[3,3]</td>
<td>[0,0]</td>
<td>[2,2]</td>
<td>[2,2]</td>
</tr>
<tr>
<td>$\tilde{c}_2$</td>
<td>3</td>
<td>3</td>
<td>[1,1]</td>
<td>[5,5]</td>
<td>[3,3]</td>
<td>[9,9]</td>
<td>[3,3]</td>
<td>[3,3]</td>
</tr>
<tr>
<td>$\tilde{c}_3$</td>
<td>1</td>
<td>3</td>
<td>[1,1]</td>
<td>[5,5]</td>
<td>[3,3]</td>
<td>[9,9]</td>
<td>[4,5]</td>
<td>[3,3]</td>
</tr>
<tr>
<td>$\tilde{c}_4$</td>
<td>2</td>
<td>1</td>
<td>[2,2]</td>
<td>[5,5]</td>
<td>[3,3]</td>
<td>[9,9]</td>
<td>[6,7]</td>
<td>[7,8]</td>
</tr>
<tr>
<td>$\tilde{c}_5$</td>
<td>3</td>
<td>1</td>
<td>[2,2]</td>
<td>[12,12]</td>
<td>[3,3]</td>
<td>[9,9]</td>
<td>[7,8]</td>
<td>[7,8]</td>
</tr>
<tr>
<td>$\tilde{c}_6$</td>
<td>4</td>
<td>2</td>
<td>[2,2]</td>
<td>[12,12]</td>
<td>[4,4]</td>
<td>[9,9]</td>
<td>[8,10]</td>
<td>[9,10]</td>
</tr>
<tr>
<td>$\tilde{c}_7$</td>
<td>4</td>
<td>3</td>
<td>[2,2]</td>
<td>[12,12]</td>
<td>[4,4]</td>
<td>[20,20]</td>
<td>[8,10]</td>
<td>[10,11]</td>
</tr>
<tr>
<td>$\tilde{c}_8$</td>
<td>5</td>
<td>3</td>
<td>[2,2]</td>
<td>[12,12]</td>
<td>[4,4]</td>
<td>[20,20]</td>
<td>[9,12]</td>
<td>[10,11]</td>
</tr>
<tr>
<td>$\tilde{c}_{72}$</td>
<td>–</td>
<td>4</td>
<td>–</td>
<td>–</td>
<td>[4,4]</td>
<td>[20,20]</td>
<td>–</td>
<td>[14,16]</td>
</tr>
<tr>
<td>$\tilde{c}_8$</td>
<td>6</td>
<td>3</td>
<td>[0,0]</td>
<td>[12,12]</td>
<td>[4,4]</td>
<td>[20,20]</td>
<td>[11,15]</td>
<td>[10,11]</td>
</tr>
<tr>
<td>$\tilde{c}_9$</td>
<td>7</td>
<td>4</td>
<td>[12,12]</td>
<td>[12,12]</td>
<td>[4,4]</td>
<td>[20,20]</td>
<td>[12,16]</td>
<td>[14,16]</td>
</tr>
<tr>
<td>$\tilde{c}_{10}$</td>
<td>8</td>
<td>4</td>
<td>[12,12]</td>
<td>[12,12]</td>
<td>[4,4]</td>
<td>[20,20]</td>
<td>[14,19]</td>
<td>[14,16]</td>
</tr>
<tr>
<td>$\tilde{c}_{11}$</td>
<td>9</td>
<td>4</td>
<td>[12,12]</td>
<td>[12,12]</td>
<td>[4,4]</td>
<td>[20,20]</td>
<td>[16,22]</td>
<td>[14,16]</td>
</tr>
<tr>
<td>$\tilde{c}_{12}$</td>
<td>9</td>
<td>4</td>
<td>[12,12]</td>
<td>[12,12]</td>
<td>[4,4]</td>
<td>[20,20]</td>
<td>[16,22]</td>
<td>[14,16]</td>
</tr>
<tr>
<td>$\tilde{c}_{13}$</td>
<td>9</td>
<td>5</td>
<td>[12,12]</td>
<td>[12,12]</td>
<td>[4,4]</td>
<td>[20,20]</td>
<td>[16,22]</td>
<td>[19,28]</td>
</tr>
<tr>
<td>$\tilde{c}_{14}$</td>
<td>9</td>
<td>6</td>
<td>[12,12]</td>
<td>[12,12]</td>
<td>[12,12]</td>
<td>[20,20]</td>
<td>[16,22]</td>
<td>[21,33]</td>
</tr>
<tr>
<td>$\tilde{c}_{15}$</td>
<td>9</td>
<td>7</td>
<td>[12,12]</td>
<td>[12,12]</td>
<td>[32,32]</td>
<td>[20,20]</td>
<td>[16,22]</td>
<td>[23,35]</td>
</tr>
<tr>
<td>$\tilde{c}_{16}$</td>
<td>9</td>
<td>8</td>
<td>[12,12]</td>
<td>[12,12]</td>
<td>[32,32]</td>
<td>[20,20]</td>
<td>[16,22]</td>
<td>[25,39]</td>
</tr>
<tr>
<td>$\tilde{c}_{17}$</td>
<td>9</td>
<td>9</td>
<td>[12,12]</td>
<td>[12,12]</td>
<td>[32,32]</td>
<td>[20,20]</td>
<td>[16,22]</td>
<td>[27,42]</td>
</tr>
</tbody>
</table>
Table 7.17: Data parallel loop – Configurations (variable states).

<table>
<thead>
<tr>
<th>( \bar{c} )</th>
<th>(( \tilde{x} x )) T(_1)</th>
<th>(( \tilde{x} x )) T(_2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \bar{c}_0 )</td>
<td>{([0, 0], [0, 0])}</td>
<td>{([0, 0], [0, 0])}</td>
</tr>
<tr>
<td>( \bar{c}_1 )</td>
<td>{([0, 0], [0, 0])}</td>
<td>{([0, 0], [0, 0])}</td>
</tr>
<tr>
<td>( \bar{c}_2 )</td>
<td>{([0, 0], [0, 0])}</td>
<td>{([0, 0], [0, 0])}</td>
</tr>
<tr>
<td>( \bar{c}_3 )</td>
<td>{([0, 0], [0, 0])}</td>
<td>{([0, 0], [0, 0])}</td>
</tr>
<tr>
<td>( \bar{c}_4 )</td>
<td>{([0, 0], [0, 0])}</td>
<td>{([0, 0], [0, 0])}</td>
</tr>
<tr>
<td>( \bar{c}_5 )</td>
<td>{([0, 0], [0, 0])}</td>
<td>{([0, 0], [0, 0])}</td>
</tr>
<tr>
<td>( \bar{c}_6 )</td>
<td>{([0, 0], [0, 0])}</td>
<td>{([0, 0], [0, 0])}</td>
</tr>
<tr>
<td>( \bar{c}_7 )</td>
<td>{([0, 0], [0, 0])}</td>
<td>{([0, 0], [0, 0])}</td>
</tr>
<tr>
<td>( \bar{c}_8 )</td>
<td>{([0, 0], [0, 0])}</td>
<td>{([0, 0], [0, 0])}</td>
</tr>
<tr>
<td>( \bar{c}_9 )</td>
<td>{([0, 0], [0, 0])}</td>
<td>{([0, 0], [0, 0])}</td>
</tr>
<tr>
<td>( \bar{c}_{10} )</td>
<td>{([0, 0], [0, 0]), ([12, 12], [14, 19])}</td>
<td>{([0, 0], [0, 0])}</td>
</tr>
<tr>
<td>( \bar{c}_{11} )</td>
<td>{([0, 0], [0, 0]), ([12, 12], [14, 19])}</td>
<td>{([0, 0], [0, 0])}</td>
</tr>
<tr>
<td>( \bar{c}_{12} )</td>
<td>{([0, 0], [0, 0]), ([12, 12], [14, 19])}</td>
<td>{([0, 0], [0, 0])}</td>
</tr>
<tr>
<td>( \bar{c}_{13} )</td>
<td>{([12, 12], [14, 19])}</td>
<td>{([\tilde{i}<em>{val}, \tilde{i}</em>{l}])}</td>
</tr>
<tr>
<td>( \bar{c}_{14} )</td>
<td>{([12, 12], [14, 19])}</td>
<td>{([\tilde{i}<em>{val}, \tilde{i}</em>{l}])}</td>
</tr>
<tr>
<td>( \bar{c}_{15} )</td>
<td>{([12, 12], [14, 19])}</td>
<td>{([32, 32], [25, 39])}</td>
</tr>
<tr>
<td>( \bar{c}_{16} )</td>
<td>{([12, 12], [14, 19])}</td>
<td>{([32, 32], [25, 39])}</td>
</tr>
</tbody>
</table>

\( \bar{c} \) | (\( \tilde{x} x \)) T\(_1\) | (\( \tilde{x} x \)) T\(_2\)
### Table 7.18: Data parallel loop – Configurations (lock states).

<table>
<thead>
<tr>
<th>( \bar{c} )</th>
<th>( \bar{l} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \bar{c}_0 )</td>
<td>(unlocked, ( t_{thrd} ), ( l_{thrd} ), ( t_{thrd} ), ( l_{thrd} ))</td>
</tr>
<tr>
<td>( \bar{c}_1 )</td>
<td>(unlocked, ( t_{thrd} ), ( l_{thrd} ), ( t_{thrd} ), ( l_{thrd} ))</td>
</tr>
<tr>
<td>( \bar{c}_2 )</td>
<td>(unlocked, ( t_{thrd} ), ( l_{thrd} ), ( t_{thrd} ), ( l_{thrd} ))</td>
</tr>
<tr>
<td>( \bar{c}_3 )</td>
<td>(unlocked, ( t_{thrd} ), ( l_{thrd} ), ( t_{thrd} ), ( l_{thrd} ))</td>
</tr>
<tr>
<td>( \bar{c}_4 )</td>
<td>(unlocked, ( t_{thrd} ), ( l_{thrd} ), ( t_{thrd} ), ( l_{thrd} ))</td>
</tr>
<tr>
<td>( \bar{c}_5 )</td>
<td>(unlocked, ( t_{thrd} ), ( l_{thrd} ), ( t_{thrd} ), ( l_{thrd} ))</td>
</tr>
<tr>
<td>( \bar{c}_6 )</td>
<td>(unlocked, ( t_{thrd} ), ( l_{thrd} ), ( t_{thrd} ), ( l_{thrd} ))</td>
</tr>
<tr>
<td>( \bar{c}_7 )</td>
<td>(unlocked, ( T_2, [\infty, 12] ), ( t_{thrd} ), ( l_{thrd} ))</td>
</tr>
<tr>
<td>( \bar{c}_7 )</td>
<td>(locked, ( T_1, [\infty, 12] ), ( t_{thrd} ), ( l_{thrd} ))</td>
</tr>
<tr>
<td>( \bar{c}_71 )</td>
<td>(locked, ( T_1, [\infty, 12] ), ( t_{thrd} ), ( l_{thrd} ))</td>
</tr>
<tr>
<td>( \bar{c}_72 )</td>
<td>(locked, ( T_1, [\infty, 12] ), ( t_{thrd} ), ( l_{thrd} ))</td>
</tr>
<tr>
<td>( \bar{c}_8 )</td>
<td>(locked, ( T_1, [\infty, 12] ), ( t_{thrd} ), ( l_{thrd} ))</td>
</tr>
<tr>
<td>( \bar{c}_9 )</td>
<td>(locked, ( T_1, [\infty, 12] ), ( t_{thrd} ), ( l_{thrd} ))</td>
</tr>
<tr>
<td>( \bar{c}_10 )</td>
<td>(locked, ( T_1, [\infty, 12] ), ( t_{thrd} ), ( l_{thrd} ))</td>
</tr>
<tr>
<td>( \bar{c}_{11} )</td>
<td>(unlocked, ( t_{thrd}, [-\infty, 12] ), ( T_1, [16, 22] ))</td>
</tr>
<tr>
<td>( \bar{c}_{12} )</td>
<td>(unlocked, ( T_1, [\infty, 28] ), ( T_1, [16, 22] ))</td>
</tr>
<tr>
<td>( \bar{c}_{12} )</td>
<td>(locked, ( T_2, [\infty, 28] ), ( T_1, [16, 22] ))</td>
</tr>
<tr>
<td>( \bar{c}_{13} )</td>
<td>(locked, ( T_2, [\infty, 28] ), ( T_1, [16, 22] ))</td>
</tr>
<tr>
<td>( \bar{c}_{14} )</td>
<td>(locked, ( T_2, [\infty, 28] ), ( T_1, [16, 22] ))</td>
</tr>
<tr>
<td>( \bar{c}_{15} )</td>
<td>(locked, ( T_2, [\infty, 28] ), ( T_1, [16, 22] ))</td>
</tr>
<tr>
<td>( \bar{c}_{16} )</td>
<td>(unlocked, ( t_{thrd}, [-\infty, 28] ), ( T_2, [27, 42] ))</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( \bar{c} )</th>
<th>( \bar{l} )</th>
</tr>
</thead>
</table>
Figure 7.19: Data parallel loop – Configuration relations.
Chapter 8

An Implementation of the Execution Time Analysis

In this chapter, a prototype implementation of the analysis presented in Chapters 5 and 6 is described. The implementation strives toward being somewhat more user-friendly compared to just invoking the algorithms presented in this thesis.

8.1 Choosing an Implementation Language

When choosing an implementation language for the analysis presented in this thesis, the main criteria to be considered are:

1. the effort required to write the implementation should be fairly low,

2. the implementation should not have an unreasonably low performance due to the chosen language,

3. it should be easy to make a parallel implementation of the worklist algorithm (i.e., of ABSEXE, presented in Algorithm 6.1 on page 165) that scales well with the hardware used to run it,

4. it should be easy to balance the load over the parallel computational nodes taking part in running the analysis,
5. it should be possible to have any degree of detail and complexity in the implementation of the timing model (i.e., in the implementation of ABSTIME), and

6. it should be possible to decide at runtime what implementation of the timing model (i.e., of ABSTIME) to use.

Considering only performance, an imperative language such as C or C++ would probably be the first to come to mind. However, considering the mathematical content of this thesis, using a functional language, or a language that mixes the functional and imperative domains, would probably make the most sense since writing the implementation becomes much easier compared to using a pure imperative language.

The ERLANG programming language meets all the above criteria, and it is functional. There might be other languages that also meet the criteria but the chosen language is ERLANG [8, 71].

ERLANG was developed at the telecom company Ericsson with the purpose to build massively scalable and distributed parallel soft real-time systems. A consequence is that the language offers very cheap and robust parallelism: it is comparably easy and cheap to create new processes and to communicate between processes, with regards to the software overhead.

ERLANG’s runtime system has built-in support for concurrency, distribution of the concurrent processes to different computational nodes, and node and process fault tolerance. There is also an ERLANG module providing load-balancing opportunities, which adapts the load on each computational node according to the node’s individual resources.

An ERLANG module is basically a software library, providing some functionality. The beauty of ERLANG is that modules can be loaded, unloaded and replaced at runtime. This means that the timing model (i.e., ABSTIME) can be implemented as an ERLANG module which allows for an arbitrary degree of complexity in this model and allows the use of it to become quite efficient. It also means that which implementation to use can be decided at runtime.

Regarding the performance of a program written in ERLANG, the ERLANG Web team states the following [32].

“Code which involves mainly number crunching and data processing will run about 10 times slower than an equivalent C program. This includes almost all ‘micro benchmarks’.

Large systems which spent most of their time communicating with other systems, recovering from faults and making complex decisions run at least as fast as equivalent C programs.”
5. it should be possible to have any degree of detail and complexity in
6. it should be possible to decide at runtime what implementation of the
The ERLANG programming language meets all the above criteria, and it is
functional. There might be other languages that also meet the criteria but the
mathematical content of this thesis, using a functional language, or a language
would probably be the first to come to mind. However, considering the
Considering only performance, an imperative language such as C or C++
written in ERLANG [8, 71].
An ERLANG module is basically a software library, providing some func-
tionality. The beauty of ERLANG is that modules can be loaded, unloaded and
switched at runtime. This means that the timing model (i.e., ABS TIME ) can be
chosen language is ERLANG [8, 71].

Since the main concern for implementing the analysis presented in this thesis
is not cutting edge performance but instead scalability, along with the other
bullets presented above, ERLANG is considered a great alternative.

8.2 UPPL: a User-Friendly Version of PPL

The concurrent programming language PPL, as presented in Chapter 4, is not
very user-friendly due to that the statements of each thread must be explicitly
labeled with consecutive labels. Therefore, the implementation will accept a
program, \( P \), written in UPPL, as presented in Table 8.1, and then translate it
into a PPL program.

The translation process is very straightforward: consecutive PPL state-
ments within each thread are labeled with consecutive integers, starting from
1; then the textual label \( l \) in \( \text{if } b \text{ goto } l \) is translated into the integer-label of
the statement \( s \) such that \( l : s \) occurs in the considered UPPL thread; lastly, all
textual labels \( l \) (and the colon, ‘:’) in \( l : s \) for some statement \( s \) are removed
from the thread.

An example UPPL program is given in Table 8.2. The program consists
of two threads, T1 and T2. T1 contains a loop that will iterate 9 times before exiting. When the loop exits, the thread immediately halts. T2 performs a simple calculation involving the register r and then halts.

8.3 Generating Initial Configurations

To ease the generation of initial configurations, another interface is provided. If not specifying anything, the implementation will assume that a default initial configuration is desired. The default initial configuration, \(\tilde{c}_{\text{def}} \in \text{Conf}\), is defined in Definition 8.1. Remember that sets of values and sets of time points are abstracted using intervals.

**Definition 8.1 (Default initial configuration):**

\[
\tilde{c}_{\text{def}} := \langle [T, 1, \tilde{x}^{\text{def}}_T, [0, 0]]_{T \in \text{Thrd}}, \tilde{x}^{\text{def}}, \tilde{u}^{\text{def}} \rangle
\]

such that

\[
\forall T \in \text{Thrd} : \forall r \in \text{Reg}_T : \tilde{x}^{\text{def}}_T r = \tilde{t}_{\text{val}} \land \\
\forall x \in \text{Var} : \forall T \in \text{Thrd} : (\tilde{x}^{\text{def}} x) T = \{ \tilde{t}_{\text{w}} \} \land \\
\forall \text{lck} \in \text{Lck} : \tilde{u}^{\text{def}} \text{lck} = (\text{unlocked}, \perp_{\text{thrd}}, \perp_{\text{l}}, \perp_{\text{thrd}}, \perp_{\text{l}})
\]
8.3 Generating Initial Configurations

The interface presented in Table 8.3 can be used to change the entries in the default initial configuration. The meaning of the different changes are as follows.

thread $d$ not included means that even though thread $d$ is defined in the program, it should not be included in the created configuration.

thread $d$ pc $pc$ means that the program counter of thread $d$ will be initialized to $pc$.

thread $d$ register $r$ nums means that the register $r$, defined in thread $d$, is initialized to the interval $[l, u]$, where $l$ is the smallest number defined in the list of numbers, $nums$; and $u$ is the largest number defined in $nums$. The comma-separated list of numbers, $nums$, can contain either single values found in $\mathbb{Z} \cup \{\text{neg\_infinity, infinity}\}$, or ranges of such values. A range is specified using the to keyword. The range “to” means all numbers from, and including, negative infinity to, and including, positive infinity. The range “to $num$” means all numbers from, and including, negative infinity to, and including, $num$. The range “$num_1$ to $num_2$” means all numbers from, and including, $num_1$ to, and including, $num_2$. The range “$num$ to” means all numbers from, and including, $num$ to, and including, positive infinity.

thread $d$ time $nums$ means that the accumulated execution time of thread $d$ is initialized to the interval $[l, u]$, where $l$ is the smallest number defined in the list of numbers, $nums$; and $u$ is the largest number defined in $nums$.

variable $x$ thread $d$ { writes } means that the write history for thread $d$ on the variable $x$ is initialized to a set containing the writes found in the comma-separated list writes. A write, $((l_v, u_v), [l_u, u_u])$, is created from write $::=$ ( ( $nums_v$ ) , ( $nums_l$ ) ) such that $l_v$ is the smallest number found in $nums_v$, $u_v$ is the largest number found in $nums_v$, $l_l$ is the smallest number found in $nums_l$, and $u_l$ is the largest number found in $nums_l$.

lock $lck$ ( state , own$_c$ , ( $nums_{dl}$ ) , own$_p$ , ( $nums_{r}$ ) ) means that the value of the lock $lck$ is initialized to the tuple $(s, o_c, [l_{dl}, u_{dl}], o_p, [l_r, u_r])$ such that: its state, $s$, is unlocked if state = unlocked or locked if state = locked; its current owner, $o_c$, is $\bot$ if $own_c = \bot$Threads or ($d$, $s_c$) for the statement $s_c$ if $own_c = d$ and thread $d$: $s_c$ is defined in the program; the deadline for the lock owner assignment, $[l_{dl}, u_{dl}]$, is such that
Table 8.3: Interface for changing initial configurations.

changes ::= change | change changes
change ::= thread d notincluded |
   thread d pc pc |
   thread d register r nums |
   thread d time nums |
   variable x thread d { writes } |
   lock lck ( state , ownc , ( nums dl ) , ownp , ( nums r ) )
nums ::= range | range , nums
range ::= to | to num | num1 to num2 | num to | num
writes ::= write | write , writes
write ::= ( ( nums v ) , ( nums t ) )
state ::= unlocked | locked
own ::= botThreads | d

where

num ∈ Z ∪ {neg_infinity,infinity}
pc : one of the integer labels of the considered thread’s statements
d : textual name of a thread
r : name of register within the considered thread
x : name of a variable
lck : name of a lock
Table 8.4: Example specification for changing an initial configuration.

<table>
<thead>
<tr>
<th>Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>thread T1 register r 4</td>
</tr>
<tr>
<td>thread T2 register r 3, 9, -5 to 6</td>
</tr>
<tr>
<td>thread T2 time 6, 7</td>
</tr>
</tbody>
</table>


\(l_{dl}\) is the smallest number found in \(nums_{dl}\) and \(u_{dl}\) is the largest number found in \(nums_{dl}\); its previous owner, \(o_p\), is \(\bot_{\text{thrd}}\) if \(own_p = \text{botThreads}\) or \((d, s_p)\) for the statement \(s_p\) if \(own_p = d\) and thread \(d\): \(s_p\) is defined in the program; the time when the lock last was released (i.e., received the state \(\text{unlocked}\)), \([l_r, u_r]\), is such that \(l_r\) is the smallest number found in \(nums_r\) and \(u_r\) is the largest number found in \(nums_r\).

An example specification for changing the default initial configuration for the program presented in Table 8.2 is given in Table 8.4. The specification in the example would generate a configuration where the value of the register \(r\) in \(T1\) is \([4, 4]\), the value of the register \(r\) in \(T2\) is \([-5, 9]\), the value of the accumulated execution time of \(T2\) is \([6, 7]\), and every other value is the same as the default value as given in Definition 8.1.

### 8.4 Implementation Architecture

The implementation exploits the parallel nature of the ERLANG programming language to achieve scalability with the available computational nodes. The ERLANG runtime system handles the scheduling of the running ERLANG threads and automatically distributes all ERLANG threads among the available computational nodes, thus achieving close to perfect load balancing (and hence scalability).

One ERLANG-thread is created for each of the initial abstract configurations in the input configuration set. Each thread then evaluates its corresponding configuration in accordance to Algorithm 6.1, defined on page 165. If a configuration is determined to be final, deadlocked, timed-out or invalid, then the handling thread terminates. Otherwise, if there are \(n\) possible transitions
from a configuration, where \( 0 < n \), then \( n - 1 \) new threads are created to further handle \( n - 1 \) of the resulting new configurations and the original thread handles the new configuration number \( n \). When all threads have terminated (or when a timeout etc. is reached, as further discussed in Section 8.5), then the bounds on the possible execution times are derived in accordance to Algorithm 6.13.

The architecture is basically such that each of the different sections (and in some cases, even algorithms) in Chapters 5 and 6 are implemented in one ERLANG module each. Together with these modules, a lexical analyzer and a parser for UPPL (as presented in Section 8.2), and a lexical analyzer and a parser for the language used to generate initial configurations (as presented in Section 8.3), are also implemented. The lexical analyzers are implemented based on the leex ERLANG module while the parsers are implemented based on the yecc ERLANG module, which accepts a Backus-Naur form (BNF) grammar definition.

One auxiliary module, corresponding to the root thread (i.e., entry point of the implementation), is also implemented to handle runtime options and thus control the behavior of the analysis (as further discussed in Section 8.5). This module also continuously collects information about the analysis progression from the threads evaluating the configurations and determines the execution time bounds from the derived final, deadlocked and timed-out configurations.

### 8.5 Runtime Options

The main input to the implementation of the analysis is a set of files defining the program to be analyzed and the name of an ERLANG module implementing the desired timing model. The main output of the implementation is the derived bounds on the BCET and WCET of the program, given the specified timing model and initial configuration(s).

It is possible to change the default behavior of the analysis (cf. Chapters 5 and 6) as described below and summarized in Table 8.5. One example of changing the default behavior could be to use the additionally implemented timeout-counters to better handle possibly nonterminating UPPL-programs. The table also describes how to invoke the analysis and how to use the options described below.

```
help:
```

Specifying this option shows information about how to invoke the analysis system and the available options. The output is similar to what is
Chapter 8. An Implementation of the Execution Time Analysis

from a configuration, where \( n < n_{\text{new}} \), then \( n_{\text{new}}-1 \) new threads are created to further handle \( n_{\text{new}}-1 \) of the resulting new configurations and the original thread handles the new configuration number \( n_{\text{new}} \). When all threads have terminated (or when a timeout etc. is reached, as further discussed in Section 8.5), then the bounds on the possible execution times are derived in accordance to Algorithm 6.13.

The architecture is basically such that each of the different sections (and in some cases, even algorithms) in Chapters 5 and 6 are implemented in one ERLANG module each. Together with these modules, a lexical analyzer and a parser for UPPL (as presented in Section 8.2), and a lexical analyzer and a parser for the language used to generate initial configurations (as presented in Section 8.3), are also implemented. The lexical analyzers are implemented based on the `leex` ERLANG module while the parsers are implemented based on the `yecc` ERLANG module, which accepts a Backus-Naur form (BNF) grammar definition.

One auxiliary module, corresponding to the root thread (i.e., entry point of the implementation), is also implemented to handle runtime options and thus control the behavior of the analysis (as further discussed in Section 8.5). This module also continuously collects information about the analysis progression from the threads evaluating the configurations and determines the execution time bounds from the derived final, deadlocked and timed-out configurations.

### 8.5 Runtime Options

The main input to the implementation of the analysis is a set of files defining the program to be analyzed and the name of an ERLANG module implementing the desired timing model. The main output of the implementation is the derived bounds on the BCET and WCET of the program, given the specified timing model and initial configuration(s).

It is possible to change the default behavior of the analysis (cf. Chapters 5 and 6) as described below and summarized in Table 8.5. One example of changing the default behavior could be to use the additionally implemented timeout-counters to better handle possibly nonterminating UPPL-programs.

The table also describes how to invoke the analysis and how to use the options described below.

#### Table 8.5: Invocation and list of controlling options.

**USAGE:**
```
erl -noshell -run engine start [OPTIONS [...]]
```

**OPTIONS:**
```
:help:
:files: file1 [file2 [...]]
:timing_model: module_name
:init.states: [file1 [file2 [...]]]
:continue_on_deadlock:
:discard_deadlocks:
:continue_on_timeout:
:timeout: time
:transition_timeout: count
:transition_predict_timeout: count
:wcet_timeout: time
:analysis_predict_timeout: count
:wcet_timeout: time
:memory_threshold: threshold
:memory_check_interval: time
:verbose:
:print_early_termination_reason:
:print_nr_transitions:
:print_analysis_time:
:final_file: file
:deadlock_file: file
:timeout_file: file
:invalid_file: file
:store_traces:
:trace_file: file
:append:
:trace_graph_html: file
:show_result:
:browser: browser
```
presented in this section.

:files:  file1 [file2 [...]]

The files file1, file2, ... should contain the UPPL program to be analyzed. The specified files are concatenated. This means that the UPPL program can be split between the files in whatever way desired as long as the concatenation of the files represents a valid UPPL program. One example of how to split the UPPL program between the files is to specify one thread in each file. This option should be specified whenever :help is not specified.

:timing_model:  module_name

The name of the ERLANG module implementing the timing model, representing ABSTIME, to be used is specified by module_name beams. This option should be specified whenever :help is not specified.

:init_states:  [file1 [file2 [...]]]

The files file1, file2, ... should contain desired modifications to the default initial configuration. One initial configuration is created per specified file. If a file is empty, then the default initial configuration, as given by Definition 8.1, is included as an initial configuration. If this option is not specified, then only the default initial configuration will be used.

:continue_on_deadlock:

The default behavior is to stop running the abstract execution algorithm if a deadlocked configuration is encountered since it then could be that the WCET is infinite. Specifying this option continues running the analysis even if a deadlocked configuration is encountered.

:discard_deadlocks:

This option will discard all encountered deadlocked configurations. This option is useful when the user knows that the analyzed system can never deadlock since deadlocked configurations can nevertheless result from over-approximations in the analysis. :continue_on_deadlock: is implied by this option.

:continue_on_timeout:

The default behavior is to stop running the abstract execution algorithm if a timed-out configuration is encountered since it then could be that the
WCET is infinite. Specifying this option continues running the analysis even if a timed-out configuration is encountered.

:timeout: time

This option sets the timeout, $\tilde{t}_{to} \in \text{Time}$, as used by Algorithm 6.13, to $\alpha_t(\{-\infty, \text{time}\})$. If not specified, then the timeout is set to $\alpha_t(\{-\infty, \infty\})$. If a timeout occurs, then the resulting bound on the WCET is set to infinity.

:transition_timeout: count

This option aborts the analysis if more than count transitions have been taken on the main recursion level (i.e., the level for which all threads specified in the program are included in the considered configurations). If this option is not specified, then there is no limit to the allowed number of transitions. If the analysis is aborted, then the resulting bound on the WCET is set to infinity.

:transition_predict_timeout: count

This option aborts the analysis if more than count transitions have been taken on recursion levels other than the main level (i.e., on levels for which not all threads specified in the program are included in the considered configurations). If this option is not specified, then there is no limit to the allowed number of transitions. If the analysis is aborted, then the resulting bound on the WCET is set to infinity.

:wct_timeout: time

This option aborts the analysis if the accumulated execution time, $\tilde{t}^a \in \text{Time}$, of some thread is such that $\text{time} < \max(\gamma_t(\tilde{t}^a))$. If the analysis is aborted, then the resulting bound on the WCET is set to infinity.

:analysis_timeout: time

This option aborts the analysis if the analysis has run for more than time seconds. If the analysis is aborted, then the resulting bound on the WCET is set to infinity.

:memory_threshold: threshold

This option aborts the analysis if the analysis has allocated more than threshold megabytes of memory. If the analysis is aborted, then the resulting bound on the WCET is set to infinity.
Chapter 8. An Implementation of the Execution Time Analysis

memory_check_interval: time

If this option is specified, then the memory usage will be checked every time milliseconds. memory_threshold must be specified for this option to be effective.

verbose:

If this option is specified, then verbose information of what is analyzed and how the analysis is performed is printed to standard output. If this option is not specified, then only the resulting bounds on the BCET and WCET are printed (unless specifying one of the options below).

print_early_termination_reason:

If this option is specified and the analysis terminates due to a detected deadlocked or timed-out configuration, or if some other timeout occurs, then the cause of the termination is printed.

print_nr_transitions:

If this option is specified, then the number of transitions taken on the main recursion level, together with the number of transitions taken on all other recursion levels except the main one, are printed.

print_analysis_time:

If this option is specified, then the approximate time required to perform the analysis is printed.

final_file: file

If this option is specified, then the derived final configurations are printed to file.

deadlock_file: file

If this option is specified, then the derived deadlocked configurations are printed to file.

timeout_file: file

If this option is specified, then the derived timed-out configurations are printed to file.

invalid_file: file

If this option is specified, then the derived invalid configurations are printed to file.
8.6 Verifying the Implementation

As a simple sanity check, the implementation of the analysis has been tested against the examples presented in Chapter 7. The result is that the derived transitions between configurations and the bounds on the BCET and WCET
for the examples are the same as the manually derived ones as presented in Chapter 7.

Also the operators \( \boxtimes_{\text{int}}', \boxtimes_{\text{int}} \) and \( \boxtimes_{\text{int}}' \), defined in Section 5.4 and used within boolean restriction of expressions containing multiplication and division, have been tested. The used tests are performed by first creating two intervals, \( i_1 \) and \( i_2 \). The interval \( i_1 \cap [l_1, u_1] \) is created by combining two values from the set \( \{-\infty, -27, -5, -2, -1, 0, 1, 2, 5, 27, \infty\} \). All combinations, which render valid intervals, are taken into account; i.e., \( i_1 \) represents one of the intervals \( [-\infty, -\infty], [-\infty, -27], \ldots, [0, 0], \ldots, [27, \infty], [\infty, \infty]. \)

The interval \( i_2 \cap [l_2, u_2] \) is created by combining two values from the set \( \{-\infty, -23, -7, -2, -1, 0, 1, 2, 7, 23, \infty\}. \) Like for \( i_1 \), all combinations which render valid intervals are taken into account.

Next, \( i_1 \boxtimes_{\text{int}}' (i_2 \cap \text{int} [-\infty, -1]) \) is derived. (Remember that all three operators are used on strictly negative intervals, strictly positive intervals or the \([0, 0]\) interval in the manner following below; cf. Tables 5.3, 5.4, 5.5, 5.6 and 5.7.)

For the cases where the result is \( \perp_{\text{int}}, \) which occurs for example when \( i_2 \cap \text{int} [-\infty, -1] = \perp_{\text{int}} \) (cf. Table 5.5), then the result is manually evaluated.

If the result of \( i_1 \boxtimes_{\text{int}}' (i_2 \cap \text{int} [-\infty, -1]) \) is a valid interval, \( [l_r, u_r] \in \text{Intv} \), and thus \( i_2 \cap \text{int} [-\infty, -1] = [l_3, u_3] \) is also a valid interval (cf. Table 5.5), then it must be that \( \frac{l_r-1}{l_3} < l_1 \lor u_1 < \frac{l_r-1}{u_3} \), \( \frac{u_r+1}{l_3} < l_1 \lor u_1 < \frac{u_r+1}{u_3} \), \( \frac{l_r-1}{l_3} < l_1 \lor u_1 < \frac{l_r-1}{u_3} \), \( \frac{u_r+1}{l_3} < l_1 \lor u_1 < \frac{u_r+1}{u_3} \) for the derived interval, \( [l_r, u_r] \), to be safe (cf. Table 4.3) whenever the extreme cases, where \( l_r \) or \( u_r \) is either \(-\infty \) or \( \infty \), are not considered. The extreme cases and division by zero are manually evaluated.

The remaining cases are evaluated in a similar manner and the details for when each case can be considered safe, including the one described above and when division by zero, \( \perp_{\text{int}} \) and the extreme cases are not considered, are summarized below.

\[
[i_r, u_r] = i_1 \boxtimes_{\text{int}}' [l_3, u_3] \quad \text{where} \\
[l_3, u_3] \in \{i_2 \cap \text{int} [-\infty, -1], i_2 \cap \text{int} [0, 0], i_2 \cap \text{int} [1, \infty]\};
\]

\[
\left( \left( \frac{l_r-1}{l_3} \right) < l_1 \lor u_1 < \left( \frac{l_r-1}{u_3} \right) \right) \wedge \\
\left( \left( \frac{l_r-1}{u_3} \right) < l_1 \lor u_1 < \left( \frac{l_r-1}{u_3} \right) \right) \wedge \\
\left( \left( \frac{u_r+1}{l_3} \right) < l_1 \lor u_1 < \left( \frac{u_r+1}{u_3} \right) \right) \wedge \\
\left( \left( \frac{u_r+1}{l_3} \right) < l_1 \lor u_1 < \left( \frac{u_r+1}{u_3} \right) \right)
\]
\[ [l_r, u_r] = i_1 \odot_{\text{int}}^* [l_3, u_3] \text{ where} \]
\[ [l_3, u_3] \in \{ I_2 \cap_{\text{int}} [\infty, -1], I_2 \cap_{\text{int}} [0, 0], I_2 \cap_{\text{int}} [1, \infty] \}: \]
\[
((l_r - 1) l_3 < l_1 \lor u_1 < (l_r - 1) l_3) \land \\
((l_r - 1) u_3 < l_1 \lor u_1 < (l_r - 1) u_3) \land \\
((u_r + 1) l_3 < l_1 \lor u_1 < (u_r + 1) l_3) \land \\
((u_r + 1) u_3 < l_1 \lor u_1 < (u_r + 1) u_3) \]

\[ [l_r, u_r] = [l_3, u_3] \odot_{\text{int}}^t i_2 \text{ where} \]
\[ [l_3, u_3] \in \{ I_1 \cap_{\text{int}} [\infty, -1], I_1 \cap_{\text{int}} [0, 0], I_1 \cap_{\text{int}} [1, \infty] \}: \]
\[
\left( \left\lfloor \frac{l_3}{l_r - 1} \right\rfloor < l_2 \land u_2 < \left\lceil \frac{l_3}{l_r - 1} \right\rceil \right) \land \\
\left( \left\lfloor \frac{u_3}{l_r - 1} \right\rfloor < l_2 \land u_2 < \left\lceil \frac{u_3}{l_r - 1} \right\rceil \right) \land \\
\left( \left\lfloor \frac{l_3}{u_r + 1} \right\rfloor < l_2 \land u_2 < \left\lceil \frac{l_3}{u_r + 1} \right\rceil \right) \land \\
\left( \left\lfloor \frac{u_3}{u_r + 1} \right\rfloor < l_2 \land u_2 < \left\lceil \frac{u_3}{u_r + 1} \right\rceil \right) \]

The outcome of the tests described above is that no unsafe cases have been encountered. This includes division by zero, the extreme cases and the different cases involving \( \bot_{\text{int}} \). Note, however, that it is very difficult to guarantee that the operators do not contain any errors by testing since the test space is infinite and human errors might occur.
Chapter 9

Evaluation

In this chapter, an evaluation of the execution time analysis presented in this thesis is performed. First, the used UPPL benchmark programs and benchmark timing models are described and discussed. Next, the hardware and software platform used to perform the benchmarking and how the benchmark programs and timing models are combined are discussed. Then, the different measured properties (namely, the measured analysis running times, the derived numbers of transitions and the derived execution time bounds) are presented, analyzed and discussed.

9.1 Benchmark Programs

Seven benchmark program types are used in the evaluation. Each program type consists of a number of (in some cases, almost) identical threads. Each table presented below defines the thread(s) within each respective program type.

The UPPL threads defined in the below tables are written in a generic way so that the programs can consist of an arbitrary number of threads. Therefore, NR THREADS is replaced with the number of threads defined in the program, NR is replaced with a consecutive number for each thread in the program, starting at 1 and ending at NR THREADS, and ITER is replaced with a desired number of loop iterations (the same number in all threads) for some loops in the definitions so that an actual UPPL program is created.
Chapter 9

Evaluation

In this chapter, an evaluation of the execution time analysis presented in this thesis is performed. First, the used UPPL benchmark programs and benchmark timing models are described and discussed. Next, the hardware and software platform used to perform the benchmarking and how the benchmark programs and timing models are combined are discussed. Then, the different measured properties (namely, the measured analysis running times, the derived numbers of transitions and the derived execution time bounds) are presented, analyzed and discussed.

9.1 Benchmark Programs

Seven benchmark program types are used in the evaluation. Each program type consists of a number of (in some cases, almost) identical threads. Each table presented below defines the thread(s) within each respective program type.

The UPPL threads defined in the below tables are written in a generic way so that the programs can consist of an arbitrary number of threads. Therefore, \( NR\_THREADS \) is replaced with the number of threads defined in the program, \( NR \) is replaced with a consecutive number for each thread in the program, starting at 1 and ending at \( NR\_THREADS \), and \( ITER \) is replaced with a desired number of loop iterations (the same number in all threads) for some loops in the definitions so that an actual UPPL program is created.

The first program type is used to evaluate how the analysis scales with regards to the number of threads defined in the analyzed program and is defined
Table 9.1: Short independent thread.

<table>
<thead>
<tr>
<th>thread T_NR:</th>
</tr>
</thead>
<tbody>
<tr>
<td>a := 0;</td>
</tr>
<tr>
<td>b := 100;</td>
</tr>
<tr>
<td>loop:</td>
</tr>
<tr>
<td>if b (\leq) a goto loop_exit;</td>
</tr>
<tr>
<td>a := a + 1;</td>
</tr>
<tr>
<td>if true goto loop;</td>
</tr>
<tr>
<td>loop_exit:</td>
</tr>
<tr>
<td>halt</td>
</tr>
</tbody>
</table>

in Table 9.1. The program consists of independent threads with only a few hundred instructions to execute. In this context, an independent thread does not communicate (i.e., does not read or write global variables) and does not synchronize (i.e., does not acquire any lock) with any other thread.

The second program type is mainly used to evaluate how the analysis scales with regards to the total number of statements to execute in the analyzed program. The program type is defined in Table 9.2 and consists of independent threads with several hundred thousand instructions to execute. If defined, the thousandth thread in this program will execute more than one hundred million statements.

The third program type is used to evaluate how the analysis scales with regards to the number of branches taken in the threads of the analyzed program. The program type is defined in Table 9.3 and the number of branches taken depends on the number of threads defined in the program and the used timing model.

The fourth program type is used to evaluate how the analysis scales with regards to the amount of thread communication. The program type is defined in Table 9.4. The communication is taking place via the variable named \(x\). Depending on the used timing model, \(x\) could have many possible values at each point in time at which some thread wants to read the value of \(x\).

The fifth program type is used to evaluate how the analysis scales with regards to the amount of thread synchronization. The program type is defined in Table 9.5. The synchronization is taking place via the lock named \(lck\).
Table 9.1: Short independent thread.

<table>
<thead>
<tr>
<th>thread</th>
<th>T_NR</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0</td>
</tr>
<tr>
<td>b</td>
<td>100</td>
</tr>
<tr>
<td>loop</td>
<td></td>
</tr>
<tr>
<td>if b &lt;= a goto loop_exit;</td>
<td></td>
</tr>
<tr>
<td>a := a + 1;</td>
<td></td>
</tr>
<tr>
<td>if true goto loop;</td>
<td></td>
</tr>
</tbody>
</table>
loop_exit: goto loop

Table 9.2: Long independent thread.

<table>
<thead>
<tr>
<th>thread</th>
<th>T_NR</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0</td>
</tr>
<tr>
<td>b</td>
<td>100000 * NR</td>
</tr>
<tr>
<td>loop</td>
<td></td>
</tr>
<tr>
<td>if b &lt;= a goto loop_exit;</td>
<td></td>
</tr>
<tr>
<td>a := a + 1;</td>
<td></td>
</tr>
<tr>
<td>if true goto loop;</td>
<td></td>
</tr>
</tbody>
</table>
loop_exit: goto loop

Table 9.3: Branching heavy thread.

<table>
<thead>
<tr>
<th>thread</th>
<th>T_NR</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>NR</td>
</tr>
<tr>
<td>store a to x;</td>
<td></td>
</tr>
<tr>
<td>load a from x;</td>
<td></td>
</tr>
<tr>
<td>i</td>
<td>0</td>
</tr>
<tr>
<td>n</td>
<td>2 * NR_THREADS</td>
</tr>
<tr>
<td>loop</td>
<td></td>
</tr>
<tr>
<td>if n &lt;= i goto loop_exit;</td>
<td></td>
</tr>
<tr>
<td>if a &lt;= n/4 &amp;&amp; a == n/4 goto inc_i;</td>
<td></td>
</tr>
<tr>
<td>a := a - 1;</td>
<td></td>
</tr>
<tr>
<td>inc_i:</td>
<td></td>
</tr>
<tr>
<td>i := i + 1;</td>
<td></td>
</tr>
<tr>
<td>if true goto loop;</td>
<td></td>
</tr>
</tbody>
</table>
loop_exit: goto loop
halt
Depending on the used timing model, lck could be heavily congested at each point in time when some thread wants to acquire it.

The sixth program type is used to evaluate how the analysis scales with regards to the amount of communication and synchronization combined. The program type is defined in Table 9.6. The communication is taking place via the variable named x and the synchronization is taking place via the lock named lck. Depending on the used timing model, x could have many possible values when some thread wants to read the value of x, and lck could be heavily congested at each point in time when some thread wants to acquire it. The programming structure used in this program type is, in general, very bad from a parallel scalability point of view, both regarding execution time and analyzability due to the large amount of dependencies introduced between the threads. The structure of the loop body (i.e., the mutually exclusive communication) is, however, very common in real-life programs since it protects shared memory from race conditions.

In the fourth, fifth and sixth program types, \textit{ITER} is replaced by 1, 5 and 10, which in practice creates three program types from each of these program types. The resulting program types are referred to as communication and/or synchronization light, medium heavy, and heavy, respectively.

The seventh program type is used to evaluate how the analysis scales with regards to the number of threads cooperating to solve a given data parallel
Table 9.5: Synchronization thread.

```plaintext
thread T_NR:
    a := 0;
    b := ITER;
    loop:
        if b <= a goto loop_exit;
        lock lck;
        unlock lck;
        a := a + 1;
        if true goto loop;
    loop_exit:
    halt
```

Table 9.6: Communication and synchronization thread.

```plaintext
thread T_NR:
    a := 0;
    b := ITER;
    loop:
        if b <= a goto loop_exit;
        lock lck;
        load c from x;
        store a to x;
        unlock lck;
        a := a + 1;
        if true goto loop;
    loop_exit:
    halt
```
Table 9.7: Well-structured data parallel thread.

thread T_NR:
    n := 256;
    lower_limit := (NR - 1)*(n / NR_THREADS);
    upper_limit := lower_limit + (n / NR_THREADS);
    i := lower_limit;
    local_sum := 0;
    if !NR == 1 goto loop;
    lock lck;
    store local_sum to sum;
    unlock lck;
    loop:
        if upper_limit <= i goto loop_exit;
        local_sum := (local_sum + i) / 256;
        i := i + 1;
        if true goto loop;
    loop_exit:
    lock lck;
    load a from sum;
    a := a + local_sum;
    store a to sum;
    unlock lck;
    halt

problem—a fairly realistic parallel problem. The program type is defined in Table 9.7. Each thread first performs local operations to calculate a given part of the problem (i.e., a local sum). When a thread has finished its local calculation, it adds the result to a global sum and halts. All local calculations are performed independently from the other threads; i.e., no communication or synchronization occurs. However, when the global sum is updated, the operations must be protected to avoid race conditions. The programming structure used in this program type is, in general, fairly good from a parallel scalability point of view, especially regarding execution time due to the low amount of dependencies introduced between the threads.
9.2 Benchmark Timing Models

Four timing models are presented in the tables below and these are used in the evaluation. Each timing model is a simple look-up table indexed by the executed statement type. The first three timing models assume that only the executed statement type affects the execution time. The fourth timing model, on the other hand, is dependent on both the executed statement type and which thread is executing the statement.

The function $A_{op} : \mathbb{A} \exp \rightarrow \mathbb{N}^0$ gives the number of operations in a given arithmetical expression and is defined in Definition 9.1. The function $B_{op} : \mathbb{B} \exp \rightarrow \mathbb{N}^0$ gives the number of operations in a given boolean expression and is defined in Definition 9.2.

**Definition 9.1 (Number of operations in an arithmetical expression):**

$$A_{op}(a) = \begin{cases} 
1 + A_{op}(a') + A_{op}(a'') & \text{if } a = a' + a'' \lor \\
1 + A_{op}(a') - A_{op}(a'') & \text{if } a = a' - a'' \lor \\
1 + A_{op}(a') \cdot A_{op}(a'') & \text{if } a = a' \cdot a'' \lor \\
1 + A_{op}(a') / A_{op}(a'') & \text{if } a = a' / a'' \\
0 & \text{otherwise}
\end{cases}$$

**Definition 9.2 (Number of operations in a boolean expression):**

$$B_{op}(b) = \begin{cases} 
1 + B_{op}(b') & \text{if } b = ! b' \\
1 + B_{op}(b') + B_{op}(b'') & \text{if } b = b' \& b'' \\
1 + A_{op}(a') + A_{op}(a'') & \text{if } b = a' == a'' \lor \\
1 + A_{op}(a') & \text{if } b = a' <= a'' \\
0 & \text{otherwise}
\end{cases}$$

The first timing model is defined in Table 9.8. This timing model has a relatively short timing range for (i.e., a relatively short distance between the bounds on the BCET and WCET of) each statement type. The execution time of each statement type is relatively large in comparison to the timing range. This timing model is meant to represent a very precise timing model; e.g., a model of a highly timing-predictable architecture.

The second timing model is defined in Table 9.9. This timing model has a moderately short timing range for each statement type. The execution time of each statement type is still relatively large in comparison to the timing range,
Table 9.8: Short range and large separation timing model.

<table>
<thead>
<tr>
<th>Statement</th>
<th>Execution time</th>
</tr>
</thead>
<tbody>
<tr>
<td>skip</td>
<td>$\alpha_t({1})$</td>
</tr>
<tr>
<td>$r := a$</td>
<td>$\alpha_t({2}) \oplus \alpha_t({\mathcal{A}_{op}(a)}) \oplus \alpha_t({10, 11})$</td>
</tr>
<tr>
<td>if $b$ goto $label$</td>
<td>$\alpha_t({2}) \oplus \alpha_t({\mathcal{B}_{op}(b)}) \oplus \alpha_t({8})$</td>
</tr>
<tr>
<td>load $r$ from $x$</td>
<td>$\alpha_t({203, 204})$</td>
</tr>
<tr>
<td>store $r$ to $x$</td>
<td>$\alpha_t({200, 201})$</td>
</tr>
<tr>
<td>lock $lck$</td>
<td>$\alpha_t({150, 151})$</td>
</tr>
<tr>
<td>unlock $lck$</td>
<td>$\alpha_t({100})$</td>
</tr>
</tbody>
</table>

Table 9.9: Medium range and medium separation timing model.

<table>
<thead>
<tr>
<th>Statement</th>
<th>Execution time</th>
</tr>
</thead>
<tbody>
<tr>
<td>skip</td>
<td>$\alpha_t({1})$</td>
</tr>
<tr>
<td>$r := a$</td>
<td>$\alpha_t({2}) \oplus \alpha_t({\mathcal{A}_{op}(a)}) \oplus \alpha_t({10, 12})$</td>
</tr>
<tr>
<td>if $b$ goto $label$</td>
<td>$\alpha_t({2}) \oplus \alpha_t({\mathcal{B}_{op}(b)}) \oplus \alpha_t({8})$</td>
</tr>
<tr>
<td>load $r$ from $x$</td>
<td>$\alpha_t({203, 214})$</td>
</tr>
<tr>
<td>store $r$ to $x$</td>
<td>$\alpha_t({200, 210})$</td>
</tr>
<tr>
<td>lock $lck$</td>
<td>$\alpha_t({150, 160})$</td>
</tr>
<tr>
<td>unlock $lck$</td>
<td>$\alpha_t({100, 110})$</td>
</tr>
</tbody>
</table>
Table 9.10: Long range and low separation timing model.

<table>
<thead>
<tr>
<th>Statement</th>
<th>Execution time</th>
</tr>
</thead>
<tbody>
<tr>
<td>skip</td>
<td>(\alpha_t({1,2}))</td>
</tr>
<tr>
<td>(r := a)</td>
<td>(\alpha_t({2,3}) \tilde{\cdot} \alpha_t({A_{\text{op}}(a)}) \tilde{\cdot} \alpha_t({8,13}))</td>
</tr>
<tr>
<td>if (b) goto (\text{label})</td>
<td>(\alpha_t({2,3}) \tilde{\cdot} \alpha_t({B_{\text{op}}(b)}) \tilde{\cdot} \alpha_t({7,9}))</td>
</tr>
<tr>
<td>load (r) from (x)</td>
<td>(\alpha_t({60,240}))</td>
</tr>
<tr>
<td>store (r) to (x)</td>
<td>(\alpha_t({50,260}))</td>
</tr>
<tr>
<td>lock (\text{lk})</td>
<td>(\alpha_t({110,211}))</td>
</tr>
<tr>
<td>unlock (\text{lk})</td>
<td>(\alpha_t({70,150}))</td>
</tr>
</tbody>
</table>

but not as large as for the first timing model. This timing model is meant to represent a moderately timing-predictable architecture.

The third timing model is defined in Table 9.10. This timing model has a relatively large timing range for each statement type. The execution time of each statement type is relatively short in comparison to the timing range. This timing model is meant to represent a quite timing-unpredictable architecture; e.g., a commercial off-the-shelf (COTS) multi-core CPU. However, no comparison to the actual timing properties of such an architecture has been performed.

The fourth timing model, defined in Table 9.11, is of the heterogeneous kind. In the table, \(\text{NR}\) is replaced by the number of the thread that is executing the given statement (cf. the benchmark programs presented in Section 9.1). Intuitively, each thread can be seen as being executed on a separate core which executes statements with a unique clock frequency; cf. a distributed system with processors executing statements at different speeds.

### 9.3 Benchmarking Setups

The evaluation is conducted by combining each of the seven program types presented in Section 9.1 (where three of them actually results in three program types each after instantiating the number of loop iterations) with each of the
Table 9.11: Heterogeneous timing model.

<table>
<thead>
<tr>
<th>Statement</th>
<th>Execution time</th>
</tr>
</thead>
<tbody>
<tr>
<td>skip</td>
<td>$\alpha_t({NR})$</td>
</tr>
<tr>
<td>$r := a$</td>
<td>$\alpha_t({NR}) \oplus_t \alpha_t({{{2}}}) \oplus_t \alpha_t({{{\alpha_{op}(a)}}}) \oplus_t \alpha_t({{10,11}})$</td>
</tr>
<tr>
<td>if $b$ goto $label$</td>
<td>$\alpha_t({NR}) \oplus_t \alpha_t({{2}}) \oplus_t \alpha_t({{{\beta_{op}(b)}}}) \oplus_t \alpha_t({{8}})$</td>
</tr>
<tr>
<td>load $r$ from $x$</td>
<td>$\alpha_t({NR}) \oplus_t \alpha_t({{203,214}})$</td>
</tr>
<tr>
<td>store $r$ to $x$</td>
<td>$\alpha_t({NR}) \oplus_t \alpha_t({{200,210}})$</td>
</tr>
<tr>
<td>lock $lck$</td>
<td>$\alpha_t({NR}) \oplus_t \alpha_t({{150,160}})$</td>
</tr>
<tr>
<td>unlock $lck$</td>
<td>$\alpha_t({NR}) \oplus_t \alpha_t({{100,110}})$</td>
</tr>
</tbody>
</table>

four timing models presented in Section 9.2. On top of this combination strategy, each program is instantiated using different numbers of threads to evaluate how the analysis scales with regards to the number of threads in the analyzed program, given a specific timing model. Each program is instantiated using 1, 2, 4, 8, 16, 32, 64, 128, 256, 512 and 1024 threads of the given type and the initial state of the system is in each case given by the default initial configuration (cf. Definition 8.1 on page 212). This combination and instantiation scheme results in $(4 + 3 \times |\{1,5,10\}|) \times 4 \times |\{1,2,4,8,16,32,64,128,256,512,1024\}| = 13 \times 4 \times 11 = 572$ benchmarking setups.

Three measures are derived for each setup and then used in the evaluation. The first measure is the measured analysis running time. This measure is very informative with regards to how the analysis scales with basically any property. The second measure is the number of derived abstract transitions between configurations. This measure is also very informative with regards to how the analysis scales with basically any property but also gives a hint on how the state space grows and how the number of derived transitions is related to the measured analysis running time. The third measure is the derived execution time bounds. This measure is not really useful for evaluating how the analysis scales but it gives a hint on how the precision of the timing model affects the precision of the derived timing bounds.

The benchmarking is conducted on an ordinary laptop. The laptop is an
9.4 Measured Analysis Running Times

In this section, the measured analysis running times for the different benchmarking setups are presented. The measured analysis running times are collected and presented in tables based on which benchmark program type, as presented in Section 9.1, is considered. The entries in the tables are the measured analysis running times for each benchmark timing model, as presented in Section 9.2, and each number of threads used in the benchmark program. I.e., each entry in the tables corresponds to the measured analysis running time for a unique benchmarking setup. The measured analysis running times for each benchmark program type are also visualized in a graph for each benchmark timing model; i.e., four graphs, one for each benchmark timing model, are presented for each benchmark program type.

Short independent threads setups

The measured analysis running times for the benchmarking setups which are based on the short independent threads program type, as presented in Table 9.1, are presented in Table 9.12 and visualized in Figure 9.13. As can be seen for all considered benchmark timing models, there is a non-linear growth in the analysis running time as the number of threads in the analyzed benchmark program increases. The complexity of the non-linear growth in the analysis time could be cubic, quadratic, or even exponential. The exact complexity is not

Apple MacBook Pro Retina. It has a quad-core Intel i7 CPU, running at 2.4 GHz (but clocked to 3.2 GHz) and with the ability to execute eight (hyper-) threads in parallel, and 16 GB of 1,600 MHz DDR3L synchronous dynamic random access memory (SDRAM). The operating system is Debian 7.6 (code-name “wheezy”) with Linux kernel version 3.10.

To avoid that the benchmarking process does not terminate within the laptop’s (or the author’s) lifetime, the analysis of each setup is aborted if the analysis runs for more than three hours (i.e., 10,800,000 milliseconds), if the memory usage of the analysis exceeds 10 GB, or if a deadlock is encountered (no deadlocks are encountered for the defined benchmark setups). If the analysis is aborted for some setup, then the setups consisting of the same program type and timing model but a larger number of threads are not considered. This is because these setups are expected to require even more analysis time or memory, or also be subject to deadlocks.
of much interest since the non-linearity is worrying enough in itself. However, the precision of the considered timing model does not seem to noticeably affect the analysis running time. On the other hand, when the threads of the analyzed benchmark program are executing with very different speeds (the heterogeneous timing model), there is a large increase in the analysis running time.

It can be noticed that no benchmark timing model seems to be noticeably advantageous for this benchmark program type, but the heterogeneous benchmark timing model does give a larger analysis running time overall.
Table 9.12: Analysis running time in milliseconds for the short independent threads benchmark.

<table>
<thead>
<tr>
<th>Nr threads</th>
<th>Large sep.</th>
<th>Medium sep.</th>
<th>Low sep.</th>
<th>Heterogeneous</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>30</td>
<td>26</td>
<td>31</td>
<td>30</td>
</tr>
<tr>
<td>2</td>
<td>34</td>
<td>34</td>
<td>35</td>
<td>51</td>
</tr>
<tr>
<td>4</td>
<td>50</td>
<td>50</td>
<td>52</td>
<td>98</td>
</tr>
<tr>
<td>8</td>
<td>82</td>
<td>81</td>
<td>84</td>
<td>207</td>
</tr>
<tr>
<td>16</td>
<td>144</td>
<td>146</td>
<td>153</td>
<td>455</td>
</tr>
<tr>
<td>32</td>
<td>285</td>
<td>288</td>
<td>303</td>
<td>1073</td>
</tr>
<tr>
<td>64</td>
<td>595</td>
<td>595</td>
<td>642</td>
<td>2667</td>
</tr>
<tr>
<td>128</td>
<td>1380</td>
<td>1375</td>
<td>1440</td>
<td>7119</td>
</tr>
<tr>
<td>256</td>
<td>3586</td>
<td>3664</td>
<td>3746</td>
<td>21306</td>
</tr>
<tr>
<td>512</td>
<td>10394</td>
<td>10388</td>
<td>10749</td>
<td>78072</td>
</tr>
<tr>
<td>1024</td>
<td>35009</td>
<td>35020</td>
<td>35516</td>
<td>372604</td>
</tr>
</tbody>
</table>

9.4 Measured Analysis Running Times

It can be noticed that no benchmark timing model seems to be noticeably advantageous for this benchmark program type, but the heterogeneous benchmark timing model does give a larger analysis running time overall.

Figure 9.13: Analysis running time in milliseconds for the short independent threads benchmark.
Table 9.14: Analysis running time in milliseconds for the long independent threads benchmark.

<table>
<thead>
<tr>
<th>Nr threads</th>
<th>Large sep.</th>
<th>Medium sep.</th>
<th>Low sep.</th>
<th>Heterogeneous</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>16773</td>
<td>16280</td>
<td>17636</td>
<td>19997</td>
</tr>
<tr>
<td>2</td>
<td>43774</td>
<td>43290</td>
<td>45236</td>
<td>64490</td>
</tr>
<tr>
<td>4</td>
<td>125981</td>
<td>125180</td>
<td>131285</td>
<td>222434</td>
</tr>
<tr>
<td>8</td>
<td>405004</td>
<td>404522</td>
<td>421072</td>
<td>795933</td>
</tr>
<tr>
<td>16</td>
<td>1476968</td>
<td>1468103</td>
<td>1525681</td>
<td>2990845</td>
</tr>
<tr>
<td>32</td>
<td>5838657</td>
<td>5834892</td>
<td>6068432</td>
<td>10800009</td>
</tr>
<tr>
<td>64</td>
<td>10800007</td>
<td>10800008</td>
<td>10800008</td>
<td>—</td>
</tr>
</tbody>
</table>

Long independent threads setups

The measured analysis running times for the benchmarking setups which are based on the long independent threads program type, as presented in Table 9.2, are presented in Table 9.14 and visualized in Figure 9.15. As can be seen, the analysis is terminated early (due to reaching the three hours analysis running time limit) for all benchmark timing models for some number of threads in the analyzed benchmark program.

For all considered benchmark timing models, there is a non-linear growth in the analysis running time as the number of threads in the analyzed benchmark program increases. The complexity of the non-linear growth in the analysis time could be cubic, quadratic, or even exponential. The exact complexity is not of much interest since the non-linearity is worrying enough in itself. However, the precision of the considered timing model does not seem to noticeably affect the analysis running time. On the other hand, when the threads of the analyzed benchmark program are executing with very different speeds (the heterogeneous timing model), there is also here an increase in the analysis running time.
Table 9.14: Analysis running time in milliseconds for the long independent threads benchmark.

<table>
<thead>
<tr>
<th>Nr threads</th>
<th>Large sep.</th>
<th>Medium sep.</th>
<th>Low sep.</th>
<th>Heterogeneous</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>16773</td>
<td>16280</td>
<td>17636</td>
<td>19997</td>
</tr>
<tr>
<td>2</td>
<td>43774</td>
<td>43290</td>
<td>45236</td>
<td>64490</td>
</tr>
<tr>
<td>4</td>
<td>125981</td>
<td>125180</td>
<td>131285</td>
<td>222434</td>
</tr>
<tr>
<td>8</td>
<td>405004</td>
<td>404522</td>
<td>421072</td>
<td>795933</td>
</tr>
<tr>
<td>16</td>
<td>1476968</td>
<td>1468103</td>
<td>1525681</td>
<td>2990845</td>
</tr>
<tr>
<td>32</td>
<td>5838657</td>
<td>5834892</td>
<td>6068432</td>
<td>10800009</td>
</tr>
<tr>
<td>64</td>
<td>10800007</td>
<td>10800008</td>
<td>10800008</td>
<td></td>
</tr>
</tbody>
</table>

Long independent threads setups

The measured analysis running times for the benchmarking setups which are based on the long independent threads program type, as presented in Table 9.2, are presented in Table 9.14 and visualized in Figure 9.15. As can be seen, the analysis is terminated early (due to reaching the three hours analysis running time limit) for all benchmark timing models for some number of threads in the analyzed benchmark program.

For all considered benchmark timing models, there is a non-linear growth in the analysis running time as the number of threads in the analyzed benchmark program increases. The complexity of the non-linear growth in the analysis time could be cubic, quadratic, or even exponential. The exact complexity is not of much interest since the non-linearity is worrying enough in itself. However, the precision of the considered timing model does not seem to noticeably affect the analysis running time. On the other hand, when the threads of the analyzed benchmark program are executing with very different speeds (the heterogeneous timing model), there is also here an increase in the analysis running time.

Figure 9.15: Analysis running time in milliseconds for the long independent threads benchmark.
Table 9.16: Analysis running time in milliseconds for the branching heavy benchmark.

<table>
<thead>
<tr>
<th>Nr threads</th>
<th>Large sep.</th>
<th>Medium sep.</th>
<th>Low sep.</th>
<th>Heterogeneous</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>10</td>
<td>16</td>
<td>13</td>
</tr>
<tr>
<td>2</td>
<td>13</td>
<td>13</td>
<td>17</td>
<td>14</td>
</tr>
<tr>
<td>4</td>
<td>410</td>
<td>333</td>
<td>820</td>
<td>32</td>
</tr>
<tr>
<td>8</td>
<td>6596914</td>
<td>5112064</td>
<td>–</td>
<td>120</td>
</tr>
<tr>
<td>16</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>1156</td>
</tr>
<tr>
<td>32</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>2438393</td>
</tr>
</tbody>
</table>

Branching heavy setups

The measured analysis running times for the benchmarking setups which are based on the branching heavy program type, as presented in Table 9.3, are presented in Table 9.16 and visualized in Figure 9.17. As can be seen, the analysis is terminated early for all benchmark timing models for some number of threads in the analyzed benchmark program. The early termination occurs due to reaching the 10 GB memory usage limit for the homogeneous benchmark timing models with large and medium timing separation and the heterogeneous benchmark timing model, and due to spawning too many ERLANG processes and reaching the ERLANG process number system limit for the homogeneous benchmark timing model with low timing separation (which gave no measurement result for the terminating instance, 8 threads, due to technical reasons).

Reaching the 10 GB memory usage limit and spawning too many ERLANG processes are both indications on that there is a large number of possible paths through the analyzed program. This is since all paths are analyzed concurrently and information concerning all considered paths must thus be kept in memory.

For all considered benchmark timing models, there is a non-linear growth in the analysis running time as the number of threads in the analyzed benchmark program increases. The exact complexity is not of much interest since the non-linearity is worrying enough, but unfortunately, it seems to be exponential.

It can be noticed that the heterogeneous timing model seems to be advantageous for this benchmark program type since programs consisting of a larger number of threads can be analyzed compared to using the homogeneous timing models.
Table 9.16: Analysis running time in milliseconds for the branching heavy benchmark.

<table>
<thead>
<tr>
<th>Nr threads</th>
<th>Large sep.</th>
<th>Medium sep.</th>
<th>Low sep.</th>
<th>Heterogeneous</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>10</td>
<td>16</td>
<td>13</td>
</tr>
<tr>
<td>2</td>
<td>13</td>
<td>13</td>
<td>17</td>
<td>14</td>
</tr>
<tr>
<td>4</td>
<td>410</td>
<td>333</td>
<td>820</td>
<td>32</td>
</tr>
<tr>
<td>8</td>
<td>659</td>
<td>691</td>
<td>5112</td>
<td>64</td>
</tr>
<tr>
<td>16</td>
<td>1156</td>
<td>1156</td>
<td>2438</td>
<td>2438</td>
</tr>
</tbody>
</table>

Branching heavy setups

The measured analysis running times for the benchmarking setups which are based on the branching heavy program type, as presented in Table 9.3, are presented in Table 9.16 and visualized in Figure 9.17. As can be seen, the analysis is terminated early for all benchmark timing models for some number of threads in the analyzed benchmark program. The early termination occurs due to reaching the 10 GB memory usage limit for the homogeneous benchmark timing models with large and medium timing separation and the heterogeneous benchmark timing model, and due to spawning too many ERLANG processes and reaching the ERLANG process number system limit for the homogeneous benchmark timing model with low timing separation (which gave no measurement result for the terminating instance, 8 threads, due to technical reasons).

Reaching the 10 GB memory usage limit and spawning too many ERLANG processes are both indications on that there is a large number of possible paths through the analyzed program. This is since all paths are analyzed concurrently and information concerning all considered paths must thus be kept in memory. For all considered benchmark timing models, there is a non-linear growth in the analysis running time as the number of threads in the analyzed benchmark program increases. The exact complexity is not of much interest since the non-linearity is worrying enough, but unfortunately, it seems to be exponential.

It can be noticed that the heterogeneous timing model seems to be advantageous for this benchmark program type since programs consisting of a larger number of threads can be analyzed compared to using the homogeneous timing models.

Figure 9.17: Analysis running time in milliseconds for the branching heavy benchmark.
Table 9.18: Analysis running time in milliseconds for the communication light (1 loop iteration) benchmark.

<table>
<thead>
<tr>
<th>Nr threads</th>
<th>Large sep.</th>
<th>Medium sep.</th>
<th>Low sep.</th>
<th>Heterogeneous</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>10</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>4</td>
<td>13</td>
<td>13</td>
<td>26</td>
<td>14</td>
</tr>
<tr>
<td>8</td>
<td>7830</td>
<td>7855</td>
<td>34037</td>
<td>24</td>
</tr>
<tr>
<td>16</td>
<td>10800007</td>
<td>10800007</td>
<td>10800008</td>
<td>66</td>
</tr>
<tr>
<td>32</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>277</td>
</tr>
<tr>
<td>64</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>2164</td>
</tr>
<tr>
<td>128</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>119168</td>
</tr>
<tr>
<td>256</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>10800010</td>
</tr>
</tbody>
</table>

**Communication setups**

The measured analysis running times for the benchmarking setups which are based on the communication program type, as presented in Table 9.4, are presented in Tables 9.18, 9.19 and 9.20, and visualized in Figure 9.21. As can be seen, the analysis is terminated early (due to reaching the three hours analysis running time limit) for all benchmark timing models, for some number of threads in the analyzed benchmark program and all numbers of loop iterations.

For all considered benchmark timing models, there is a non-linear growth in the analysis running time as the number of threads in the analyzed benchmark program increases. The exact complexity is not really of much interest since the non-linearity is worrying enough in itself, but unfortunately, it seems to be exponential. It also seems that a higher number of loop iterations leads to an exponential growth in the analysis time (especially consider the heterogeneous benchmark timing model up until 32 threads).

It can be noticed that the heterogeneous timing model seems to be somewhat advantageous for this benchmark program type since programs consisting of a larger number of threads can be analyzed compared to the homogeneous benchmark timing models. This is since the main characteristic of the heterogeneous timing model is to separate the execution of the instructions in the different threads, which delays communication for high-numbered threads.
Table 9.18: Analysis running time in milliseconds for the communication light (1 loop iteration) benchmark.

<table>
<thead>
<tr>
<th>Nr threads</th>
<th>Large sep.</th>
<th>Medium sep.</th>
<th>Low sep.</th>
<th>Heterogeneous</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>2</td>
<td>12</td>
<td>12</td>
<td>18</td>
<td>14</td>
</tr>
<tr>
<td>4</td>
<td>32</td>
<td>32</td>
<td>481</td>
<td>25</td>
</tr>
<tr>
<td>8</td>
<td>52387</td>
<td>52356</td>
<td>5185814</td>
<td>84</td>
</tr>
<tr>
<td>16</td>
<td>10800008</td>
<td>10800007</td>
<td>10800007</td>
<td>518</td>
</tr>
<tr>
<td>32</td>
<td></td>
<td></td>
<td></td>
<td>6410</td>
</tr>
<tr>
<td>64</td>
<td></td>
<td></td>
<td></td>
<td>1020553</td>
</tr>
<tr>
<td>128</td>
<td></td>
<td></td>
<td></td>
<td>10800007</td>
</tr>
</tbody>
</table>

Table 9.19: Analysis running time in milliseconds for the communication medium heavy (5 loop iterations) benchmark.

<table>
<thead>
<tr>
<th>Nr threads</th>
<th>Large sep.</th>
<th>Medium sep.</th>
<th>Low sep.</th>
<th>Heterogeneous</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>10</td>
<td>12</td>
<td>13</td>
</tr>
<tr>
<td>2</td>
<td>12</td>
<td>12</td>
<td>42</td>
<td>18</td>
</tr>
<tr>
<td>4</td>
<td>32</td>
<td>32</td>
<td>3905</td>
<td>39</td>
</tr>
<tr>
<td>8</td>
<td>106698</td>
<td>125547</td>
<td>10800007</td>
<td>177</td>
</tr>
<tr>
<td>16</td>
<td>10800007</td>
<td>10800007</td>
<td></td>
<td>7202</td>
</tr>
<tr>
<td>32</td>
<td></td>
<td></td>
<td></td>
<td>919730</td>
</tr>
<tr>
<td>64</td>
<td></td>
<td></td>
<td></td>
<td>10800008</td>
</tr>
</tbody>
</table>

Table 9.20: Analysis running time in milliseconds for the communication heavy (10 loop iterations) benchmark.

<table>
<thead>
<tr>
<th>Nr threads</th>
<th>Large sep.</th>
<th>Medium sep.</th>
<th>Low sep.</th>
<th>Heterogeneous</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>13</td>
</tr>
<tr>
<td>2</td>
<td>14</td>
<td>15</td>
<td>42</td>
<td>18</td>
</tr>
<tr>
<td>4</td>
<td>56</td>
<td>65</td>
<td>3905</td>
<td>39</td>
</tr>
<tr>
<td>8</td>
<td>106698</td>
<td>125547</td>
<td>10800007</td>
<td>177</td>
</tr>
<tr>
<td>16</td>
<td>10800007</td>
<td>10800007</td>
<td></td>
<td>7202</td>
</tr>
<tr>
<td>32</td>
<td></td>
<td></td>
<td></td>
<td>919730</td>
</tr>
<tr>
<td>64</td>
<td></td>
<td></td>
<td></td>
<td>10800008</td>
</tr>
</tbody>
</table>
Figure 9.21: Analysis running time in milliseconds for the communication benchmarks.

(a) Timing model with large separation.

(b) Timing model with medium separation.

(c) Timing model with low separation.

(d) Heterogeneous timing model.
9.4 Measured Analysis Running Times

Table 9.22: Analysis running time in milliseconds for the synchronization light (1 loop iteration) benchmark.

<table>
<thead>
<tr>
<th>Nr threads</th>
<th>Large sep.</th>
<th>Medium sep.</th>
<th>Low sep.</th>
<th>Heterogeneous</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>14</td>
<td>15</td>
<td>14</td>
<td>13</td>
</tr>
<tr>
<td>2</td>
<td>11</td>
<td>11</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>4</td>
<td>23</td>
<td>24</td>
<td>22</td>
<td>14</td>
</tr>
<tr>
<td>8</td>
<td>4888114</td>
<td>4510576</td>
<td>10800023</td>
<td>37</td>
</tr>
<tr>
<td>16</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>50940</td>
</tr>
</tbody>
</table>

Table 9.23: Analysis running time in milliseconds for the synchronization medium heavy (5 loop iterations) benchmark.

<table>
<thead>
<tr>
<th>Nr threads</th>
<th>Large sep.</th>
<th>Medium sep.</th>
<th>Low sep.</th>
<th>Heterogeneous</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>2</td>
<td>14</td>
<td>51</td>
<td>146</td>
<td>19</td>
</tr>
<tr>
<td>4</td>
<td>14584</td>
<td>9410076</td>
<td>4120555</td>
<td>10800041</td>
</tr>
<tr>
<td>8</td>
<td>572161</td>
<td>−</td>
<td>−</td>
<td>−</td>
</tr>
</tbody>
</table>

**Synchronization setups**

The measured analysis running times for the benchmarking setups which are based on the synchronization program type, as presented in Table 9.5, are presented in Tables 9.22, 9.23 and 9.24, and visualized in Figure 9.25. As can be seen, the analysis is terminated early for all benchmark timing models and all numbers of loop iterations. For the light benchmarking setups, the early termination occurs due to reaching the three hours analysis running time limit for the homogeneous benchmark timing model with low separation of the timing behavior of the statements within each thread, and due to reaching the 10 GB memory usage limit for all other benchmark timing models.

For the medium heavy benchmarking setups, the early termination occurs due to reaching the three hours analysis running time limit for the heterogeneous benchmark timing model, and due to reaching the 10 GB memory usage...
Table 9.24: Analysis running time in milliseconds for the synchronization heavy (10 loop iterations) benchmark.

<table>
<thead>
<tr>
<th>Nr threads</th>
<th>Large sep.</th>
<th>Medium sep.</th>
<th>Low sep.</th>
<th>Heterogeneous</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>12</td>
<td>12</td>
<td>14</td>
<td>13</td>
</tr>
<tr>
<td>2</td>
<td>16</td>
<td>10800018</td>
<td>10800019</td>
<td>184282</td>
</tr>
<tr>
<td>4</td>
<td>10800017</td>
<td>–</td>
<td>–</td>
<td>3584782</td>
</tr>
</tbody>
</table>

For the heavy benchmarking setups, the early termination occurs due to reaching the three hours analysis running time limit for the homogeneous benchmark timing models, and due to reaching the 10 GB memory usage limit for the heterogeneous benchmark timing model.

For all considered benchmark timing models, there is a non-linear growth in the analysis running time as the number of threads in the analyzed benchmark program increases. The exact complexity is not really of much interest since the non-linearity is worrying enough in itself, but unfortunately, it seems to be exponential. It also seems that a higher number of loop iterations leads to an exponential growth in the analysis time.

As for the communication benchmarking setups, it can be noticed that the heterogeneous timing model seems to be somewhat advantageous for this benchmark program type since programs consisting of a larger number of threads can be analyzed compared to the homogeneous benchmark timing models, especially when the number of loop iterations is small. However, this is not as visible as for the communication benchmarking setups.

It can also be noticed that only using the homogeneous benchmark timing model with large separation in the timing behavior of the statements within each thread or the heterogeneous benchmark timing model allows for analyzing this benchmark program type when the program does in fact consist of more than one thread when the synchronization is heavy. However, using any of these two timing models does not allow for analyzing the benchmark program when it consists of more than two threads and the synchronization is heavy, given the specified termination conditions.
Table 9.24: Analysis running time in milliseconds for the synchronization heavy (10 loop iterations) benchmark.

<table>
<thead>
<tr>
<th>Nr threads</th>
<th>Large sep.</th>
<th>Medium sep.</th>
<th>Low sep.</th>
<th>Heterogeneous</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>12</td>
<td>12</td>
<td>14</td>
<td>13</td>
</tr>
<tr>
<td>2</td>
<td>16</td>
<td>18</td>
<td>18</td>
<td>20</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td>17</td>
<td>17</td>
<td>358478</td>
</tr>
</tbody>
</table>

For the homogeneous benchmark timing models, the early termination occurs due to reaching the three hours analysis running time limit. For the heterogeneous benchmark timing model, due to reaching the 10 GB memory usage limit.

For all considered benchmark timing models, there is a non-linear growth in the analysis running time as the number of threads in the analyzed benchmark program increases. The exact complexity is not really of much interest since the non-linearity is worrying enough in itself, but unfortunately, it seems to be exponential. It also seems that a higher number of loop iterations leads to an exponential growth in the analysis time.

As for the communication benchmarking setups, it can be noticed that the heterogeneous timing model seems to be somewhat advantageous for this benchmark program type since programs consisting of a larger number of threads can be analyzed compared to the homogeneous benchmark timing models, especially when the number of loop iterations is small. However, this is not as visible as for the communication benchmarking setups.

It can also be noticed that only using the homogeneous benchmark timing model with large separation in the timing behavior of the statements within each thread or the heterogeneous benchmark timing model allows for analyzing this benchmark program type when the program does in fact consist of more than one thread when the synchronization is heavy. However, using any of these two timing models does not allow for analyzing the benchmark program when it consists of more than two threads and the synchronization is heavy, given the specified termination conditions.

Figure 9.25: Analysis running time in milliseconds for the synchronization benchmarks.
Table 9.26: Analysis running time in milliseconds for the communication and synchronization light (1 loop iteration) benchmark.

<table>
<thead>
<tr>
<th>Nr threads</th>
<th>Large sep.</th>
<th>Medium sep.</th>
<th>Low sep.</th>
<th>Heterogeneous</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>4</td>
<td>26</td>
<td>26</td>
<td>28</td>
<td>17</td>
</tr>
<tr>
<td>8</td>
<td>815404</td>
<td>748279</td>
<td>674230</td>
<td>22992</td>
</tr>
<tr>
<td>16</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>54030</td>
</tr>
</tbody>
</table>

Table 9.27: Analysis running time in milliseconds for the communication and synchronization medium heavy (5 loop iterations) benchmark.

<table>
<thead>
<tr>
<th>Nr threads</th>
<th>Large sep.</th>
<th>Medium sep.</th>
<th>Low sep.</th>
<th>Heterogeneous</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td>2</td>
<td>16</td>
<td>108</td>
<td>265</td>
<td>88</td>
</tr>
<tr>
<td>4</td>
<td>8441123</td>
<td>4419199</td>
<td>3298978</td>
<td>7533082</td>
</tr>
</tbody>
</table>

**Communication and synchronization setups**

The measured analysis running times for the benchmarking setups which are based on the communication and synchronization program type, as presented in Table 9.6, are presented in Tables 9.26, 9.27 and 9.28, and visualized in Figure 9.29. As can be seen, the analysis is terminated early for all benchmark timing models and all numbers of loop iterations. For the light and medium heavy benchmarking setups, the early termination occurs due to reaching the 10 GB memory usage limit for all benchmark timing models.

For the heavy benchmarking setups, the early termination occurs due to reaching the 10 GB memory usage limit for the homogeneous benchmark timing model with large separation in the timing behavior of the statements within each thread, and due to reaching the three hours analysis running time limit for all other benchmark timing models.

For all considered benchmark timing models, there is a non-linear growth
Table 9.26: Analysis running time in milliseconds for the communication and synchronization light (1 loop iteration) benchmark.

<table>
<thead>
<tr>
<th>Nr threads</th>
<th>Large sep.</th>
<th>Medium sep.</th>
<th>Low sep.</th>
<th>Heterogeneous</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>4</td>
<td>26</td>
<td>26</td>
<td>28</td>
<td>17</td>
</tr>
<tr>
<td>8</td>
<td>815404</td>
<td>748279</td>
<td>674230</td>
<td>22992</td>
</tr>
<tr>
<td>16</td>
<td>−−−</td>
<td>−−−</td>
<td>−−−</td>
<td>−−−</td>
</tr>
</tbody>
</table>

Table 9.27: Analysis running time in milliseconds for the communication and synchronization medium heavy (5 loop iterations) benchmark.

<table>
<thead>
<tr>
<th>Nr threads</th>
<th>Large sep.</th>
<th>Medium sep.</th>
<th>Low sep.</th>
<th>Heterogeneous</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td>2</td>
<td>16</td>
<td>10800009</td>
<td>265</td>
<td>88</td>
</tr>
<tr>
<td>4</td>
<td>8441123</td>
<td>4419199</td>
<td>3298978</td>
<td>7533082</td>
</tr>
</tbody>
</table>

Communication and synchronization setups

The measured analysis running times for the benchmarking setups which are based on the communication and synchronization program type, as presented in Table 9.6, are presented in Tables 9.26, 9.27 and 9.28, and visualized in Figure 9.29. As can be seen, the analysis is terminated early for all benchmark timing models and all numbers of loop iterations. For the light and medium heavy benchmarking setups, the early termination occurs due to reaching the 10 GB memory usage limit for all benchmark timing models.

For the heavy benchmarking setups, the early termination occurs due to reaching the 10 GB memory usage limit for the homogeneous benchmark timing model with large separation of the timing behavior of the statements within each thread, and due to reaching the three hours analysis running time limit for all other benchmark timing models.

For all considered benchmark timing models, there is a non-linear growth in the analysis running time as the number of threads in the analyzed benchmark program increases. The exact complexity is not really of much interest since the non-linearity is worrying enough in itself, but unfortunately, it seems to be exponential. It also seems that a higher number of loop iterations leads to an exponential growth in the analysis time.

As for the communication benchmarking setups and the synchronization benchmarking setups, it can be noticed that the heterogeneous timing model seems to be somewhat advantageous for this benchmark program type since programs consisting of a larger number of threads can be analyzed compared to the homogeneous benchmark timing models, especially when the number of loop iterations is small. However, this is not as visible as for the communication benchmarking setups.

It can also be noticed that only using the homogeneous benchmark timing model with large separation in the timing behavior of the statements within each thread allows for analyzing this benchmark program type when the program does in fact consist of more than one thread when the synchronization is heavy. However, not even using this timing model allows for analyzing the benchmark program when it consists of more than two threads and the synchronization is heavy.
Figure 9.29: Analysis running time in milliseconds for the communication and synchronization benchmarks.
Table 9.30: Analysis running time in milliseconds for the well-structured data parallel benchmark.

<table>
<thead>
<tr>
<th>Nr threads</th>
<th>Large sep.</th>
<th>Medium sep.</th>
<th>Low sep.</th>
<th>Heterogeneous</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>73</td>
<td>75</td>
<td>79</td>
<td>89</td>
</tr>
<tr>
<td>2</td>
<td>82</td>
<td>81</td>
<td>83</td>
<td>110</td>
</tr>
<tr>
<td>4</td>
<td>5524</td>
<td>9164</td>
<td>13819</td>
<td>425</td>
</tr>
<tr>
<td>8</td>
<td>7889672</td>
<td>10800055</td>
<td>6269353</td>
<td>10800018</td>
</tr>
</tbody>
</table>

**Well-structured data parallel setups**

The measured analysis running times for the benchmarking setups which are based on the well-structured data parallel program type, as presented in Table 9.7, are presented in Table 9.30 and visualized in Figure 9.31. As can be seen, the analysis is terminated early for all benchmark timing models. The early termination occurs due to reaching the three hours analysis running time limit for the homogeneous benchmark timing model with medium separation of the timing behavior of the statements within each thread and the heterogeneous benchmark timing model, and due to reaching the 10 GB memory usage limit for the other two homogeneous benchmark timing models.

For all considered benchmark timing models, there is a non-linear growth in the analysis running time as the number of threads in the analyzed benchmark program increases. Unfortunately, the complexity of the non-linear growth seems to be exponential even though the program is structured in an advantageous manner.
In this section, the derived numbers of transitions for the different benchmarking setups are presented. The derived numbers of transitions are collected and presented in tables based on which benchmark program type, as presented in Section 9.1, is considered. Two tables are presented for each benchmark program type, if applicable. The first table presents the numbers of derived transitions on the main recursion level (i.e., the level on which no recursion has occurred in the algorithm presented in Algorithm 6.1, presented on page 165, and on which the considered configurations thus contain all the threads defined in the analyzed program). The second table presents the numbers of derived transitions on all recursion levels other than the main recursion level (i.e., the numbers of transitions needed to derive all possible values a load-statement in some given thread could see).

The entries in the tables are the derived numbers of transitions for each benchmark timing model, as presented in Section 9.2, and each number of threads used in the benchmark program. I.e., each entry in the tables corresponds to the derived number of transitions for a unique benchmarking setup. The derived numbers of transitions for each benchmark program type are also visualized in a graph for each benchmark timing model and recursion level type, if applicable; i.e., eight graphs, one for each benchmark timing model, combined once with the main recursion level and once with all other recursion levels, are presented for each benchmark program type, if applicable. In the graphs, the value 0 is not displayed due to the limitations of the logarithmic axes.

Short independent threads setups

The derived numbers of transitions on the main recursion level for the benchmarking setups which are based on the short independent threads program type, as presented in Table 9.1, are presented in Table 9.32 and visualized in Figure 9.33. There are no transitions on recursion levels other than the main recursion level for any of the setups (which is natural since no communication occurs between the threads in this benchmark program type).

As can be seen, the numbers of transitions for the homogeneous benchmark timing models are the same regardless of how many threads the program consists of. This is since the threads in the program are identical and the timing of a statement is the same regardless of which thread executes it. Thus, it can be established that the non-linear growth of the analysis running time, presented...
9.5 Numbers of Derived Transitions

In this section, the derived numbers of transitions for the different benchmarking setups are presented. The derived numbers of transitions are collected and presented in tables based on which benchmark program type, as presented in Section 9.1, is considered. Two tables are presented for each benchmark program type, if applicable. The first table presents the numbers of derived transitions on the main recursion level (i.e., the level on which no recursion has occurred in the algorithm presented in Algorithm 6.1, presented on page 165, and on which the considered configurations thus contain all the threads defined in the analyzed program). The second table presents the numbers of derived transitions on all recursion levels other than the main recursion level (i.e., the numbers of transitions needed to derive all possible values a load-statement in some given thread could see).

The entries in the tables are the derived numbers of transitions for each benchmark timing model, as presented in Section 9.2, and each number of threads used in the benchmark program. I.e., each entry in the tables corresponds to the derived number of transitions for a unique benchmarking setup. The derived numbers of transitions for each benchmark program type are also visualized in a graph for each benchmark timing model and recursion level type, if applicable; i.e., eight graphs, one for each benchmark timing model, combined once with the main recursion level and once with all other recursion levels, are presented for each benchmark program type, if applicable. In the graphs, the value 0 is not displayed due to the limitations of the logarithmic axes.

Short independent threads setups

The derived numbers of transitions on the main recursion level for the benchmarking setups which are based on the short independent threads program type, as presented in Table 9.1, are presented in Table 9.32 and visualized in Figure 9.33. There are no transitions on recursion levels other than the main recursion level for any of the setups (which is natural since no communication occurs between the threads in this benchmark program type).

As can be seen, the numbers of transitions for the homogeneous benchmark timing models are the same regardless of how many threads the program consists of. This is since the threads in the program are identical and the timing of a statement is the same regardless of which thread executes it. Thus, it can be established that the non-linear growth of the analysis running time, presented
Table 9.32: Transitions on the main recursion level for the short independent threads benchmark.

<table>
<thead>
<tr>
<th>Nr threads</th>
<th>Large sep.</th>
<th>Medium sep.</th>
<th>Low sep.</th>
<th>Heterogeneous</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>303</td>
<td>303</td>
<td>303</td>
<td>303</td>
</tr>
<tr>
<td>2</td>
<td>303</td>
<td>303</td>
<td>303</td>
<td>450</td>
</tr>
<tr>
<td>4</td>
<td>303</td>
<td>303</td>
<td>303</td>
<td>614</td>
</tr>
<tr>
<td>8</td>
<td>303</td>
<td>303</td>
<td>303</td>
<td>783</td>
</tr>
<tr>
<td>16</td>
<td>303</td>
<td>303</td>
<td>303</td>
<td>954</td>
</tr>
<tr>
<td>32</td>
<td>303</td>
<td>303</td>
<td>303</td>
<td>1128</td>
</tr>
<tr>
<td>64</td>
<td>303</td>
<td>303</td>
<td>303</td>
<td>1319</td>
</tr>
<tr>
<td>128</td>
<td>303</td>
<td>303</td>
<td>303</td>
<td>1550</td>
</tr>
<tr>
<td>256</td>
<td>303</td>
<td>303</td>
<td>303</td>
<td>1861</td>
</tr>
<tr>
<td>512</td>
<td>303</td>
<td>303</td>
<td>303</td>
<td>2392</td>
</tr>
<tr>
<td>1024</td>
<td>303</td>
<td>303</td>
<td>303</td>
<td>3553</td>
</tr>
</tbody>
</table>

In Section 9.4, for this benchmark program type is due to the increasing number of threads in the program. Strategies similar to symmetry reductions in model checking [22] could be incorporated into the implementation to reduce the analysis running time complexity for this case and other similar setups.

For the heterogeneous benchmark timing model, the increasing number of transitions explains the growth of the analysis running time as the number of threads grows. The non-linear polynomial degree in the numbers of derived transitions for this timing model is most likely due to that execution of statements in several threads in one and the same transition is sometimes more or less possible. However, this has not been verified.
Table 9.32: Transitions on the main recursion level for the short independent threads benchmark.

<table>
<thead>
<tr>
<th>Nr threads</th>
<th>Large sep.</th>
<th>Medium sep.</th>
<th>Low sep.</th>
<th>Heterogeneous</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>303</td>
<td>303</td>
<td>303</td>
<td>303</td>
</tr>
<tr>
<td>2</td>
<td>303</td>
<td>303</td>
<td>303</td>
<td>450</td>
</tr>
<tr>
<td>4</td>
<td>303</td>
<td>303</td>
<td>303</td>
<td>614</td>
</tr>
<tr>
<td>8</td>
<td>303</td>
<td>303</td>
<td>303</td>
<td>783</td>
</tr>
<tr>
<td>16</td>
<td>303</td>
<td>303</td>
<td>303</td>
<td>954</td>
</tr>
<tr>
<td>32</td>
<td>303</td>
<td>303</td>
<td>303</td>
<td>1128</td>
</tr>
<tr>
<td>64</td>
<td>303</td>
<td>303</td>
<td>303</td>
<td>1319</td>
</tr>
<tr>
<td>128</td>
<td>303</td>
<td>303</td>
<td>303</td>
<td>1550</td>
</tr>
<tr>
<td>256</td>
<td>303</td>
<td>303</td>
<td>303</td>
<td>1861</td>
</tr>
<tr>
<td>512</td>
<td>303</td>
<td>303</td>
<td>303</td>
<td>2392</td>
</tr>
<tr>
<td>1024</td>
<td>303</td>
<td>303</td>
<td>303</td>
<td>3553</td>
</tr>
</tbody>
</table>

In Section 9.4, for this benchmark program type is due to the increasing number of threads in the program. Strategies similar to symmetry reductions in model checking [22] could be incorporated into the implementation to reduce the analysis running time complexity for this case and other similar setups.

For the heterogeneous benchmark timing model, the increasing number of transitions explains the growth of the analysis running time as the number of threads grows. The non-linear polynomial degree in the numbers of derived transitions for this timing model is most likely due to that execution of state-memories in several threads in one and the same transition is sometimes more or less possible. However, this has not been verified.

Figure 9.33: Transitions on the main recursion level for the short independent threads benchmark.
Table 9.34: Transitions on the main recursion level for the long independent threads benchmark.

<table>
<thead>
<tr>
<th>Nr threads</th>
<th>Large sep.</th>
<th>Medium sep.</th>
<th>Low sep.</th>
<th>Heterogeneous</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>300003</td>
<td>300003</td>
<td>300003</td>
<td>300003</td>
</tr>
<tr>
<td>2</td>
<td>600003</td>
<td>600003</td>
<td>600003</td>
<td>743759</td>
</tr>
<tr>
<td>4</td>
<td>1200003</td>
<td>1200003</td>
<td>1200003</td>
<td>1723974</td>
</tr>
<tr>
<td>8</td>
<td>2400003</td>
<td>2400003</td>
<td>2400003</td>
<td>3715648</td>
</tr>
<tr>
<td>16</td>
<td>4800003</td>
<td>4800003</td>
<td>4800003</td>
<td>7711030</td>
</tr>
<tr>
<td>32</td>
<td>9600003</td>
<td>9600003</td>
<td>9600003</td>
<td>13003081</td>
</tr>
<tr>
<td>64</td>
<td>6052665</td>
<td>6051817</td>
<td>5769538</td>
<td>—</td>
</tr>
</tbody>
</table>

**Long independent threads setups**

The derived numbers of transitions on the main recursion level for the benchmarking setups which are based on the long independent threads program type, as presented in Table 9.2, are presented in Table 9.34 and visualized in Figure 9.35. There are no transitions on recursion levels other than the main recursion level for any of the setups (which is natural since no communication occurs between the threads in this benchmark program type).

As can be seen, the numbers of derived transitions increase linearly with the numbers of threads in the program for the homogeneous timing models. This is since the number of executed statements in one of the threads is proportional to how many threads the program consists of and all other threads execute a smaller number of statements, and the timing of executing a statement is the same regardless of which thread executes it. The dip in the derived numbers of transitions for the last considered number of threads in the program is due to reaching the 10 GB memory usage limit as discussed in Section 9.4.

The non-linear polynomial degree in the numbers of derived transitions for the heterogeneous benchmark timing model is most likely due to that execution of statements in several threads in one and the same transition is sometimes more or less possible. However, this has not been verified.
### 9.5 Numbers of Derived Transitions

The derived numbers of transitions on the main recursion level for the benchmarking setups which are based on the long independent threads program type, as presented in Table 9.2, are presented in Table 9.34 and visualized in Figure 9.35. There are no transitions on recursion levels other than the main recursion level for any of the setups (which is natural since no communication occurs between the threads in this benchmark program type).

As can be seen, the numbers of derived transitions increase linearly with the numbers of threads in the program for the homogeneous timing models. This is since the number of executed statements in one of the threads is proportional to how many threads the program consists of and all other threads execute a smaller number of statements, and the timing of executing a statement is the same regardless of which thread executes it. The dip in the derived numbers of transitions for the last considered number of threads in the program is due to reaching the 10 GB memory usage limit as discussed in Section 9.4.

The non-linear polynomial degree in the numbers of derived transitions for the heterogeneous benchmark timing model is most likely due to that execution of statements in several threads in one and the same transition is sometimes more or less possible. However, this has not been verified.

![Figure 9.35: Transitions on the main recursion level for the long independent threads benchmark.](image)
Table 9.36: Transitions on the main recursion level for the branching heavy benchmark.

<table>
<thead>
<tr>
<th>Nr threads</th>
<th>Large sep.</th>
<th>Medium sep.</th>
<th>Low sep.</th>
<th>Heterogeneous</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>2</td>
<td>68</td>
<td>62</td>
<td>62</td>
<td>81</td>
</tr>
<tr>
<td>4</td>
<td>8343</td>
<td>6793</td>
<td>6793</td>
<td>378</td>
</tr>
<tr>
<td>8</td>
<td>522103</td>
<td>455103</td>
<td>−</td>
<td>1263</td>
</tr>
<tr>
<td>16</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>8288</td>
</tr>
<tr>
<td>32</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>164456</td>
</tr>
</tbody>
</table>

Table 9.37: Transitions on recursion levels other than the main recursion level for the branching heavy benchmark.

<table>
<thead>
<tr>
<th>Nr threads</th>
<th>Large sep.</th>
<th>Medium sep.</th>
<th>Low sep.</th>
<th>Heterogeneous</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>6</td>
<td>68</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>80</td>
<td>120</td>
<td>15384</td>
<td>40</td>
</tr>
<tr>
<td>8</td>
<td>138560</td>
<td>207840</td>
<td>−</td>
<td>246</td>
</tr>
<tr>
<td>16</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>1203</td>
</tr>
<tr>
<td>32</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>76992</td>
</tr>
</tbody>
</table>
9.5 Numbers of Derived Transitions

Table 9.36: Transitions on the main recursion level for the branching heavy benchmark.

<table>
<thead>
<tr>
<th>Nr threads</th>
<th>Large sep.</th>
<th>Medium sep.</th>
<th>Low sep.</th>
<th>Heterogeneous</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>2</td>
<td>68</td>
<td>62</td>
<td>62</td>
<td>81</td>
</tr>
<tr>
<td>4</td>
<td>8343</td>
<td>6793</td>
<td>6793</td>
<td>378</td>
</tr>
<tr>
<td>8</td>
<td>522103</td>
<td>455103</td>
<td>-1263</td>
<td>-</td>
</tr>
<tr>
<td>16</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>32</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 9.37: Transitions on recursion levels other than the main recursion level for the branching heavy benchmark.

<table>
<thead>
<tr>
<th>Nr threads</th>
<th>Large sep.</th>
<th>Medium sep.</th>
<th>Low sep.</th>
<th>Heterogeneous</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>6</td>
<td>68</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>80</td>
<td>120</td>
<td>15384</td>
<td>40</td>
</tr>
<tr>
<td>8</td>
<td>138560</td>
<td>207840</td>
<td>-246</td>
<td>-</td>
</tr>
<tr>
<td>16</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>32</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Figure 9.38: Transitions on the main recursion level for the branching heavy benchmark.
Branching heavy setups

The derived numbers of transitions on the main recursion level for the benchmarking setups which are based on the branching heavy program type, as presented in Table 9.3, are presented in Table 9.36 and visualized in Figure 9.38. The derived numbers of transitions on all recursion levels other than the main recursion level are presented in Table 9.37 and visualized in Figure 9.39. Remember that there is no result due to technical reasons for the terminating instance (8 threads) for the homogeneous benchmark timing model with low timing separation of statements since the termination occurred due to spawning too many ERLANG processes and thus reaching the ERLANG process number system limit.

As can be seen, there is a large, non-linear increase in the numbers of derived transitions, on all recursion levels and for all benchmark timing models, as the number of threads in the program increases. The increase is due to an increase in the number of possible paths through the program. Note that this is true also for the recursion levels other than the main recursion level.

Communication setups

The derived numbers of transitions on the main recursion level for the benchmarking setups which are based on the communication program type, as presented in Table 9.4, are presented in Tables 9.40, 9.42 and 9.44, and visualized in Figure 9.46. The derived numbers of transitions on all recursion levels other than the main recursion level are presented in Tables 9.41, 9.43 and 9.45, and visualized in Figure 9.47.

Since the number of statements in these benchmarking setups is not dependent on the number of threads defined in the setup, the number of transitions on the main recursion level is constant for the homogeneous benchmark timing models. When a dip occurs for the last considered number of threads in the derived number of transitions for some setup, the dip is due to the early termination of the analysis, as discussed in Section 9.4.

As can be seen, there is an increase in the number of needed transitions on recursion levels other than the main recursion level as the timing model gets less precise (i.e., a larger number of transitions is needed to derive all possible values for the load-statement in this benchmark program type for timing models with lower separation of the execution times of statements). This explains the non-linear increase in the analysis running time for these setups.

Figure 9.39: Transitions on recursion levels other than the main recursion level for the branching heavy benchmark.
9.5 Numbers of Derived Transitions

Branching heavy setups

The derived numbers of transitions on the main recursion level for the benchmarking setups which are based on the branching heavy program type, as presented in Table 9.3, are presented in Table 9.36 and visualized in Figure 9.38. The derived numbers of transitions on all recursion levels other than the main recursion level are presented in Table 9.37 and visualized in Figure 9.39. Remember that there is no result due to technical reasons for the terminating instance (8 threads) for the homogeneous benchmark timing model with low timing separation of statements since the termination occurred due to spawning too many ERLANG processes and thus reaching the ERLANG process number system limit.

As can be seen, there is a large, non-linear increase in the numbers of derived transitions, on all recursion levels and for all benchmark timing models, as the number of threads in the program increases. The increase is due to an increase in the number of possible paths through the program. Note that this is true also for the recursion levels other than the main recursion level.

Communication setups

The derived numbers of transitions on the main recursion level for the benchmarking setups which are based on the communication program type, as presented in Table 9.4, are presented in Tables 9.40, 9.42 and 9.44, and visualized in Figure 9.46. The derived numbers of transitions on all recursion levels other than the main recursion level are presented in Tables 9.41, 9.43 and 9.45, and visualized in Figure 9.47.

Since the number of statements in these benchmarking setups is not dependent on the number of threads defined in the setup, the number of transitions on the main recursion level is constant for the homogeneous benchmark timing models. When a dip occurs for the last considered number of threads in the derived number of transitions for some setup, the dip is due to the early termination of the analysis, as discussed in Section 9.4.

As can be seen, there is an increase in the number of needed transitions on recursion levels other than the main recursion level as the timing model gets less precise (i.e., a larger number of transitions is needed to derive all possible values for the load-statement in this benchmark program type for timing models with lower separation of the execution times of statements). This explains the non-linear increase in the analysis running time for these setups.
Table 9.40: Transitions on the main recursion level for the communication light (1 loop iteration) benchmark.

<table>
<thead>
<tr>
<th>Nr threads</th>
<th>Large sep.</th>
<th>Medium sep.</th>
<th>Low sep.</th>
<th>Heterogeneous</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>15</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>30</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>58</td>
</tr>
<tr>
<td>16</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>108</td>
</tr>
<tr>
<td>32</td>
<td></td>
<td></td>
<td></td>
<td>179</td>
</tr>
<tr>
<td>64</td>
<td></td>
<td></td>
<td></td>
<td>295</td>
</tr>
<tr>
<td>128</td>
<td></td>
<td></td>
<td></td>
<td>485</td>
</tr>
<tr>
<td>256</td>
<td></td>
<td></td>
<td></td>
<td>754</td>
</tr>
</tbody>
</table>

Table 9.41: Transitions on recursion levels other than the main recursion level for the communication light (1 loop iteration) benchmark.

<table>
<thead>
<tr>
<th>Nr threads</th>
<th>Large sep.</th>
<th>Medium sep.</th>
<th>Low sep.</th>
<th>Heterogeneous</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>10</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>40</td>
<td>40</td>
<td>200</td>
<td>8</td>
</tr>
<tr>
<td>8</td>
<td>69280</td>
<td>69280</td>
<td>346400</td>
<td>16</td>
</tr>
<tr>
<td>16</td>
<td>60845889</td>
<td>59646041</td>
<td>79741898</td>
<td>39</td>
</tr>
<tr>
<td>32</td>
<td></td>
<td></td>
<td></td>
<td>146</td>
</tr>
<tr>
<td>64</td>
<td></td>
<td></td>
<td></td>
<td>822</td>
</tr>
<tr>
<td>128</td>
<td></td>
<td></td>
<td></td>
<td>22563</td>
</tr>
<tr>
<td>256</td>
<td></td>
<td></td>
<td></td>
<td>652390</td>
</tr>
</tbody>
</table>

Table 9.42: Transitions on the main recursion level for the communication medium heavy (5 loop iterations) benchmark.

<table>
<thead>
<tr>
<th>Nr threads</th>
<th>Large sep.</th>
<th>Medium sep.</th>
<th>Low sep.</th>
<th>Heterogeneous</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>28</td>
<td>28</td>
<td>28</td>
<td>28</td>
</tr>
<tr>
<td>2</td>
<td>28</td>
<td>28</td>
<td>28</td>
<td>47</td>
</tr>
<tr>
<td>4</td>
<td>28</td>
<td>28</td>
<td>28</td>
<td>79</td>
</tr>
<tr>
<td>8</td>
<td>28</td>
<td>28</td>
<td>28</td>
<td>135</td>
</tr>
<tr>
<td>16</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>229</td>
</tr>
<tr>
<td>32</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 9.43: Transitions on recursion levels other than the main recursion level for the communication medium heavy (5 loop iterations) benchmark.

<table>
<thead>
<tr>
<th>Nr threads</th>
<th>Large sep.</th>
<th>Medium sep.</th>
<th>Low sep.</th>
<th>Heterogeneous</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>10</td>
<td>114</td>
<td>12</td>
</tr>
<tr>
<td>4</td>
<td>200</td>
<td>200</td>
<td>6000</td>
<td>42</td>
</tr>
<tr>
<td>8</td>
<td>346400</td>
<td>346400</td>
<td>49953296</td>
<td>238</td>
</tr>
<tr>
<td>16</td>
<td>59819767</td>
<td>60380435</td>
<td>73510987</td>
<td>1097</td>
</tr>
<tr>
<td>32</td>
<td></td>
<td></td>
<td></td>
<td>7410</td>
</tr>
<tr>
<td>64</td>
<td></td>
<td></td>
<td></td>
<td>497170</td>
</tr>
<tr>
<td>128</td>
<td></td>
<td></td>
<td></td>
<td>1958018</td>
</tr>
<tr>
<td>256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 9.42: Transitions on the main recursion level for the communication medium heavy (5 loop iterations) benchmark.

<table>
<thead>
<tr>
<th>Nr threads</th>
<th>Large sep.</th>
<th>Medium sep.</th>
<th>Low sep.</th>
<th>Heterogeneous</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>28</td>
<td>28</td>
<td>28</td>
<td>28</td>
</tr>
<tr>
<td>2</td>
<td>28</td>
<td>28</td>
<td>28</td>
<td>47</td>
</tr>
<tr>
<td>4</td>
<td>28</td>
<td>28</td>
<td>28</td>
<td>79</td>
</tr>
<tr>
<td>8</td>
<td>28</td>
<td>28</td>
<td>28</td>
<td>135</td>
</tr>
<tr>
<td>16</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>229</td>
</tr>
<tr>
<td>32</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>343</td>
</tr>
<tr>
<td>64</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>511</td>
</tr>
<tr>
<td>128</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>453</td>
</tr>
</tbody>
</table>

### Table 9.43: Transitions on recursion levels other than the main recursion level for the communication medium heavy (5 loop iterations) benchmark.

<table>
<thead>
<tr>
<th>Nr threads</th>
<th>Large sep.</th>
<th>Medium sep.</th>
<th>Low sep.</th>
<th>Heterogeneous</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>10</td>
<td>114</td>
<td>12</td>
</tr>
<tr>
<td>4</td>
<td>200</td>
<td>200</td>
<td>6000</td>
<td>42</td>
</tr>
<tr>
<td>8</td>
<td>346400</td>
<td>346400</td>
<td>49953296</td>
<td>238</td>
</tr>
<tr>
<td>16</td>
<td>59819767</td>
<td>60380435</td>
<td>73510987</td>
<td>1097</td>
</tr>
<tr>
<td>32</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>7410</td>
</tr>
<tr>
<td>64</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>497170</td>
</tr>
<tr>
<td>128</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>1958018</td>
</tr>
</tbody>
</table>
Table 9.44: Transitions on the main recursion level for the communication heavy (10 loop iterations) benchmark.

<table>
<thead>
<tr>
<th>Nr threads</th>
<th>Large sep.</th>
<th>Medium sep.</th>
<th>Low sep.</th>
<th>Heterogeneous</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>53</td>
<td>53</td>
<td>53</td>
<td>53</td>
</tr>
<tr>
<td>2</td>
<td>53</td>
<td>53</td>
<td>53</td>
<td>85</td>
</tr>
<tr>
<td>4</td>
<td>53</td>
<td>53</td>
<td>53</td>
<td>133</td>
</tr>
<tr>
<td>8</td>
<td>53</td>
<td>53</td>
<td>13</td>
<td>189</td>
</tr>
<tr>
<td>16</td>
<td>3</td>
<td>3</td>
<td>—</td>
<td>296</td>
</tr>
<tr>
<td>32</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>441</td>
</tr>
<tr>
<td>64</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>356</td>
</tr>
</tbody>
</table>

Table 9.45: Transitions on recursion levels other than the main recursion level for the communication heavy (10 loop iterations) benchmark.

<table>
<thead>
<tr>
<th>Nr threads</th>
<th>Large sep.</th>
<th>Medium sep.</th>
<th>Low sep.</th>
<th>Heterogeneous</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>20</td>
<td>26</td>
<td>418</td>
<td>21</td>
</tr>
<tr>
<td>4</td>
<td>400</td>
<td>520</td>
<td>51308</td>
<td>98</td>
</tr>
<tr>
<td>8</td>
<td>692800</td>
<td>900640</td>
<td>107308856</td>
<td>593</td>
</tr>
<tr>
<td>16</td>
<td>61121808</td>
<td>61166805</td>
<td>—</td>
<td>18460</td>
</tr>
<tr>
<td>32</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>1068537</td>
</tr>
<tr>
<td>64</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>4989179</td>
</tr>
</tbody>
</table>
Table 9.44: Transitions on the main recursion level for the communication heavy (10 loop iterations) benchmark.

<table>
<thead>
<tr>
<th>Nr threads</th>
<th>Large sep.</th>
<th>Medium sep.</th>
<th>Low sep.</th>
<th>Heterogeneous</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>53</td>
<td>53</td>
<td>53</td>
<td>53</td>
</tr>
<tr>
<td>2</td>
<td>53</td>
<td>53</td>
<td>53</td>
<td>85</td>
</tr>
<tr>
<td>4</td>
<td>53</td>
<td>53</td>
<td>53</td>
<td>133</td>
</tr>
<tr>
<td>8</td>
<td>53</td>
<td>53</td>
<td>13</td>
<td>189</td>
</tr>
<tr>
<td>16</td>
<td>33</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>32</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>64</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 9.45: Transitions on recursion levels other than the main recursion level for the communication heavy (10 loop iterations) benchmark.

<table>
<thead>
<tr>
<th>Nr threads</th>
<th>Large sep.</th>
<th>Medium sep.</th>
<th>Low sep.</th>
<th>Heterogeneous</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>20</td>
<td>26</td>
<td>418</td>
<td>21</td>
</tr>
<tr>
<td>4</td>
<td>40</td>
<td>520</td>
<td>51308</td>
<td>98</td>
</tr>
<tr>
<td>8</td>
<td>692</td>
<td>800</td>
<td>900640</td>
<td>1073088</td>
</tr>
<tr>
<td>16</td>
<td>6112</td>
<td>61805</td>
<td>-</td>
<td>18460</td>
</tr>
<tr>
<td>32</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1068537</td>
</tr>
<tr>
<td>64</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>4989179</td>
</tr>
</tbody>
</table>

Figure 9.46: Transitions on the main recursion level for the communication benchmarks.
Figure 9.47: Transitions on recursion levels other than the main recursion level for the communication benchmarks.
### Table 9.48: Transitions on the main recursion level for the synchronization light (1 loop iteration) benchmark.

<table>
<thead>
<tr>
<th>Nr threads</th>
<th>Large sep.</th>
<th>Medium sep.</th>
<th>Low sep.</th>
<th>Heterogeneous</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>31</td>
<td>31</td>
<td>23</td>
<td>17</td>
</tr>
<tr>
<td>4</td>
<td>623</td>
<td>575</td>
<td>431</td>
<td>47</td>
</tr>
<tr>
<td>8</td>
<td>633346</td>
<td>612979</td>
<td>784165</td>
<td>510</td>
</tr>
<tr>
<td>16</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>57864</td>
</tr>
</tbody>
</table>

### Table 9.49: Transitions on the main recursion level for the synchronization medium heavy (5 loop iterations) benchmark.

<table>
<thead>
<tr>
<th>Nr threads</th>
<th>Large sep.</th>
<th>Medium sep.</th>
<th>Low sep.</th>
<th>Heterogeneous</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>28</td>
<td>28</td>
<td>28</td>
<td>28</td>
</tr>
<tr>
<td>2</td>
<td>119</td>
<td>2107</td>
<td>4803</td>
<td>374</td>
</tr>
<tr>
<td>4</td>
<td>55007</td>
<td>802657</td>
<td>520477</td>
<td>853722</td>
</tr>
<tr>
<td>8</td>
<td>231140</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>

### Table 9.50: Transitions on the main recursion level for the synchronization heavy (10 loop iterations) benchmark.

<table>
<thead>
<tr>
<th>Nr threads</th>
<th>Large sep.</th>
<th>Medium sep.</th>
<th>Low sep.</th>
<th>Heterogeneous</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>53</td>
<td>53</td>
<td>53</td>
<td>53</td>
</tr>
<tr>
<td>2</td>
<td>247</td>
<td>1267501</td>
<td>1083311</td>
<td>204176</td>
</tr>
<tr>
<td>4</td>
<td>1023203</td>
<td>—</td>
<td>—</td>
<td>451143</td>
</tr>
</tbody>
</table>
Synchronization setups

The derived numbers of transitions on the main recursion level for the benchmarking setups which are based on the synchronization program type, as presented in Table 9.5, are presented in Tables 9.48, 9.49 and 9.50, and visualized in Figure 9.51. There are no transitions on recursion levels other than the main recursion level for any of the setups (which is natural since no communication occurs between the threads in this benchmark program type).

As can be seen, there is a large, exponential increase in the needed number of transitions, both as the number of loop iterations increases and as the number of threads increases in the benchmark program. This is since all possible orders in which the threads can acquire the given lock must be considered. From the benchmark program code, it is easy to see that the possible number of orders logically increases exponentially with the number of threads (the actual complexity is factorial), and possibly also with the number of loop iterations depending on the actual timing of executing the statements in the loop.

Communication and synchronization setups

The derived numbers of transitions on the main recursion level for the benchmarking setups which are based on the communication and synchronization program type, as presented in Table 9.6, are presented in Tables 9.52, 9.54 and 9.56, and visualized in Figure 9.58. The derived numbers of transitions on all recursion levels other than the main recursion level are presented in Tables 9.53, 9.55 and 9.57, and visualized in Figure 9.59.

As can be seen, there is a large, exponential increase in the needed number of transitions, both as the number of loop iterations increases and as the number of threads increases in the benchmark program. This is since all possible orders in which the threads can acquire the given lock must be considered, and all possible values a load-statement in some thread might see must be derived, which means that the complexities of the pure communication and synchronization cases are merged.

Figure 9.51: Transitions on the main recursion level for the synchronization benchmarks.
Synchronization setups

The derived numbers of transitions on the main recursion level for the benchmarking setups which are based on the synchronization program type, as presented in Table 9.5, are presented in Tables 9.48, 9.49 and 9.50, and visualized in Figure 9.51. There are no transitions on recursion levels other than the main recursion level for any of the setups (which is natural since no communication occurs between the threads in this benchmark program type).

As can be seen, there is a large, exponential increase in the needed number of transitions, both as the number of loop iterations increases and as the number of threads increases in the benchmark program. This is since all possible orders in which the threads can acquire the given lock must be considered. From the benchmark program code, it is easy to see that the possible number of orders logically increases exponentially with the number of threads (the actual complexity is factorial), and possibly also with the number of loop iterations depending on the actual timing of executing the statements in the loop.

Communication and synchronization setups

The derived numbers of transitions on the main recursion level for the benchmarking setups which are based on the communication and synchronization program type, as presented in Table 9.6, are presented in Tables 9.52, 9.54 and 9.56, and visualized in Figure 9.58. The derived numbers of transitions on all recursion levels other than the main recursion level are presented in Tables 9.53, 9.55 and 9.57, and visualized in Figure 9.59.

As can be seen, there is a large, exponential increase in the needed number of transitions, both as the number of loop iterations increases and as the number of threads increases in the benchmark program. This is since all possible orders in which the threads can acquire the given lock must be considered, and all possible values a load-statement in some thread might see must be derived, which means that the complexities of the pure communication and synchronization cases are merged.
Table 9.52: Transitions on the main recursion level for the communication and synchronization light (1 loop iteration) benchmark.

<table>
<thead>
<tr>
<th>Nr threads</th>
<th>Large sep.</th>
<th>Medium sep.</th>
<th>Low sep.</th>
<th>Heterogeneous</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>39</td>
<td>39</td>
<td>33</td>
<td>24</td>
</tr>
<tr>
<td>4</td>
<td>751</td>
<td>727</td>
<td>571</td>
<td>188</td>
</tr>
<tr>
<td>8</td>
<td>273256</td>
<td>231281</td>
<td>215148</td>
<td>53406</td>
</tr>
<tr>
<td>16</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>34191</td>
</tr>
</tbody>
</table>

Table 9.53: Transitions on recursion levels other than the main recursion level for the communication and synchronization light (1 loop iteration) benchmark.

<table>
<thead>
<tr>
<th>Nr threads</th>
<th>Large sep.</th>
<th>Medium sep.</th>
<th>Low sep.</th>
<th>Heterogeneous</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>72</td>
<td>120</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>35218</td>
<td>33732</td>
<td>6819</td>
</tr>
<tr>
<td>16</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>397</td>
</tr>
</tbody>
</table>

Table 9.54: Transitions on the main recursion level for the communication and synchronization medium heavy (5 loop iterations) benchmark.

<table>
<thead>
<tr>
<th>Nr threads</th>
<th>Large sep.</th>
<th>Medium sep.</th>
<th>Low sep.</th>
<th>Heterogeneous</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>38</td>
<td>38</td>
<td>38</td>
<td>38</td>
</tr>
<tr>
<td>2</td>
<td>241</td>
<td>3879</td>
<td>6641</td>
<td>2830</td>
</tr>
<tr>
<td>4</td>
<td>717429</td>
<td>439281</td>
<td>382985</td>
<td>571675</td>
</tr>
</tbody>
</table>

Table 9.55: Transitions on recursion levels other than the main recursion level for the communication and synchronization medium heavy (5 loop iterations) benchmark.

<table>
<thead>
<tr>
<th>Nr threads</th>
<th>Large sep.</th>
<th>Medium sep.</th>
<th>Low sep.</th>
<th>Heterogeneous</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>548</td>
<td>1004</td>
<td>372</td>
</tr>
<tr>
<td>4</td>
<td>37852</td>
<td>65071</td>
<td>58064</td>
<td>63549</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>35218</td>
<td>33732</td>
<td>6819</td>
</tr>
<tr>
<td>16</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>397</td>
</tr>
</tbody>
</table>

Table 9.56: Transitions on the main recursion level for the communication and synchronization heavy (10 loop iterations) benchmark.

<table>
<thead>
<tr>
<th>Nr threads</th>
<th>Large sep.</th>
<th>Medium sep.</th>
<th>Low sep.</th>
<th>Heterogeneous</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>73</td>
<td>73</td>
<td>73</td>
<td>73</td>
</tr>
<tr>
<td>2</td>
<td>31241</td>
<td>983844</td>
<td>905082</td>
<td>1059223</td>
</tr>
<tr>
<td>4</td>
<td>379781</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
</tbody>
</table>

Table 9.57: Transitions on recursion levels other than the main recursion level for the communication and synchronization heavy (10 loop iterations) benchmark.

<table>
<thead>
<tr>
<th>Nr threads</th>
<th>Large sep.</th>
<th>Medium sep.</th>
<th>Low sep.</th>
<th>Heterogeneous</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>4124</td>
<td>140067</td>
<td>127995</td>
<td>150681</td>
</tr>
<tr>
<td>4</td>
<td>10479</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
</tbody>
</table>
### Table 9.52: Transitions on the main recursion level for the communication and synchronization light (1 loop iteration) benchmark.

<table>
<thead>
<tr>
<th>Nr threads</th>
<th>Large sep.</th>
<th>Medium sep.</th>
<th>Low sep.</th>
<th>Heterogeneous</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>39</td>
<td>39</td>
<td>33</td>
<td>24</td>
</tr>
<tr>
<td>4</td>
<td>751</td>
<td>727</td>
<td>571</td>
<td>188</td>
</tr>
<tr>
<td>8</td>
<td>273256</td>
<td>231281</td>
<td>215148</td>
<td>53406</td>
</tr>
<tr>
<td>16</td>
<td>34191</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Table 9.53: Transitions on recursion levels other than the main recursion level for the communication and synchronization light (1 loop iteration) benchmark.

<table>
<thead>
<tr>
<th>Nr threads</th>
<th>Large sep.</th>
<th>Medium sep.</th>
<th>Low sep.</th>
<th>Heterogeneous</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>48</td>
<td>1004</td>
<td>372</td>
</tr>
<tr>
<td>4</td>
<td>37852</td>
<td>65071</td>
<td>58064</td>
<td>63549</td>
</tr>
</tbody>
</table>

### Table 9.54: Transitions on the main recursion level for the communication and synchronization medium heavy (5 loop iterations) benchmark.

<table>
<thead>
<tr>
<th>Nr threads</th>
<th>Large sep.</th>
<th>Medium sep.</th>
<th>Low sep.</th>
<th>Heterogeneous</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>38</td>
<td>38</td>
<td>38</td>
<td>38</td>
</tr>
<tr>
<td>2</td>
<td>241</td>
<td>3879</td>
<td>6641</td>
<td>2830</td>
</tr>
<tr>
<td>4</td>
<td>717429</td>
<td>439281</td>
<td>382985</td>
<td>571675</td>
</tr>
</tbody>
</table>

### Table 9.55: Transitions on recursion levels other than the main recursion level for the communication and synchronization medium heavy (5 loop iterations) benchmark.

<table>
<thead>
<tr>
<th>Nr threads</th>
<th>Large sep.</th>
<th>Medium sep.</th>
<th>Low sep.</th>
<th>Heterogeneous</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>548</td>
<td>1004</td>
<td>372</td>
</tr>
<tr>
<td>4</td>
<td>37852</td>
<td>65071</td>
<td>58064</td>
<td>63549</td>
</tr>
</tbody>
</table>

### Table 9.56: Transitions on the main recursion level for the communication and synchronization heavy (10 loop iterations) benchmark.

<table>
<thead>
<tr>
<th>Nr threads</th>
<th>Large sep.</th>
<th>Medium sep.</th>
<th>Low sep.</th>
<th>Heterogeneous</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>73</td>
<td>73</td>
<td>73</td>
<td>73</td>
</tr>
<tr>
<td>2</td>
<td>31241</td>
<td>983844</td>
<td>905082</td>
<td>1059223</td>
</tr>
<tr>
<td>4</td>
<td>379781</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Table 9.57: Transitions on recursion levels other than the main recursion level for the communication and synchronization heavy (10 loop iterations) benchmark.

<table>
<thead>
<tr>
<th>Nr threads</th>
<th>Large sep.</th>
<th>Medium sep.</th>
<th>Low sep.</th>
<th>Heterogeneous</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>4124</td>
<td>140067</td>
<td>127995</td>
<td>150681</td>
</tr>
<tr>
<td>4</td>
<td>10479</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Figure 9.58: Transitions on the main recursion level for the communication and synchronization benchmarks.
Figure 9.58: Transitions on the main recursion level for the communication and synchronization benchmarks.

(a) Timing model with large separation.

(b) Timing model with medium separation.

(c) Timing model with low separation.

(d) Heterogeneous timing model.

Figure 9.59: Transitions on recursion levels other than the main recursion level for the communication and synchronization benchmarks.
Table 9.60: Transitions on the main recursion level for the well-structured data parallel benchmark.

<table>
<thead>
<tr>
<th>Nr threads</th>
<th>Large sep.</th>
<th>Medium sep.</th>
<th>Low sep.</th>
<th>Heterogeneous</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2070</td>
<td>2070</td>
<td>2070</td>
<td>2070</td>
</tr>
<tr>
<td>2</td>
<td>2936</td>
<td>2942</td>
<td>2914</td>
<td>3195</td>
</tr>
<tr>
<td>4</td>
<td>47866</td>
<td>60568</td>
<td>79668</td>
<td>11980</td>
</tr>
<tr>
<td>8</td>
<td>698458</td>
<td>810784</td>
<td>507968</td>
<td>951582</td>
</tr>
</tbody>
</table>

Table 9.61: Transitions on recursion levels other than the main recursion level for the well-structured data parallel benchmark.

<table>
<thead>
<tr>
<th>Nr threads</th>
<th>Large sep.</th>
<th>Medium sep.</th>
<th>Low sep.</th>
<th>Heterogeneous</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>118</td>
<td>106</td>
<td>30</td>
<td>61</td>
</tr>
<tr>
<td>4</td>
<td>8004</td>
<td>13752</td>
<td>5040</td>
<td>1086</td>
</tr>
<tr>
<td>8</td>
<td>3598</td>
<td>7692</td>
<td>24286</td>
<td>142710</td>
</tr>
</tbody>
</table>

Well-structured data parallel setups

The derived numbers of transitions on the main recursion level for the benchmarking setups which are based on the well-structured data parallel program type, as presented in Table 9.7, are presented in Table 9.60 and visualized in Figure 9.62. The derived numbers of transitions on all recursion levels other than the main recursion level are presented in Table 9.61 and visualized in Figure 9.63.

As can be seen, there is a large, non-linear increase in the needed number of transitions on the main recursion level (and also on all other recursion levels) as the number of threads in the program increases. Unfortunately, the complexity of the non-linear growth seems to be exponential even though the program is structured in an advantageous manner.
The derived numbers of transitions on the main recursion level for the benchmarking setups which are based on the well-structured data parallel program type, as presented in Table 9.7, are presented in Table 9.60 and visualized in Figure 9.62. The derived numbers of transitions on all recursion levels other than the main recursion level are presented in Table 9.61 and visualized in Figure 9.63.

As can be seen, there is a large, non-linear increase in the needed number of transitions on the main recursion level (and also on all other recursion levels) as the number of threads in the program increases. Unfortunately, the complexity of the non-linear growth seems to be exponential even though the program is structured in an advantageous manner.

Figure 9.62: Transitions on the main recursion level for the well-structured data parallel benchmark.
In this section, the derived bounds on the BCET and WCET for the different benchmarking setups are presented. The derived bounds are collected and presented in two tables based on which benchmark program type, as presented in Section 9.1, is considered. The tables present the derived bounds on the BCET and WCET, respectively, for each benchmark program type.

The entries in the tables are the derived execution time bounds for each benchmark timing model, as presented in Section 9.2, and each number of threads used in the benchmark program. I.e., each entry in the tables corresponds to the derived execution time bound for a unique benchmarking setup.

The derived bounds on the BCET and WCET for each benchmark program type are also visualized in a graph for each benchmark timing model; i.e., four graphs, one for each benchmark timing model, are presented for each benchmark program type. In the graphs, the values 0 and $\infty$ are not shown due to the limitations of the logarithmic axes.

Short independent threads setups

The derived execution time bounds for the benchmarking setups which are based on the short independent threads program type, as presented in Table 9.1, are presented in Tables 9.64 and 9.65, and visualized in Figure 9.66.

As can be seen, just as for the derived numbers of transitions presented in Section 9.5, the derived execution time bounds for the homogeneous benchmark timing models are the same regardless of how many threads the program consists of. This is since the threads in the program are identical and the timing of a statement is the same regardless of which thread executes it.

For the heterogeneous benchmark timing model, there is a linear increase in the derived execution time bounds. This is since the heterogeneous benchmark timing model simply scales the execution time of a statement based on which thread executes the statement.

It can be noticed that the difference between the derived bounds on the BCET and the WCET grows when the precision of the timing model decreases.

Figure 9.63: Transitions on recursion levels other than the main recursion level for the well-structured data parallel benchmark.
9.6 Derived Execution Time Bounds

In this section, the derived bounds on the BCET and WCET for the different benchmarking setups are presented. The derived bounds are collected and presented in two tables based on which benchmark program type, as presented in Section 9.1, is considered. The tables present the derived bounds on the BCET and WCET, respectively, for each benchmark program type.

The entries in the tables are the derived execution time bounds for each benchmark timing model, as presented in Section 9.2, and each number of threads used in the benchmark program. I.e., each entry in the tables corresponds to the derived execution time bound for a unique benchmarking setup.

The derived bounds on the BCET and WCET for each benchmark program type are also visualized in a graph for each benchmark timing model; i.e., four graphs, one for each benchmark timing model, are presented for each benchmark program type. In the graphs, the values 0 and $\infty$ are not shown due to the limitations of the logarithmic axes.

Short independent threads setups

The derived execution time bounds for the benchmarking setups which are based on the short independent threads program type, as presented in Table 9.1, are presented in Tables 9.64 and 9.65, and visualized in Figure 9.66.

As can be seen, just as for the derived numbers of transitions presented in Section 9.5, the derived execution time bounds for the homogeneous benchmark timing models are the same regardless of how many threads the program consists of. This is since the threads in the program are identical and the timing of a statement is the same regardless of which thread executes it.

For the heterogeneous benchmark timing model, there is a linear increase in the derived execution time bounds. This is since the heterogeneous benchmark timing model simply scales the execution time of a statement based on which thread executes the statement.

It can be noticed that the difference between the derived bounds on the BCET and the WCET grows when the precision of the timing model decreases.
Table 9.64: Derived bounds on the BCET for the short independent threads benchmark.

<table>
<thead>
<tr>
<th>Nr threads</th>
<th>Large sep.</th>
<th>Medium sep.</th>
<th>Low sep.</th>
<th>Heterogeneous</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2414</td>
<td>2414</td>
<td>2113</td>
<td>2414</td>
</tr>
<tr>
<td>2</td>
<td>2414</td>
<td>2414</td>
<td>2113</td>
<td>4828</td>
</tr>
<tr>
<td>4</td>
<td>2414</td>
<td>2414</td>
<td>2113</td>
<td>9656</td>
</tr>
<tr>
<td>8</td>
<td>2414</td>
<td>2414</td>
<td>2113</td>
<td>19312</td>
</tr>
<tr>
<td>16</td>
<td>2414</td>
<td>2414</td>
<td>2113</td>
<td>38624</td>
</tr>
<tr>
<td>32</td>
<td>2414</td>
<td>2414</td>
<td>2113</td>
<td>77248</td>
</tr>
<tr>
<td>64</td>
<td>2414</td>
<td>2414</td>
<td>2113</td>
<td>154496</td>
</tr>
<tr>
<td>128</td>
<td>2414</td>
<td>2414</td>
<td>2113</td>
<td>308992</td>
</tr>
<tr>
<td>256</td>
<td>2414</td>
<td>2414</td>
<td>2113</td>
<td>617984</td>
</tr>
<tr>
<td>512</td>
<td>2414</td>
<td>2414</td>
<td>2113</td>
<td>1235968</td>
</tr>
<tr>
<td>1024</td>
<td>2414</td>
<td>2414</td>
<td>2113</td>
<td>2471936</td>
</tr>
</tbody>
</table>

Table 9.65: Derived bounds on the WCET for the short independent threads benchmark.

<table>
<thead>
<tr>
<th>Nr threads</th>
<th>Large sep.</th>
<th>Medium sep.</th>
<th>Low sep.</th>
<th>Heterogeneous</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2514</td>
<td>2614</td>
<td>3118</td>
<td>2514</td>
</tr>
<tr>
<td>2</td>
<td>2514</td>
<td>2614</td>
<td>3118</td>
<td>5028</td>
</tr>
<tr>
<td>4</td>
<td>2514</td>
<td>2614</td>
<td>3118</td>
<td>10056</td>
</tr>
<tr>
<td>8</td>
<td>2514</td>
<td>2614</td>
<td>3118</td>
<td>20112</td>
</tr>
<tr>
<td>16</td>
<td>2514</td>
<td>2614</td>
<td>3118</td>
<td>40224</td>
</tr>
<tr>
<td>32</td>
<td>2514</td>
<td>2614</td>
<td>3118</td>
<td>80448</td>
</tr>
<tr>
<td>64</td>
<td>2514</td>
<td>2614</td>
<td>3118</td>
<td>160896</td>
</tr>
<tr>
<td>128</td>
<td>2514</td>
<td>2614</td>
<td>3118</td>
<td>321792</td>
</tr>
<tr>
<td>256</td>
<td>2514</td>
<td>2614</td>
<td>3118</td>
<td>643584</td>
</tr>
<tr>
<td>512</td>
<td>2514</td>
<td>2614</td>
<td>3118</td>
<td>1287168</td>
</tr>
<tr>
<td>1024</td>
<td>2514</td>
<td>2614</td>
<td>3118</td>
<td>2574336</td>
</tr>
</tbody>
</table>
Table 9.64: Derived bounds on the BCET for the short independent threads benchmark.

<table>
<thead>
<tr>
<th>Nr threads</th>
<th>Large sep.</th>
<th>Medium sep.</th>
<th>Low sep.</th>
<th>Heterogeneous</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2414</td>
<td>2414</td>
<td>2113</td>
<td>2414</td>
</tr>
<tr>
<td>2</td>
<td>2414</td>
<td>2414</td>
<td>2113</td>
<td>4828</td>
</tr>
<tr>
<td>4</td>
<td>2414</td>
<td>2414</td>
<td>2113</td>
<td>9656</td>
</tr>
<tr>
<td>8</td>
<td>2414</td>
<td>2414</td>
<td>2113</td>
<td>19312</td>
</tr>
<tr>
<td>16</td>
<td>2414</td>
<td>2414</td>
<td>2113</td>
<td>38624</td>
</tr>
<tr>
<td>32</td>
<td>2414</td>
<td>2414</td>
<td>2113</td>
<td>77248</td>
</tr>
<tr>
<td>64</td>
<td>2414</td>
<td>2414</td>
<td>2113</td>
<td>154496</td>
</tr>
<tr>
<td>128</td>
<td>2414</td>
<td>2414</td>
<td>2113</td>
<td>308992</td>
</tr>
<tr>
<td>256</td>
<td>2414</td>
<td>2414</td>
<td>2113</td>
<td>617984</td>
</tr>
<tr>
<td>512</td>
<td>2414</td>
<td>2414</td>
<td>2113</td>
<td>1235968</td>
</tr>
<tr>
<td>1024</td>
<td>2414</td>
<td>2414</td>
<td>2113</td>
<td>2471936</td>
</tr>
</tbody>
</table>

Table 9.65: Derived bounds on the WCET for the short independent threads benchmark.

<table>
<thead>
<tr>
<th>Nr threads</th>
<th>Large sep.</th>
<th>Medium sep.</th>
<th>Low sep.</th>
<th>Heterogeneous</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2514</td>
<td>2614</td>
<td>3118</td>
<td>2514</td>
</tr>
<tr>
<td>2</td>
<td>2514</td>
<td>2614</td>
<td>3118</td>
<td>5028</td>
</tr>
<tr>
<td>4</td>
<td>2514</td>
<td>2614</td>
<td>3118</td>
<td>10056</td>
</tr>
<tr>
<td>8</td>
<td>2514</td>
<td>2614</td>
<td>3118</td>
<td>20112</td>
</tr>
<tr>
<td>16</td>
<td>2514</td>
<td>2614</td>
<td>3118</td>
<td>40224</td>
</tr>
<tr>
<td>32</td>
<td>2514</td>
<td>2614</td>
<td>3118</td>
<td>80448</td>
</tr>
<tr>
<td>64</td>
<td>2514</td>
<td>2614</td>
<td>3118</td>
<td>160896</td>
</tr>
<tr>
<td>128</td>
<td>2514</td>
<td>2614</td>
<td>3118</td>
<td>321792</td>
</tr>
<tr>
<td>256</td>
<td>2514</td>
<td>2614</td>
<td>3118</td>
<td>643584</td>
</tr>
<tr>
<td>512</td>
<td>2514</td>
<td>2614</td>
<td>3118</td>
<td>1287168</td>
</tr>
<tr>
<td>1024</td>
<td>2514</td>
<td>2614</td>
<td>3118</td>
<td>2574336</td>
</tr>
</tbody>
</table>

(a) Timing model with large separation.
(b) Timing model with medium separation.
(c) Timing model with low separation.
(d) Heterogeneous timing model.

Figure 9.66: Derived execution time bounds for the short independent threads benchmark.
Table 9.67: Derived bounds on the BCET for the long independent threads benchmark.

<table>
<thead>
<tr>
<th>Nr threads</th>
<th>Large sep.</th>
<th>Medium sep.</th>
<th>Low sep.</th>
<th>Heterogeneous</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2400024</td>
<td>2400024</td>
<td>2100021</td>
<td>2400024</td>
</tr>
<tr>
<td>2</td>
<td>4800024</td>
<td>4800024</td>
<td>4200021</td>
<td>9600048</td>
</tr>
<tr>
<td>4</td>
<td>9600024</td>
<td>9600024</td>
<td>8400021</td>
<td>38400096</td>
</tr>
<tr>
<td>8</td>
<td>19200024</td>
<td>19200024</td>
<td>16800021</td>
<td>153600192</td>
</tr>
<tr>
<td>16</td>
<td>38400024</td>
<td>38400024</td>
<td>33600021</td>
<td>614400384</td>
</tr>
<tr>
<td>32</td>
<td>76800024</td>
<td>76800024</td>
<td>67200021</td>
<td>0</td>
</tr>
<tr>
<td>64</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>–</td>
</tr>
</tbody>
</table>

Table 9.68: Derived bounds on the WCET for the long independent threads benchmark.

<table>
<thead>
<tr>
<th>Nr threads</th>
<th>Large sep.</th>
<th>Medium sep.</th>
<th>Low sep.</th>
<th>Heterogeneous</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2500025</td>
<td>2600026</td>
<td>3100031</td>
<td>2500025</td>
</tr>
<tr>
<td>2</td>
<td>5000025</td>
<td>5200026</td>
<td>6200031</td>
<td>10000050</td>
</tr>
<tr>
<td>4</td>
<td>10000025</td>
<td>10400026</td>
<td>12400031</td>
<td>40000100</td>
</tr>
<tr>
<td>8</td>
<td>20000025</td>
<td>20800026</td>
<td>24800031</td>
<td>160000200</td>
</tr>
<tr>
<td>16</td>
<td>40000025</td>
<td>41600026</td>
<td>49600031</td>
<td>640000400</td>
</tr>
<tr>
<td>32</td>
<td>80000025</td>
<td>83200026</td>
<td>99200031</td>
<td>∞</td>
</tr>
<tr>
<td>64</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>–</td>
</tr>
</tbody>
</table>
9.6 Derived Execution Time Bounds

Table 9.67: Derived bounds on the BCET for the long independent threads benchmark.

<table>
<thead>
<tr>
<th>Nr threads</th>
<th>Large sep.</th>
<th>Medium sep.</th>
<th>Low sep.</th>
<th>Heterogeneous</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2400024</td>
<td>2400024</td>
<td>2100021</td>
<td>2400024</td>
</tr>
<tr>
<td>2</td>
<td>4800024</td>
<td>4800024</td>
<td>4200021</td>
<td>9600048</td>
</tr>
<tr>
<td>4</td>
<td>9600024</td>
<td>9600024</td>
<td>8400021</td>
<td>38400096</td>
</tr>
<tr>
<td>8</td>
<td>19200024</td>
<td>19200024</td>
<td>16800021</td>
<td>153600192</td>
</tr>
<tr>
<td>16</td>
<td>38400024</td>
<td>38400024</td>
<td>33600021</td>
<td>614400384</td>
</tr>
<tr>
<td>32</td>
<td>76800024</td>
<td>76800024</td>
<td>67200021</td>
<td>0</td>
</tr>
<tr>
<td>64</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 9.68: Derived bounds on the WCET for the long independent threads benchmark.

<table>
<thead>
<tr>
<th>Nr threads</th>
<th>Large sep.</th>
<th>Medium sep.</th>
<th>Low sep.</th>
<th>Heterogeneous</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2500025</td>
<td>2600026</td>
<td>3100031</td>
<td>2500025</td>
</tr>
<tr>
<td>2</td>
<td>5000025</td>
<td>5200026</td>
<td>6200031</td>
<td>10000050</td>
</tr>
<tr>
<td>4</td>
<td>10000025</td>
<td>10400026</td>
<td>12400031</td>
<td>40000100</td>
</tr>
<tr>
<td>8</td>
<td>20000025</td>
<td>20800026</td>
<td>24800031</td>
<td>160000200</td>
</tr>
<tr>
<td>16</td>
<td>40000025</td>
<td>41600026</td>
<td>49600031</td>
<td>640000400</td>
</tr>
<tr>
<td>32</td>
<td>80000025</td>
<td>83200026</td>
<td>99200031</td>
<td>∞</td>
</tr>
<tr>
<td>64</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
</tbody>
</table>

Figure 9.69: Derived execution time bounds for the long independent threads benchmark.
Table 9.70: Derived bounds on the BCET for the branching heavy benchmark.

<table>
<thead>
<tr>
<th>Nr threads</th>
<th>Large sep.</th>
<th>Medium sep.</th>
<th>Low sep.</th>
<th>Heterogeneous</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>537</td>
<td>537</td>
<td>227</td>
<td>537</td>
</tr>
<tr>
<td>2</td>
<td>609</td>
<td>609</td>
<td>291</td>
<td>1242</td>
</tr>
<tr>
<td>4</td>
<td>765</td>
<td>765</td>
<td>429</td>
<td>3204</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>0</td>
<td>—</td>
<td>9288</td>
</tr>
<tr>
<td>16</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>30096</td>
</tr>
<tr>
<td>32</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 9.71: Derived bounds on the WCET for the branching heavy benchmark.

<table>
<thead>
<tr>
<th>Nr threads</th>
<th>Large sep.</th>
<th>Medium sep.</th>
<th>Low sep.</th>
<th>Heterogeneous</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>544</td>
<td>568</td>
<td>670</td>
<td>563</td>
</tr>
<tr>
<td>2</td>
<td>630</td>
<td>656</td>
<td>774</td>
<td>1298</td>
</tr>
<tr>
<td>4</td>
<td>815</td>
<td>846</td>
<td>998</td>
<td>3336</td>
</tr>
<tr>
<td>8</td>
<td>∞</td>
<td>∞</td>
<td>—</td>
<td>9632</td>
</tr>
<tr>
<td>16</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>31104</td>
</tr>
<tr>
<td>32</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>∞</td>
</tr>
</tbody>
</table>

**Long independent threads setups**

The derived execution time bounds for the benchmarking setups which are based on the long independent threads program type, as presented in Table 9.2, are presented in Tables 9.67 and 9.68, and visualized in Figure 9.69.

As can be seen, there is a linear growth in the derived execution time bounds for all the homogeneous benchmark timing models, which is due to the definition of the benchmark program (i.e., the number of statements to execute in the loop for each thread is proportional to the number of the executing thread). The steeper increase for the heterogeneous benchmark timing model is due to the scaling of the execution time of each statement based on the number of the thread that is executing the statement.
Table 9.70: Derived bounds on the BCET for the branching heavy benchmark.

<table>
<thead>
<tr>
<th>Nr threads</th>
<th>Large sep.</th>
<th>Medium sep.</th>
<th>Low sep.</th>
<th>Heterogeneous</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>537</td>
<td>537</td>
<td>227</td>
<td>537</td>
</tr>
<tr>
<td>2</td>
<td>609</td>
<td>609</td>
<td>291</td>
<td>1242</td>
</tr>
<tr>
<td>4</td>
<td>765</td>
<td>765</td>
<td>429</td>
<td>3204</td>
</tr>
<tr>
<td>8</td>
<td>9288</td>
<td></td>
<td></td>
<td>9288</td>
</tr>
<tr>
<td>16</td>
<td>30096</td>
<td></td>
<td></td>
<td>30096</td>
</tr>
<tr>
<td>32</td>
<td>0</td>
<td></td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>

Table 9.71: Derived bounds on the WCET for the branching heavy benchmark.

<table>
<thead>
<tr>
<th>Nr threads</th>
<th>Large sep.</th>
<th>Medium sep.</th>
<th>Low sep.</th>
<th>Heterogeneous</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>544</td>
<td>568</td>
<td>670</td>
<td>563</td>
</tr>
<tr>
<td>2</td>
<td>630</td>
<td>656</td>
<td>774</td>
<td>1298</td>
</tr>
<tr>
<td>4</td>
<td>815</td>
<td>846</td>
<td>998</td>
<td>3336</td>
</tr>
<tr>
<td>8</td>
<td>∞</td>
<td></td>
<td></td>
<td>∞</td>
</tr>
<tr>
<td>16</td>
<td>∞</td>
<td></td>
<td></td>
<td>∞</td>
</tr>
<tr>
<td>32</td>
<td>∞</td>
<td></td>
<td></td>
<td>∞</td>
</tr>
</tbody>
</table>

The derived execution time bounds for the benchmarking setups which are based on the long independent threads program type, as presented in Table 9.2, are presented in Tables 9.67 and 9.68, and visualized in Figure 9.69.

As can be seen, there is a linear growth in the derived execution time bounds for all the homogeneous benchmark timing models, which is due to the definition of the benchmark program (i.e., the number of statements to execute in the loop for each thread is proportional to the number of the executing thread). The steeper increase for the heterogeneous benchmark timing model is due to the scaling of the execution time of each statement based on the number of the thread that is executing the statement.

Figure 9.72: Derived execution time bounds for the branching heavy benchmark.
Branching heavy setups

The derived execution time bounds for the benchmarking setups which are based on the branching heavy program type, as presented in Table 9.3, are presented in Tables 9.70 and 9.71, and visualized in Figure 9.72. Remember that there is no result due to technical reasons for the terminating instance (8 threads) for the homogeneous benchmark timing model with low timing separation of statements since the termination occurred due to spawning too many ERLANG processes and thus reaching the ERLANG process number system limit.

As can be seen, the derived execution time bounds increase with the number of threads in the program. This behavior is due to the definition of the program (i.e., the maximum possible value of the register \(a\) increases as the number of threads in the program increases).

Communication setups

The derived execution time bounds for the benchmarking setups which are based on the communication program type, as presented in Table 9.4, are presented in Tables 9.73, 9.74, 9.75, 9.76, 9.77 and 9.78, and visualized in Figure 9.79.

As can be seen, for each homogeneous benchmark timing model there is no difference between the derived execution time bounds regardless of how many threads the program consists of. This is since a constant number of statements is executed by each thread in the benchmark program, regardless of the total number of defined threads.

For the heterogeneous benchmark timing model, there is a linear increase in the derived execution time bounds. This is since the execution time of a statement is directly proportional to the number of the thread executing the statement.

It can also be seen that the difference between the derived values for the upper and lower bounds on the execution time become more separated as the precision of the timing model becomes lower (i.e., as the separation of the execution times of statements becomes lower).
Branching heavy setups

The derived execution time bounds for the benchmarking setups which are based on the branching heavy program type, as presented in Table 9.3, are presented in Tables 9.70 and 9.71, and visualized in Figure 9.72. Remember that there is no result due to technical reasons for the terminating instance (8 threads) for the homogeneous benchmark timing model with low timing separation of statements since the termination occurred due to spawning too many ERLANG processes and thus reaching the ERLANG process number system limit.

As can be seen, the derived execution time bounds increase with the number of threads in the program. This behavior is due to the definition of the program (i.e., the maximum possible value of the register $a$ increases as the number of threads in the program increases).

Communication setups

The derived execution time bounds for the benchmarking setups which are based on the communication program type, as presented in Table 9.4, are presented in Tables 9.73, 9.74, 9.75, 9.76, 9.77 and 9.78, and visualized in Figure 9.79.

As can be seen, for each homogeneous benchmark timing model there is no difference between the derived execution time bounds regardless of how many threads the program consists of. This is since a constant number of statements is executed by each thread in the benchmark program, regardless of the total number of defined threads.

For the heterogeneous benchmark timing model, there is a linear increase in the derived execution time bounds. This is since the execution time of a statement is directly proportional to the number of the thread executing the statement.

It can also be seen that the difference between the derived values for the upper and lower bounds on the execution time become more separated as the precision of the timing model becomes lower (i.e., as the separation of the execution times of statements becomes lower).

### Table 9.73: Derived bounds on the BCET for the communication light (1 loop iteration) benchmark.

<table>
<thead>
<tr>
<th>Nr threads</th>
<th>Large sep.</th>
<th>Medium sep.</th>
<th>Low sep.</th>
<th>Heterogeneous</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>441</td>
<td>441</td>
<td>144</td>
<td>441</td>
</tr>
<tr>
<td>2</td>
<td>441</td>
<td>441</td>
<td>144</td>
<td>882</td>
</tr>
<tr>
<td>4</td>
<td>441</td>
<td>441</td>
<td>144</td>
<td>1764</td>
</tr>
<tr>
<td>8</td>
<td>441</td>
<td>441</td>
<td>144</td>
<td>3528</td>
</tr>
<tr>
<td>16</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>7056</td>
</tr>
<tr>
<td>32</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>14112</td>
</tr>
<tr>
<td>64</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>28224</td>
</tr>
<tr>
<td>128</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>56448</td>
</tr>
<tr>
<td>256</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>0</td>
</tr>
</tbody>
</table>

### Table 9.74: Derived bounds on the WCET for the communication light (1 loop iteration) benchmark.

<table>
<thead>
<tr>
<th>Nr threads</th>
<th>Large sep.</th>
<th>Medium sep.</th>
<th>Low sep.</th>
<th>Heterogeneous</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>444</td>
<td>464</td>
<td>549</td>
<td>463</td>
</tr>
<tr>
<td>2</td>
<td>444</td>
<td>464</td>
<td>549</td>
<td>926</td>
</tr>
<tr>
<td>4</td>
<td>444</td>
<td>464</td>
<td>549</td>
<td>1852</td>
</tr>
<tr>
<td>8</td>
<td>444</td>
<td>464</td>
<td>549</td>
<td>3704</td>
</tr>
<tr>
<td>16</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>7408</td>
</tr>
<tr>
<td>32</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>14816</td>
</tr>
<tr>
<td>64</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>29632</td>
</tr>
<tr>
<td>128</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>59264</td>
</tr>
<tr>
<td>256</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>$\infty$</td>
</tr>
</tbody>
</table>
Table 9.75: Derived bounds on the BCET for the communication medium heavy (5 loop iterations) benchmark.

<table>
<thead>
<tr>
<th>Nr threads</th>
<th>Large sep.</th>
<th>Medium sep.</th>
<th>Low sep.</th>
<th>Heterogeneous</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2149</td>
<td>2149</td>
<td>668</td>
<td>2149</td>
</tr>
<tr>
<td>2</td>
<td>2149</td>
<td>2149</td>
<td>668</td>
<td>4298</td>
</tr>
<tr>
<td>4</td>
<td>2149</td>
<td>2149</td>
<td>668</td>
<td>8596</td>
</tr>
<tr>
<td>8</td>
<td>2149</td>
<td>2149</td>
<td>668</td>
<td>17192</td>
</tr>
<tr>
<td>16</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>34384</td>
</tr>
<tr>
<td>32</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>68768</td>
</tr>
<tr>
<td>64</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>137536</td>
</tr>
<tr>
<td>128</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 9.76: Derived bounds on the WCET for the communication medium heavy (5 loop iterations) benchmark.

<table>
<thead>
<tr>
<th>Nr threads</th>
<th>Large sep.</th>
<th>Medium sep.</th>
<th>Low sep.</th>
<th>Heterogeneous</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2164</td>
<td>2264</td>
<td>2673</td>
<td>2259</td>
</tr>
<tr>
<td>2</td>
<td>2164</td>
<td>2264</td>
<td>2673</td>
<td>4518</td>
</tr>
<tr>
<td>4</td>
<td>2164</td>
<td>2264</td>
<td>2673</td>
<td>9036</td>
</tr>
<tr>
<td>8</td>
<td>2164</td>
<td>2264</td>
<td>2673</td>
<td>18072</td>
</tr>
<tr>
<td>16</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>36144</td>
</tr>
<tr>
<td>32</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>72288</td>
</tr>
<tr>
<td>64</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>144576</td>
</tr>
<tr>
<td>128</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>∞</td>
</tr>
</tbody>
</table>
### Table 9.75: Derived bounds on the BCET for the communication medium heavy (5 loop iterations) benchmark.

<table>
<thead>
<tr>
<th>Nr threads</th>
<th>Large sep.</th>
<th>Medium sep.</th>
<th>Low sep.</th>
<th>Heterogeneous</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2149</td>
<td>2149</td>
<td>668</td>
<td>2149</td>
</tr>
<tr>
<td>2</td>
<td>2149</td>
<td>2149</td>
<td>668</td>
<td>4298</td>
</tr>
<tr>
<td>4</td>
<td>2149</td>
<td>2149</td>
<td>668</td>
<td>8596</td>
</tr>
<tr>
<td>8</td>
<td>2149</td>
<td>2149</td>
<td>668</td>
<td>17192</td>
</tr>
<tr>
<td>16</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>34384</td>
</tr>
<tr>
<td>32</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>68768</td>
</tr>
<tr>
<td>64</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>137536</td>
</tr>
</tbody>
</table>

### Table 9.76: Derived bounds on the WCET for the communication medium heavy (5 loop iterations) benchmark.

<table>
<thead>
<tr>
<th>Nr threads</th>
<th>Large sep.</th>
<th>Medium sep.</th>
<th>Low sep.</th>
<th>Heterogeneous</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2164</td>
<td>2264</td>
<td>2673</td>
<td>2259</td>
</tr>
<tr>
<td>2</td>
<td>2164</td>
<td>2264</td>
<td>2673</td>
<td>4518</td>
</tr>
<tr>
<td>4</td>
<td>2164</td>
<td>2264</td>
<td>2673</td>
<td>9036</td>
</tr>
<tr>
<td>8</td>
<td>2164</td>
<td>2264</td>
<td>2673</td>
<td>18072</td>
</tr>
<tr>
<td>16</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>36144</td>
</tr>
<tr>
<td>32</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>72288</td>
</tr>
<tr>
<td>64</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>144576</td>
</tr>
</tbody>
</table>

### Table 9.77: Derived bounds on the BCET for the communication heavy (10 loop iterations) benchmark.

<table>
<thead>
<tr>
<th>Nr threads</th>
<th>Large sep.</th>
<th>Medium sep.</th>
<th>Low sep.</th>
<th>Heterogeneous</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4284</td>
<td>4284</td>
<td>1323</td>
<td>4284</td>
</tr>
<tr>
<td>2</td>
<td>4284</td>
<td>4284</td>
<td>1323</td>
<td>8568</td>
</tr>
<tr>
<td>4</td>
<td>4284</td>
<td>4284</td>
<td>1323</td>
<td>17136</td>
</tr>
<tr>
<td>8</td>
<td>4284</td>
<td>4284</td>
<td>0</td>
<td>34272</td>
</tr>
<tr>
<td>16</td>
<td>0</td>
<td>0</td>
<td>–</td>
<td>68544</td>
</tr>
<tr>
<td>32</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>137088</td>
</tr>
<tr>
<td>64</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>0</td>
</tr>
</tbody>
</table>

### Table 9.78: Derived bounds on the WCET for the communication heavy (10 loop iterations) benchmark.

<table>
<thead>
<tr>
<th>Nr threads</th>
<th>Large sep.</th>
<th>Medium sep.</th>
<th>Low sep.</th>
<th>Heterogeneous</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4314</td>
<td>4514</td>
<td>5328</td>
<td>4504</td>
</tr>
<tr>
<td>2</td>
<td>4314</td>
<td>4514</td>
<td>5328</td>
<td>9008</td>
</tr>
<tr>
<td>4</td>
<td>4314</td>
<td>4514</td>
<td>5328</td>
<td>18016</td>
</tr>
<tr>
<td>8</td>
<td>4314</td>
<td>4514</td>
<td>∞</td>
<td>36032</td>
</tr>
<tr>
<td>16</td>
<td>∞</td>
<td>∞</td>
<td>–</td>
<td>72064</td>
</tr>
<tr>
<td>32</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>144128</td>
</tr>
<tr>
<td>64</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>∞</td>
</tr>
</tbody>
</table>
Figure 9.79: Derived execution time bounds for the communication benchmarks.
Table 9.80: Derived bounds on the BCET for the synchronization light (1 loop iteration) benchmark.

<table>
<thead>
<tr>
<th>Nr threads</th>
<th>Large sep.</th>
<th>Medium sep.</th>
<th>Low sep.</th>
<th>Heterogeneous</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>288</td>
<td>288</td>
<td>214</td>
<td>288</td>
</tr>
<tr>
<td>2</td>
<td>438</td>
<td>438</td>
<td>284</td>
<td>576</td>
</tr>
<tr>
<td>4</td>
<td>738</td>
<td>738</td>
<td>424</td>
<td>1764</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4440</td>
</tr>
<tr>
<td>16</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 9.81: Derived bounds on the WCET for the synchronization light (1 loop iteration) benchmark.

<table>
<thead>
<tr>
<th>Nr threads</th>
<th>Large sep.</th>
<th>Medium sep.</th>
<th>Low sep.</th>
<th>Heterogeneous</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>290</td>
<td>310</td>
<td>410</td>
<td>309</td>
</tr>
<tr>
<td>2</td>
<td>441</td>
<td>470</td>
<td>771</td>
<td>618</td>
</tr>
<tr>
<td>4</td>
<td>743</td>
<td>790</td>
<td>1493</td>
<td>1887</td>
</tr>
<tr>
<td>8</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>7592</td>
</tr>
<tr>
<td>16</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>∞</td>
</tr>
</tbody>
</table>

Table 9.82: Derived bounds on the BCET for the synchronization medium heavy (5 loop iterations) benchmark.

<table>
<thead>
<tr>
<th>Nr threads</th>
<th>Large sep.</th>
<th>Medium sep.</th>
<th>Low sep.</th>
<th>Heterogeneous</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1384</td>
<td>1384</td>
<td>1018</td>
<td>1384</td>
</tr>
<tr>
<td>2</td>
<td>1534</td>
<td>1534</td>
<td>1088</td>
<td>2768</td>
</tr>
<tr>
<td>4</td>
<td>2884</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
</tbody>
</table>
Table 9.83: Derived bounds on the WCET for the synchronization medium heavy (5 loop iterations) benchmark.

<table>
<thead>
<tr>
<th>Nr threads</th>
<th>Large sep.</th>
<th>Medium sep.</th>
<th>Low sep.</th>
<th>Heterogeneous</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1394</td>
<td>1494</td>
<td>1978</td>
<td>1489</td>
</tr>
<tr>
<td>2</td>
<td>1545</td>
<td>2598</td>
<td>3783</td>
<td>3938</td>
</tr>
<tr>
<td>4</td>
<td>4034</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>8</td>
<td>∞</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
</tbody>
</table>

Table 9.84: Derived bounds on the BCET for the synchronization heavy (10 loop iterations) benchmark.

<table>
<thead>
<tr>
<th>Nr threads</th>
<th>Large sep.</th>
<th>Medium sep.</th>
<th>Low sep.</th>
<th>Heterogeneous</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2754</td>
<td>2754</td>
<td>2023</td>
<td>2754</td>
</tr>
<tr>
<td>2</td>
<td>2904</td>
<td>0</td>
<td>0</td>
<td>5508</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>–</td>
<td>–</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 9.85: Derived bounds on the WCET for the synchronization heavy (10 loop iterations) benchmark.

<table>
<thead>
<tr>
<th>Nr threads</th>
<th>Large sep.</th>
<th>Medium sep.</th>
<th>Low sep.</th>
<th>Heterogeneous</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2774</td>
<td>2974</td>
<td>3938</td>
<td>2964</td>
</tr>
<tr>
<td>2</td>
<td>2925</td>
<td>∞</td>
<td>∞</td>
<td>8409</td>
</tr>
<tr>
<td>4</td>
<td>∞</td>
<td>–</td>
<td>–</td>
<td>∞</td>
</tr>
</tbody>
</table>
9.6 Derived Execution Time Bounds

Table 9.83: Derived bounds on the WCET for the synchronization medium heavy (5 loop iterations) benchmark.

<table>
<thead>
<tr>
<th>Nr threads</th>
<th>Large sep.</th>
<th>Medium sep.</th>
<th>Low sep.</th>
<th>Heterogeneous</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1394</td>
<td>1494</td>
<td>1978</td>
<td>1489</td>
</tr>
<tr>
<td>2</td>
<td>1545</td>
<td>2598</td>
<td>3783</td>
<td>3938</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 9.84: Derived bounds on the BCET for the synchronization heavy (10 loop iterations) benchmark.

<table>
<thead>
<tr>
<th>Nr threads</th>
<th>Large sep.</th>
<th>Medium sep.</th>
<th>Low sep.</th>
<th>Heterogeneous</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2754</td>
<td>2754</td>
<td>2023</td>
<td>2754</td>
</tr>
<tr>
<td>2</td>
<td>2904</td>
<td>0</td>
<td>0</td>
<td>5508</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 9.85: Derived bounds on the WCET for the synchronization heavy (10 loop iterations) benchmark.

<table>
<thead>
<tr>
<th>Nr threads</th>
<th>Large sep.</th>
<th>Medium sep.</th>
<th>Low sep.</th>
<th>Heterogeneous</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2774</td>
<td>2974</td>
<td>3938</td>
<td>2964</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 9.86: Derived execution time bounds for the synchronization benchmarks.
Synchronization setups

The derived execution time bounds for the benchmarking setups which are based on the synchronization program type, as presented in Table 9.5, are presented in Tables 9.80, 9.81, 9.82, 9.83, 9.84 and 9.85, and visualized in Figure 9.86.

As can be seen, there seems to be a (non-linear) increase in the derived upper bound on the execution time as the number of threads in the benchmark program increases. This is since the execution of the synchronization regions is sequentialized; i.e., only one of the threads in the benchmark program can execute the lock- and unlock-statements at a time—all other threads must wait for their turns.

It can also be seen that the difference between the derived values for the upper and lower bounds on the execution time become more separated as the precision of the timing model becomes lower (i.e., as the separation of the execution times of statements becomes lower).

Communication and synchronization setups

The derived execution time bounds for the benchmarking setups which are based on the communication and synchronization program type, as presented in Table 9.6, are presented in Tables 9.87, 9.88, 9.89, 9.90, 9.91 and 9.92, and visualized in Figure 9.93.

As can be seen, the derived execution time bounds increase as the number of threads in the benchmark program increases. This is since the threads synchronize on a mutually exclusive code section.

It can also be seen that the difference between the derived values for the upper and lower bounds on the execution time become more separated as the precision of the timing model becomes lower (i.e., as the separation of the execution times of statements becomes lower).
The derived execution time bounds for the benchmarking setups which are based on the synchronization program type, as presented in Table 9.5, are presented in Tables 9.80, 9.81, 9.82, 9.83, 9.84 and 9.85, and visualized in Figure 9.86.

As can be seen, there seems to be a (non-linear) increase in the derived upper bound on the execution time as the number of threads in the benchmark program increases. This is since the execution of the synchronization regions is sequentialized; i.e., only one of the threads in the benchmark program can execute the lock- and unlock-statements at a time—all other threads must wait for their turns.

It can also be seen that the difference between the derived values for the upper and lower bounds on the execution time become more separated as the precision of the timing model becomes lower (i.e., as the separation of the execution times of statements becomes lower).

The derived execution time bounds for the benchmarking setups which are based on the communication and synchronization program type, as presented in Table 9.6, are presented in Tables 9.87, 9.88, 9.89, 9.90, 9.91 and 9.92, and visualized in Figure 9.93.

As can be seen, the derived execution time bounds increase as the number of threads in the benchmark program increases. This is since the threads synchronize on a mutually exclusive code section.

It can also be seen that the difference between the derived values for the upper and lower bounds on the execution time become more separated as the precision of the timing model becomes lower (i.e., as the separation of the execution times of statements becomes lower).

---

Table 9.87: Derived bounds on the BCET for the communication and synchronization light (1 loop iteration) benchmark.

<table>
<thead>
<tr>
<th>Nr threads</th>
<th>Large sep.</th>
<th>Medium sep.</th>
<th>Low sep.</th>
<th>Heterogeneous</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>691</td>
<td>691</td>
<td>324</td>
<td>691</td>
</tr>
<tr>
<td>2</td>
<td>1291</td>
<td>1291</td>
<td>504</td>
<td>1721</td>
</tr>
<tr>
<td>4</td>
<td>2491</td>
<td>2297</td>
<td>864</td>
<td>5305</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>18393</td>
</tr>
<tr>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 9.88: Derived bounds on the WCET for the communication and synchronization light (1 loop iteration) benchmark.

<table>
<thead>
<tr>
<th>Nr threads</th>
<th>Large sep.</th>
<th>Medium sep.</th>
<th>Low sep.</th>
<th>Heterogeneous</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>695</td>
<td>734</td>
<td>910</td>
<td>733</td>
</tr>
<tr>
<td>2</td>
<td>1299</td>
<td>1374</td>
<td>1771</td>
<td>2106</td>
</tr>
<tr>
<td>4</td>
<td>2507</td>
<td>2708</td>
<td>3493</td>
<td>6772</td>
</tr>
<tr>
<td>8</td>
<td>∞</td>
<td>∞∞</td>
<td>∞∞</td>
<td>24148</td>
</tr>
<tr>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td>∞</td>
</tr>
</tbody>
</table>

Table 9.89: Derived bounds on the BCET for the communication and synchronization medium heavy (5 loop iterations) benchmark.

<table>
<thead>
<tr>
<th>Nr threads</th>
<th>Large sep.</th>
<th>Medium sep.</th>
<th>Low sep.</th>
<th>Heterogeneous</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3399</td>
<td>3399</td>
<td>1568</td>
<td>3399</td>
</tr>
<tr>
<td>2</td>
<td>5799</td>
<td>5339</td>
<td>1944</td>
<td>7757</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Table 9.90: Derived bounds on the WCET for the communication and synchronization medium heavy (5 loop iterations) benchmark.

<table>
<thead>
<tr>
<th>Nr threads</th>
<th>Large sep.</th>
<th>Medium sep.</th>
<th>Low sep.</th>
<th>Heterogeneous</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3419</td>
<td>3614</td>
<td>4478</td>
<td>3609</td>
</tr>
<tr>
<td>2</td>
<td>5986</td>
<td>7004</td>
<td>8783</td>
<td>10738</td>
</tr>
<tr>
<td>4</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
</tbody>
</table>

Table 9.91: Derived bounds on the BCET for the communication and synchronization heavy (10 loop iterations) benchmark.

<table>
<thead>
<tr>
<th>Nr threads</th>
<th>Large sep.</th>
<th>Medium sep.</th>
<th>Low sep.</th>
<th>Heterogeneous</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6784</td>
<td>6784</td>
<td>3123</td>
<td>6784</td>
</tr>
<tr>
<td>2</td>
<td>10829</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
</tbody>
</table>

Table 9.92: Derived bounds on the WCET for the communication and synchronization heavy (10 loop iterations) benchmark.

<table>
<thead>
<tr>
<th>Nr threads</th>
<th>Large sep.</th>
<th>Medium sep.</th>
<th>Low sep.</th>
<th>Heterogeneous</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6824</td>
<td>7214</td>
<td>8938</td>
<td>7204</td>
</tr>
<tr>
<td>2</td>
<td>12646</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>4</td>
<td>∞</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
</tbody>
</table>
Table 9.90: Derived bounds on the WCET for the communication and synchronization medium heavy (5 loop iterations) benchmark.

<table>
<thead>
<tr>
<th>Nr threads</th>
<th>Large sep.</th>
<th>Medium sep.</th>
<th>Low sep.</th>
<th>Heterogeneous</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3419</td>
<td>3614</td>
<td>4478</td>
<td>3609</td>
</tr>
<tr>
<td>2</td>
<td>5986</td>
<td>7004</td>
<td>8783</td>
<td>10738</td>
</tr>
<tr>
<td>4</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
</tbody>
</table>

Table 9.91: Derived bounds on the BCET for the communication and synchronization heavy (10 loop iterations) benchmark.

<table>
<thead>
<tr>
<th>Nr threads</th>
<th>Large sep.</th>
<th>Medium sep.</th>
<th>Low sep.</th>
<th>Heterogeneous</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6784</td>
<td>6784</td>
<td>3123</td>
<td>6784</td>
</tr>
<tr>
<td>2</td>
<td>10829</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>−−</td>
<td>−−</td>
<td>−−</td>
</tr>
</tbody>
</table>

Table 9.92: Derived bounds on the WCET for the communication and synchronization heavy (10 loop iterations) benchmark.

<table>
<thead>
<tr>
<th>Nr threads</th>
<th>Large sep.</th>
<th>Medium sep.</th>
<th>Low sep.</th>
<th>Heterogeneous</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6824</td>
<td>7214</td>
<td>8938</td>
<td>7204</td>
</tr>
<tr>
<td>2</td>
<td>12646</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>4</td>
<td>∞</td>
<td>−−</td>
<td>−−</td>
<td>−−</td>
</tr>
</tbody>
</table>

Figure 9.93: Derived execution time bounds for the communication and synchronization benchmarks.
Table 9.94: Derived bounds on the BCET for the well-structured data parallel benchmark.

<table>
<thead>
<tr>
<th>Nr threads</th>
<th>Large sep.</th>
<th>Medium sep.</th>
<th>Low sep.</th>
<th>Heterogeneous</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>12529</td>
<td>12529</td>
<td>10359</td>
<td>12529</td>
</tr>
<tr>
<td>2</td>
<td>7156</td>
<td>7156</td>
<td>5557</td>
<td>13282</td>
</tr>
<tr>
<td>4</td>
<td>5242</td>
<td>5242</td>
<td>3441</td>
<td>14788</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 9.95: Derived bounds on the WCET for the well-structured data parallel benchmark.

<table>
<thead>
<tr>
<th>Nr threads</th>
<th>Large sep.</th>
<th>Medium sep.</th>
<th>Low sep.</th>
<th>Heterogeneous</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>13758</td>
<td>14598</td>
<td>16971</td>
<td>13824</td>
</tr>
<tr>
<td>2</td>
<td>8155</td>
<td>8650</td>
<td>10168</td>
<td>15424</td>
</tr>
<tr>
<td>4</td>
<td>6357</td>
<td>6738</td>
<td>8082</td>
<td>19584</td>
</tr>
<tr>
<td>8</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
</tbody>
</table>

**Well-structured data parallel setups**

The derived execution time bounds for the benchmarking setups which are based on the well-structured data parallel program type, as presented in Table 9.7, are presented in Tables 9.94 and 9.95 and visualized in Figure 9.96.

As can be seen, the derived execution time bounds decrease as the number of threads in the program increases for the homogeneous benchmark timing models. This is since the threads are cooperating and sharing the iterations of a loop between each other.

For the heterogeneous benchmark timing model, there is a small increase in the derived execution time bounds as the number of threads in the program increases. This is since the execution time of a statement is scaled with the number of the thread that is executing it, which means that the thread with the largest number will (probably) have the longest execution time since the number of iterations of the loop is split equally among the cooperating threads.
Table 9.94: Derived bounds on the BCET for the well-structured data parallel benchmark.

<table>
<thead>
<tr>
<th>Nr threads</th>
<th>Large sep.</th>
<th>Medium sep.</th>
<th>Low sep.</th>
<th>Heterogeneous</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>12529</td>
<td>12529</td>
<td>10359</td>
<td>12529</td>
</tr>
<tr>
<td>2</td>
<td>7156</td>
<td>7156</td>
<td>5557</td>
<td>13282</td>
</tr>
<tr>
<td>4</td>
<td>5242</td>
<td>5242</td>
<td>3441</td>
<td>14788</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 9.95: Derived bounds on the WCET for the well-structured data parallel benchmark.

<table>
<thead>
<tr>
<th>Nr threads</th>
<th>Large sep.</th>
<th>Medium sep.</th>
<th>Low sep.</th>
<th>Heterogeneous</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>13758</td>
<td>14598</td>
<td>16971</td>
<td>13824</td>
</tr>
<tr>
<td>2</td>
<td>8155</td>
<td>8650</td>
<td>10168</td>
<td>15424</td>
</tr>
<tr>
<td>4</td>
<td>6357</td>
<td>6738</td>
<td>8082</td>
<td>19584</td>
</tr>
<tr>
<td>8</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
</tbody>
</table>

Well-structured data parallel setups

The derived execution time bounds for the benchmarking setups which are based on the well-structured data parallel program type, as presented in Table 9.7, are presented in Tables 9.94 and 9.95 and visualized in Figure 9.96.

As can be seen, the derived execution time bounds decrease as the number of threads in the program increases for the homogeneous benchmark timing models. This is since the threads are cooperating and sharing the iterations of a loop between each other.

For the heterogeneous benchmark timing model, there is a small increase in the derived execution time bounds as the number of threads in the program increases. This is since the execution time of a statement is scaled with the number of the thread that is executing it, which means that the thread with the largest number will (probably) have the longest execution time since the number of iterations of the loop is split equally among the cooperating threads.

Figure 9.96: Derived execution time bounds for the well-structured data parallel benchmark.
9.7 Summary

Naturally, the analysis running time grows as the number of statements to execute in the analyzed benchmark program increases. However, the measured analysis running times indicate that the analysis also has a non-linear complexity relation to the number of threads in the analyzed program. This applies even to the simplest cases where the threads are independent from one another and the number of derived transitions is constant in relation to the number of threads in the analyzed program. Note that the structure of Algorithm 6.1 is such that each thread in a configuration is examined several times when looking at the different configuration properties (i.e., final, deadlocked, timed-out and valid). This could to some extent explain the non-linear complexity of the analysis in relation to the analyzed number of threads.

As the analyzed system becomes more complex and incorporates communication and/or synchronization (i.e., when the threads of the analyzed program become dependent on each other), both the measured analysis running times and the derived number of transitions indicate that the analysis has an exponential complexity relation to the number of threads, and possibly also to the amount of communication and/or synchronization in the analyzed program.

The precision in the timing model does not seem to affect the analysis running time as long as the threads in the analyzed program are independent from each other. If the threads are dependent on each other, then low precision in the timing model increases the number of possible conflicts between the threads and thus also increases the size of the state space, which in turn increases the number of derived transitions and thus also the needed analysis running time.

Executing threads with different speeds (cf. the heterogeneous timing model defined in Table 9.11) can result in less complexity when analyzing communicating, and mostly identical, threads since a smaller number of possible values of a read variable exist due to the timing separation of the threads. This could mean that the analyzed program can contain a larger number of threads and still be analyzable within reasonable time and memory limits.

Analyzing a program consisting of synchronizing threads seems to always be of exponential complexity in relation to the number of synchronizing threads. This is since all possible orders in which threads could acquire the given lock must be considered and applies even to well-structured data parallel programs in which threads cooperate on executing loop iterations and then synchronize once after the loop.
Chapter 10

Conclusions

In this chapter, some distinguishing properties of the presented analysis are discussed. The evaluation of the analysis implementation is used to draw conclusions on the functionality and complexity of the analysis. Feedback on the research questions and issues to be further considered and investigated are also given.

10.1 The Underlying Architecture

The analysis is defined for an arbitrary underlying architecture (that is, however, restricted to the constraints given by Assumptions 4.1 and 4.3). The actual underlying system could be an operating system as well as raw hardware as long as both thread-private and globally shared memory, and some form of synchronization primitive, correlating to the description given in the beginning of Chapter 4 are provided. The assumed architecture should be fairly realistic since any mature operating system and any common (single- or multi-core) CPU provides the described features at some abstraction level. For example, any real-time operating system should provide spin-locks for thread synchronization and any CPU instruction set should provide the ability to lock the system bus to provide atomic execution of a set of machine operations (since one single instruction of the instruction set often is mapped to a set of machine instructions).

The lock- and unlock-statements could be used to model the LOCK prefix in the x86 instruction set. This prefix is used for asserting atomic execution
of an instruction [63, 113]. The lock- and unlock-statements also trivially correspond to higher level spin-locking primitives, such as those provided by the POSIX thread library [18, 61].

Many of the principles applied in the analysis presented in Chapters 5 and 6 to solve the problems arising from abstracting time using intervals are also applicable to analysis of systems with distributed address spaces. If considering processes on one and the same CPU, then communication between these processes is often implemented using a memory buffer which is then to be considered as shared memory. This means that the same principles as those presented in this thesis would be applicable to such an analysis. If communication is performed using, for example, message passing and “Any”-communication is available (i.e., several processes could send a message to a given receiving process and/or several processes could receive a message sent from a given process), this would also require some form of prediction of what values could be transferred between processes.

The necessity of allowing \( \text{TIME}(c, T) = 0 \) for some configuration, \( c \in \text{Conf} \), and thread, \( T \in \text{Thrd} \), is apparent when considering the following case. If mutual exclusion is inherent in some instruction of the modeled instruction set, for example in store, then the lock- and unlock-statements could be regarded as macros without timing that should encapsulate all store-statements in a program.

PPL is designed to bring the focus of the analysis to thread synchronization and global data flow. The method presented in this thesis might have to be extended in order to cover all the aspects of a real instruction set, such as those of for example the ARM [7] or PowerPC [64] architectures. This will be further investigated.

If limiting the register (and variable) sizes, the architecture would become more realistic. However, wrap-around effects could render loops nonterminating in the abstract case even if this would not concretely occur. See Section 10.3 for a discussion on how nonterminating cases can be, and are, handled.

## 10.2 Algorithmic Structure & Complexity

The analysis presented in Chapters 5 and 6 is based on synchronously advancing the threads of a program between their respective program points while keeping the threads fairly synchronized in time (cf. Algorithm 6.1 and Tables 5.12 and 5.13). The advantage of this approach (i.e., abstracting time using intervals) in conjunction with the defined domain for variable states (cf. Sec-
tation 5.5) is that a relatively high precision is achieved. And, when \(|\textbf{Thrd}| = 1\), the analysis result will be comparable to that of the corresponding sequential analysis [46]. Another advantage is that the time-complexity of the analysis is more dependent on the number of program points in each thread than on the timing behavior of the program, compared to stepping through strict timing events, like in the concrete semantics.

Keeping the threads fairly synchronized in the analysis is also an advantage when considering its memory-complexity. Keeping the threads synchronized means that the write history for any thread on any variable will always be as small as possible since writes become outdated after a minimal amount of steps in the analysis and are then trimmed away from the history. In other words, the write history for any thread on any variable will never be larger than absolutely necessary.

Maintaining a write history for each thread on each variable is expected to be necessary in order to keep the over-approximations at a reasonable level. Trimming is an advantage for the memory-complexity as discussed above, but could of course be a disadvantage for the time-complexity, especially if the analyzed program consists of many variables and many write-intensive threads. As shown in Chapter 9, there are indeed complexity issues when analyzing communicating threads (cf. the communication benchmarking setups).

The definition of the abstract state for locks contains some concrete parts (e.g., the owners of the locks). This is necessary since too much precision would be lost, and the timing approximations would become useless (i.e., too over-approximate), otherwise. However, this causes complexity problems as previously predicted [48] and shown in the evaluation of the presented analysis (cf. Chapter 9, especially the synchronization benchmarking setups). The result of not abstracting some parts of a state is that (at least) all the concrete counterparts must be evaluated. No reasonably precise abstractions of the parts of the lock states that are currently kept concrete have been found.

It should be apparent that a given (abstract) configuration could result in two or more configurations for each thread issuing an `if`- or a `lock`-statement in a transition (cf. Tables 5.12 and 5.13). Merging of configurations at specific merge points could be performed to reduce the complexity of the analysis. Using the Control Flow Graph (CFG) of the program, suitable merge-points within each thread can be found [43]. Typically, such points have multiple incoming edges. However, with the current level of abstraction, it might be difficult to have merging occur when analyzing some programs. This is since all the concrete parts (i.e., the program counters, lock owners, etc.) must be equal between the configurations to merge.
It is very important to note that several configurations that lack valid concrete counterparts (cf. Definition 4.4) are added to the worklist for several situations. One such situation is when one sole thread issues lock lck for some free lock, lck ∈ Lck, in a transition. A unique transition (i.e., resulting configuration) is possible for each thread that might issue lock lck somewhere in the program. A new configuration for each such thread, where the given thread is the new owner of lck, will thus be derived.

Consider the situation depicted in Figure 10.1 (cf. the example in Section 7.3). T1 is obviously the thread issuing lock lck first in any considered case (remember that DLLOCK was defined in Algorithm 5.11). However, two new configurations are derived on the transition; one where T1 is the new owner of lck and one where T2 is the new owner of lck. Obviously, only the configuration for which T1 is the owner of lck has valid concrete counterparts since T2 will not acquire lck before some other thread (i.e., T1) is guaranteed to have acquired lck. Thus, the case that T2 is the new owner of lck will eventually be discontinued since the lock is not acquired by T2 before the deadline expires (cf. Algorithm 6.10).

Another such situation will result for the program described in Figure 10.2a, assuming that the timings of the respective first lock-statement in the two threads overlap (cf. the example presented in Section 7.2). (Note that the given code is guaranteed not to deadlock, provided that both threads eventually release the two locks again.) Assume that the program is described by ∈ Conf and that pcT1 = pcT2 = 1. The resulting lock-owner assignments (i.e., configurations) are given in Figure 10.2b. Obviously, only c′11 and c′22 have valid concrete counterparts. c′12 and c′21 will be discontinued (i.e., removed from the worklist) since there is a cycle in the dependency graph containing at least one lock (here, that lock is lck′) that has the state unlocked (cf. Algorithm 6.10). If c′12 and c′21 were not discontinued, then the analysis would itself deadlock.
An important point to notice from the above discussion is that well-structured concurrent programs are less complex to analyze compared to less well-structured concurrent programs [78]. The threads in a well-structured program typically work as much as possible on local data and do not synchronize more than is absolutely necessary. However, as shown in Chapter 9 (cf. the well-structured data parallel benchmarking setups), the complexity of analyzing well-structured data parallel programs is still very high with the current level of abstraction.

Another important point to notice is that the complexity is lowered by keeping a high precision in the calculation of the accumulated abstract execution time for threads issuing lock-statements. Since $\text{Time} = \text{Intv}$, a high precision in this calculation will give a narrow accumulated execution time (i.e., a small difference between the lower and upper bounds of the execution time). This will lead to that a minimum number of states need to be explored since the timing of individual threads will not overlap more than necessary. Of course, this part of the complexity is also dependent on the precision of $\text{ABSTIME}$; i.e., the accuracy in the abstracted model of the underlying architecture.

The precision of the timing model of the underlying architecture is a major topic for current research within timing analysis of parallel systems. For multicore CPUs with shared buses and memories, it can be very difficult to predict...
the timing behavior of memory accesses due to the lack of knowledge about what is simultaneously occurring on the other cores. To account for this lack of knowledge, information about the hardware state could be added to the configurations. This could be especially advantageous when analyzing programs consisting of threads that execute in a very predictable manner. Adding hardware state information to the configurations would render the analysis more spatially complex but could greatly reduce the time-complexity since the underlying timing model could be made more precise.

10.3 Nonterminating Transition Sequences

As previously discussed, ISDEADLOCK catches some configurations that will never reach the final state (cf. Algorithm 6.8). However, it is not guaranteed to identify all such configurations. This means that the analysis could actually deadlock for some cases that ISDEADLOCK misses to identify as never reaching the final state. The corresponding can be said if ISVALID wrongly identifies a configuration as valid (cf. Algorithm 6.10).

Infinite loops are recognized by ISTIMEOUT(\(\tilde{c}, \tilde{t}_{\infty}\)), given that abstract time moves forward and the timeout is finite; i.e., it cannot be that \(0 \in \text{ABSTIME}(\tilde{c}', T)\) for all \(\tilde{c}' \in \text{Conf}\) occurring in the loop in \(T \in \text{Thrd}\) and \(\max(\gamma_t(\tilde{t}_{\infty})) = \infty\). If ABSTIME includes 0 for all statements of an infinite loop in some thread, then the algorithm presented in Algorithm 6.1 will not terminate.

To avoid part of this problem, another timeout variable is added in the implementation of the analysis (cf. Chapter 8). This timeout is used to identify that the upper bound of a single thread’s accumulated abstract execution time has reached a limit. However, this does not resolve the case that all \(\tilde{c}' \in \text{Conf}\) in the loop are such that \(\text{ABSTIME}(\tilde{c}', T) = [0, 0]\).

To address this case, a transition counter can be used. There could be one counter for each thread individually and/or one counter for all threads combined. Or, one counter for transitions on the main recursion level and one counter for transitions on all other recursion levels. The counter(s) could either count all transitions or only transitions that are consecutively done in \([0, 0]\) amount of abstract time, depending on whether a second timeout is used. When the counter reaches a specific limit, the configuration could be considered to be timed-out, which means that the corresponding transition sequence could be of infinite length. In the implementation (cf. Chapter 8), there is one counter for transitions on the main recursion level and one counter for transitions on all
other recursion levels. These counters keep track of all transitions, regardless of their timing properties.

Two additional timeouts are implemented. The first regards the analysis running time and simply aborts the analysis when the tool has been running for a specified amount of time. The second aborts the analysis when the tool uses more than a specified amount of memory.

It is important to notice that even if all concrete transition sequences given some initial configuration terminate, all abstract transition sequences resulting from the corresponding abstract initial configuration are not guaranteed to terminate. This is due to over-approximations inherent in the abstraction of the PPL semantics. Thus, all the complications discussed above can occur in the abstract case even if they do not occur in the concrete case.

10.4 The Research Questions

**Question 1:** "Can safe and tight bounds on the execution time of concurrent programs consisting of dependent tasks be derived?"

It has been shown that a technique referred to as abstract execution that is based on abstract interpretation can be used for deriving safe timing bound estimates for a given program and timing model. The resulting tightness of the estimates depends both on the precision of the used abstract domains, the precision of the timing analysis itself and the precision of the timing model of the underlying architecture. Low precision in the timing model results in an analysis result with low precision even when there is relatively high precision in the abstract domains and in the analysis itself.

**Question 2:** "How can the timing of communicating tasks be safely and tightly estimated?"

To safely estimate the timing of communicating tasks, memory values must be safely approximated at all times since they might affect the control flow of threads. The challenge here lies in approximating shared memory values since several threads might produce and/or consume them.

A domain for abstract variable states that collects all possible values for each global memory position is used. This ensures that safe and tight values of shared (and also not shared) memory positions can be safely and tightly approximated at all times.
If a shared memory position is read by some thread, the analysis first derives all possible values for that memory position before the thread finally reads it. This ensures that the timing behavior of communicating threads is safely and tightly estimated.

**Question 3:** “How can the timing of synchronizing tasks be safely and tightly estimated?”

To safely estimate the timing behavior of synchronizing tasks, all possible execution interleavings (i.e., orders in which threads acquire locks) must inevitably be considered. Currently, this is achieved by spawning a new configuration for each possible interleaving. This strategy increases the risk of path explosion (the risk is exponentially proportional to the number of synchronizing threads). Further abstraction needs to be incorporated to decrease this risk.

However, to achieve a reasonably tight estimate on the timing behavior of synchronizing tasks, the level of abstraction on lock states must be kept fairly low. There is a risk that the analysis result becomes more or less useless when the level of abstraction becomes too high. This is a severe trade-off that must be carefully balanced.

**Question 4:** “How can programs suffering from deadlocks and other types of nonterminating programs be handled?”

Since abstract execution suffers the risk of not terminating if a nonterminating situation in the analyzed program is encountered, several techniques to increase the probability of termination have been incorporated into the analysis (as discussed in the previous section). One such technique is the discontinuation of configurations that lack concrete counterparts. Another such technique is to detect deadlocked configurations. Several additional techniques to abort the analysis when some timeout-limit has been reached have also been shown to be very useful.

When used, the discussed techniques should basically guarantee termination of the analysis even if the analyzed program might not terminate itself, provided that suitable timeout-limits etc. are chosen.

### 10.5 Other Applications of the Analysis

Given that the analysis terminates, some interesting results follow. The analysis could be used as a precise deadlock analysis including the timing behavior of
Chapter 10. Conclusions

If a shared memory position is read by some thread, the analysis first derives all possible values for that memory position before the thread finally reads it. This ensures that the timing behavior of communicating threads is safely and tightly estimated.

Question 3: "How can the timing of synchronizing tasks be safely and tightly estimated?"

To safely estimate the timing behavior of synchronizing tasks, all possible execution interleavings (i.e., orders in which threads acquire locks) must inevitably be considered. Currently, this is achieved by spawning a new configuration for each possible interleaving. This strategy increases the risk of path explosion (the risk is exponentially proportional to the number of synchronizing threads). Further abstraction needs to be incorporated to decrease this risk. However, to achieve a reasonably tight estimate on the timing behavior of synchronizing tasks, the level of abstraction on lock states must be kept fairly low. There is a risk that the analysis result becomes more or less useless when the level of abstraction becomes too high. This is a severe trade-off that must be carefully balanced.

Question 4: "How can programs suffering from deadlocks and other types of nonterminating programs be handled?"

Since abstract execution suffers the risk of not terminating if a nonterminating situation in the analyzed program is encountered, several techniques to increase the probability of termination have been incorporated into the analysis (as discussed in the previous section). One such technique is the discontinuation of configurations that lack concrete counterparts. Another such technique is to detect deadlocked configurations. Several additional techniques to abort the analysis when some timeout-limit has also been shown to be very useful.

When used, the discussed techniques should basically guarantee termination of the analysis even if the analyzed program might not terminate itself, provided that suitable timeout-limits etc. are chosen.

10.5 Other Applications of the Analysis

Given that the analysis terminates, some interesting results follow. The analysis could be used as a precise deadlock analysis including the timing behavior of the program. If the set of deadlocked configurations (cf. $\tilde{C}^d$ in Algorithm 6.1) is empty, the program is deadlock free up until (and including) the point in time described by the abstract timeout input argument (i.e., $\tilde{t}_{to}$ in Algorithm 6.1).

There are many methods and tools capable of finding deadlocks in concurrent programs available [12, 23, 26, 38, 58, 59, 90, 114, 127], but none of them includes the timing properties of the analyzed program as far as the author understands. Including timing properties can greatly improve the precision of the deadlock analysis since possible deadlock situations might logically exist in the program, but they might actually never occur due to the timing behavior of the program threads.

Furthermore, the analysis could also be used to determine whether a program is guaranteed to terminate. If the sets containing deadlocked and timed-out configurations (i.e., $\tilde{C}^d$ and $\tilde{C}^t$ in Algorithm 6.1, respectively) are empty, the program is guaranteed to terminate within the returned timing bounds.

10.6 Future Work

The most important issues to address are the different encountered complexity problems. These problems include both the traditional path explosion problem and the discovered communication- and synchronization-related problems. The path explosion problem could be addressed by introducing merging techniques. This would probably also mean that the level of abstraction would have to be increased (thus most probably lowering the precision) to increase the possibility of encountering mergeable configurations. This is since all the concrete parts of the configurations (program counters etc.) to merge must be equal in the configurations.

The communication- and synchronization-related problems are probably best handled by specializing the analysis for some specific, restricted programming model and perhaps also a more restricted (i.e., less general) instruction set. One example is to only allow synchronization using barriers instead of mutex-like primitives. Another example is to only focus on data parallel programming where threads, for example, cooperate on executing loop iterations and are basically identical. Yet another example is to assume that all communication is done within critical sections; i.e., that all writing and reading of shared memory is protected by first acquiring a lock (which means that all such operations are sequentialized). These restrictions could allow for further abstractions to be made and thus also a decrease in the complexity of the analysis. The increased abstractions should then mainly address the domains for
variable and lock states and the syntax and semantics of PPL.

Note that assuming that all communication only occurs within critical sections could greatly reduce the spatial complexity, and also increase the precision, of the analysis. This is because no write history needs to be kept track of since only one single abstract value would be possible for each shared memory location at each abstract point in time due to the sequence information given by the execution pattern of the threads in the analyzed program. However, the problem of determining all possible values that the reading thread could see might still persist if the current scheme for handling synchronization would change. If the current scheme was kept, then this problem is solved in the inherent spawning of a unique transition for each possible new owner of the given lock.

Since PPL is rudimentary and designed to put focus on global data flow and thread synchronization, the instruction set could be extended to include more functionality and thus imitate a more realistic instruction set. Some candidate instruction sets are XCore [85], LLVM [122], ALF [44, 45], ARM [7] and PowerPC [64]. It could also be possible to make the implementation a flexible framework which could allow for the analyzed instruction set to be switched. This could be done by dividing instructions into special classes. Note, however, that implementing a larger instruction set most probably would increase the complexity of the analysis.

Some model of the underlying architecture (i.e., the function ABSTIME) should be derived. Since the focus of this thesis has excluded a formal investigation and definition of ABSTIME, a more realistic model for some target architecture should be derived.

Besides using the benchmarking setups presented in this thesis, further evaluation could be performed using some extended suitable benchmark suite of concurrent programs. Such a suite is currently being established within the TACLe EU COST Action [120] (a European network of leading researchers within the field of WCET analysis). The suite will include both the sequential versions, and possibly some concurrent versions of some, of the programs in the Mälardalen WCET Benchmark suite [42]. The benchmark suite should include different types of concurrent programs, each of them stressing the analysis in a different way.
Bibliography


316 Bibliography


Appendix A

Notation & Nomenclature

\[ \text{exp}_1 \oplus \text{exp}_2 \] \text{exp}_1 \text{ and } \text{exp}_2 \text{ denote the same thing, often a short and a long notation for a configuration.}

\[ b \ ? \text{exp}_1 : \text{exp}_2 \] \text{If } b, \text{ then } \text{exp}_1, \text{ otherwise } \text{exp}_2.

\[ (o_1, \ldots, o_n) \] \text{Ordinary tuple containing } n \text{ elements.}

\[ \langle o_1, \ldots, o_n \rangle \] \text{Special tuple containing } n \text{ elements. Used to denote complete lattices, Galois connections, configurations, etc.}

\[ [o_1, \ldots, o_n]_{e \in \{e_1, \ldots, e_m\}} \] \text{Expands to } o_1^1, \ldots, o_n^1, \ldots, o_1^m, \ldots, o_n^m; \text{ i.e., one instance of } o_1, \ldots, o_n \text{ for each } e \in \{e_1, \ldots, e_m\}. \text{ Used inside special tuples.}

\[ S \] \text{An arbitrary set (capitalized, italic notation).}

\[ S \] \text{A standard set (capitalized, blackboard bold notation); e.g., } \mathbb{Z}.

\textbf{Set} \text{ A set of analysis-specific elements (first letter capitalized, bold notation); e.g., } \textbf{Thrd}.

\[ \mathcal{P}(S) \] \text{The powerset of } S; \text{ i.e., } \{S' \mid S' \subseteq S\}.

\[ S \times S' \] \text{The Cartesian product; i.e., } \{(e, e') \mid e \in S \land e' \in S'\}.

\[ \prod_{e \in \{e_1, \ldots, e_m\}}(\text{exp}(e)) \] \text{Expands to } \text{exp}(e_1) \times \ldots \times \text{exp}(e_m).
Appendix A. Notation & Nomenclature

\( e, e' \in S \) Short for \( e \in S \land e' \in S \).

\( \lambda e \in S. \exp \) Lambda notation: a function from \( e \), which is an element of \( S \), to \( \exp \), which is often dependent on the specific \( e \).

\( f(s) \) The function \( f \) applied on \( s \).

\( f \circ g(o) \) Equivalent to \( f(g(o)) \).

\( f[o_1][o_2] \) Equivalent to \( (f(o_1))(o_2) \).

\( f s \) The function \( f \) applied on \( s \). This notation is used when dereferencing mappings.

\( \mathcal{I} \) Denotes a state (i.e., a function/mapping from elements to values); e.g., \( r \).

\( \mathcal{I}[s' \mapsto \exp] \) Remapping. Defined as: \( \mathcal{I}[s' \mapsto \exp] s = \begin{cases} \exp & \text{if } s = s' \\ \mathcal{I} s & \text{otrwh} \end{cases} \)

\( ((\mathcal{I} s_1) s_2) \leftarrow \ldots \) A shorthand for \( f \leftarrow \mathcal{I}[s_1 \mapsto (\mathcal{I} s_1)[s_2 \mapsto \ldots]] \).

\text{ALGFUNC} A function defined in a table or algorithm.

\( T \) One of the threads defined in the analyzed program.

\( P, \text{Thrd} \) The analyzed program; i.e., a set of threads.

\( r \) Register (thread-local memory).

\( \text{Reg}_T \) The set of registers used by thread \( T \).

\( x \) Variable (global memory).

\( \text{Var} \) The set of variables defined in the program.

\( lck \) Lock (shared resource).

\( \text{Lck} \) The set of locks defined in the program.

\( pc \) Program counter (unique for each thread).

\( \tilde{f} \) \( f \) defined in some abstract domain; i.e., an abstraction of \( f \).

\( \tau, \tilde{\tau} \) Mapping from registers to their values (unique for each thread).

\( t^a, \tilde{t}^a \) Accumulated execution time (unique for each thread).

\( \aleph, \tilde{\aleph} \) Mapping from variables to mappings from threads to their write history for the given variable.
Mapping from locks to their values.

Configuration (system state).

Partial order relation.

The bottom element in a complete lattice.

The top element in a complete lattice.

The least upper bound operator.

The greatest lower bound operator.

Abstraction function.

Concretization function.

Transition relation for statements (i.e., axioms).

Transition relation for threads (i.e., the program).

The timeout variable used by the analysis.

Begins a comment within algorithms.

Short for otherwise.

Final configurations are configurations in which all the threads issue the halt-statement.

Final states is an alternative notation for final configurations.

Deadlocked configurations are configurations that can never reach the final state.

Timed-out configurations are configurations that cannot reach the final state before a given point in time, the timeout.

Truly deadlocked configurations are abstract configurations that are deadlocked and have valid concrete counterparts; i.e., there is at least one semantically valid concrete configuration that can be abstracted by the given configuration. It must thus be that all threads included in the deadlock are owners of some lock, which has the state locked, and are waiting to acquire some other lock, which also has the state locked.

Falsely deadlocked configurations are abstract configurations that are deadlocked and do not have any valid concrete counterpart; i.e., there is no
semantically valid concrete configuration that can be abstracted by the
given configuration. It could thus be that some thread included in the
deadlock is the owner of some lock, which has the state *unlocked*, and
that some other thread included in the deadlock is waiting to acquire that
lock.

**Axiom statements** are labeled statements; i.e., statements that are not com-
posed of several statements.

**Composed statements** are statements that are composed by two or more ax-
iom (i.e., labeled) statements.

**Active statements** are the axiom statements pointed to by the threads’ pro-
gram counters. The active statement of a thread is the statement that is
executed when the thread is executed. Only one statement in each thread
can be active at any given point in time since all the axiom statements
within a thread are uniquely labeled.

**Frozen threads** are threads in an abstract configuration whose active state-
ments are lock-statements and the locks they are trying to acquire are
currently owned by some other thread.

**Active threads** are not frozen and their active statements are not halt. Note
that this applies to all threads in any concrete configuration, given that
they are not issuing the halt-statement, since only threads in an abstract
configuration can be frozen.

**Executing threads** are the active threads that will execute their active state-
ment at the nearest point in time.

**BCET** (Best-Case Execution Time) is the shortest possible execution time of
the program, given a certain set of initial states.

**WCET** (Worst-Case Execution Time) is the longest possible execution time of
the program, given a certain set of initial states.
Appendix B

List of Assumptions

4.1 TIME is non-negative ........................................ 57
4.3 TIME is non-zero when spin-locking ...................... 57

5.51 ABSTIME is safe and non-negative ...................... 114
Appendix C
List of Definitions

3.1 Monotone function ....................... 27
3.2 Completely additive function .................. 27
3.3 Completely multiplicative function ............... 27
3.9 Galois connection ........................ 31
3.10 Galois insertion ......................... 31
3.11 Induced function ........................ 31
3.12 Adjunction ........................... 32
3.26 Partial order ........................... 42
3.27 Greatest lower bound ...................... 42
3.28 Least upper bound ........................ 42
3.29 Abstraction function,
\( \alpha \) ..................... 42
3.30 Alternative definition – Concretization function,
\( \gamma \) ............... 43
3.31 Interval ............................. 43
3.32 Concretization of interval .................... 43
3.33 Partial order for intervals .................... 44
3.34 Greatest lower bound for intervals ............... 44
3.35 Least upper bound for intervals ................. 44
3.36 Abstraction to interval ..................... 44
4.4 Valid concrete configuration .................. 61
4.7 Collecting semantics ...................... 63
5.1 Concretization of an abstract register state ........... 66
5.2 Partial order for abstract register states .......... 68
Appendix C

List of Definitions

3.1 Monotone function ........................................ 27
3.2 Completely additive function .............................. 27
3.3 Completely multiplicative function ........................ 27
3.9 Galois connection ............................................. 31
3.10 Galois insertion .............................................. 31
3.11 Induced function ............................................ 31
3.12 Adjunction .................................................... 32
3.26 Partial order .................................................. 42
3.27 Greatest lower bound ........................................ 42
3.28 Least upper bound ........................................... 42
3.29 Abstraction function, $\alpha$ ................................ 42
3.30 Alternative definition – Concretization function, $\gamma$ 43
3.31 Interval ......................................................... 43
3.32 Concretization of interval .................................... 43
3.33 Partial order for intervals ................................... 44
3.34 Greatest lower bound for intervals ........................ 44
3.35 Least upper bound for intervals ............................ 44
3.36 Abstraction to interval ........................................ 44

4.4 Valid concrete configuration .................................. 61
4.7 Collecting semantics .......................................... 63

5.1 Concretization of an abstract register state .............. 66
5.2 Partial order for abstract register states ................... 68
Appendix C. List of Definitions

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.3</td>
<td>Greatest lower bound of abstract register states</td>
<td>68</td>
</tr>
<tr>
<td>5.4</td>
<td>Least upper bound of abstract register states</td>
<td>68</td>
</tr>
<tr>
<td>5.5</td>
<td>Abstraction of a set of register states</td>
<td>68</td>
</tr>
<tr>
<td>5.7</td>
<td>Abstract evaluation of arithmetic expressions</td>
<td>69</td>
</tr>
<tr>
<td>5.8</td>
<td>Boolean restriction</td>
<td>69</td>
</tr>
<tr>
<td>5.9</td>
<td>Concretization of an abstract variable state</td>
<td>80</td>
</tr>
<tr>
<td>5.10</td>
<td>Abstraction of a set of variable states</td>
<td>80</td>
</tr>
<tr>
<td>5.12</td>
<td>Partial order of writes</td>
<td>83</td>
</tr>
<tr>
<td>5.13</td>
<td>Least upper bound of writes</td>
<td>83</td>
</tr>
<tr>
<td>5.14</td>
<td>Abstract time precedence</td>
<td>83</td>
</tr>
<tr>
<td>5.15</td>
<td>Partial order for abstract variable states</td>
<td>84</td>
</tr>
<tr>
<td>5.16</td>
<td>Greatest lower bound of abstract variable states</td>
<td>84</td>
</tr>
<tr>
<td>5.17</td>
<td>Least upper bound of abstract variable states</td>
<td>84</td>
</tr>
<tr>
<td>5.18</td>
<td>Time of most recent abstract write</td>
<td>84</td>
</tr>
<tr>
<td>5.19</td>
<td>Safe write history</td>
<td>86</td>
</tr>
<tr>
<td>5.20</td>
<td>Safe value of ( x ) as seen by thread T</td>
<td>88</td>
</tr>
<tr>
<td>5.21</td>
<td>Safe partial order of abstract variable states</td>
<td>88</td>
</tr>
<tr>
<td>5.22</td>
<td>Safe lower bound of abstract variable states</td>
<td>93</td>
</tr>
<tr>
<td>5.23</td>
<td>Safe upper bound of abstract variable states</td>
<td>93</td>
</tr>
<tr>
<td>5.29</td>
<td>Concretization of an abstract lock state</td>
<td>100</td>
</tr>
<tr>
<td>5.30</td>
<td>Abstraction of a set of lock states</td>
<td>101</td>
</tr>
<tr>
<td>5.31</td>
<td>Partial order of abstract lock states</td>
<td>101</td>
</tr>
<tr>
<td>5.32</td>
<td>Greatest lower bound of abstract lock states</td>
<td>102</td>
</tr>
<tr>
<td>5.33</td>
<td>Least upper bound of abstract lock states</td>
<td>102</td>
</tr>
<tr>
<td>5.36</td>
<td>Concretization of an abstract configuration</td>
<td>105</td>
</tr>
<tr>
<td>5.37</td>
<td>Partial ordering of two abstract configurations</td>
<td>105</td>
</tr>
<tr>
<td>5.39</td>
<td>Greatest lower bound for two abstract configurations</td>
<td>106</td>
</tr>
<tr>
<td>5.40</td>
<td>Least upper bound for two abstract configurations</td>
<td>106</td>
</tr>
<tr>
<td>5.41</td>
<td>Abstraction of a set of configurations</td>
<td>107</td>
</tr>
<tr>
<td>5.43</td>
<td>Abstraction of a set of axiom input configurations</td>
<td>109</td>
</tr>
<tr>
<td>5.44</td>
<td>Concretization of an abstract axiom input configuration</td>
<td>109</td>
</tr>
<tr>
<td>5.45</td>
<td>Abstraction of a set of axiom output configurations</td>
<td>109</td>
</tr>
<tr>
<td>5.46</td>
<td>Concretization of an abstract axiom output configuration</td>
<td>109</td>
</tr>
<tr>
<td>5.49</td>
<td>Soundness of the abstract axiom transition relation</td>
<td>110</td>
</tr>
<tr>
<td>6.9</td>
<td>BCET and WCET</td>
<td>187</td>
</tr>
<tr>
<td>6.10</td>
<td>BCET and WCET approximations</td>
<td>188</td>
</tr>
<tr>
<td>8.1</td>
<td>Default initial configuration</td>
<td>212</td>
</tr>
</tbody>
</table>
9.1 Number of operations in an arithmetical expression . . . . . . 231
9.2 Number of operations in a boolean expression . . . . . . . . . . 231
Appendix D

List of Figures

1.1 Execution time distribution of some program. .......... 4
1.2 The three phases in traditional WCET analysis. .......... 6
1.3 Research method overview. .......................... 9
1.4 Galois connection between concrete and abstract domains. .. 12
4.8 Illustration of how Thrdexe is determined. ............. 58
5.8 Illustration of most recent write time. ................. 85
5.9 Illustration of safe variable values. ................... 87
5.10 The timestamps of the writes in ˜ considered by READ . .... 97
5.14 Abstract lock state transitions. ..................... 133
6.1 Relation between concrete and abstract transitions. ...... 164
6.2 Timeout for recursion in ABS EXE. ................... 181
7.5 Communication – Configuration relations. ............... 197
7.9 Synchronization (Deadlock) – Configuration relations. .... 200
7.13 Synchronization (Deadline miss) – Configuration relations. .. 202
7.19 Data parallel loop – Configuration relations. .......... 208
9.13 Analysis running time in milliseconds for the short indepen-
dent threads benchmark. ....................... 237
9.15 Analysis running time in milliseconds for the long independent threads benchmark. ....................... 239
Appendix D

List of Figures

1.1 Execution time distribution of some program. .............. 4
1.2 The three phases in traditional WCET analysis. ............. 6
1.3 Research method overview. .................................. 9
1.4 Galois connection between concrete and abstract domains. . 12

4.8 Illustration of how Thrd_{exe} is determined. ............... 58

5.8 Illustration of most recent write time. ........................ 85
5.9 Illustration of safe variable values. ........................... 87
5.10 The timestamps of the writes in $\mathcal{X}$ considered by READ. . 97
5.14 Abstract lock state transitions. ............................... 133

6.1 Relation between concrete and abstract transitions. .......... 164
6.2 Timeout for recursion in ABSEXE. ............................. 181

7.5 Communication – Configuration relations. ..................... 197
7.9 Synchronization (Deadline miss) – Configuration relations. . 200
7.13 Synchronization (Deadline miss) – Configuration relations. . 202
7.19 Data parallel loop – Configuration relations. .................. 208

9.13 Analysis running time in milliseconds for the short independent threads benchmark. .......... 237
9.15 Analysis running time in milliseconds for the long independent threads benchmark. ............. 239
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>9.17</td>
<td>Analysis running time in milliseconds for the branching heavy benchmark.</td>
<td>241</td>
</tr>
<tr>
<td>9.21</td>
<td>Analysis running time in milliseconds for the communication benchmarks.</td>
<td>244</td>
</tr>
<tr>
<td>9.25</td>
<td>Analysis running time in milliseconds for the synchronization benchmarks.</td>
<td>247</td>
</tr>
<tr>
<td>9.29</td>
<td>Analysis running time in milliseconds for the communication and synchronization benchmarks.</td>
<td>250</td>
</tr>
<tr>
<td>9.31</td>
<td>Analysis running time in milliseconds for the well-structured data parallel benchmark.</td>
<td>252</td>
</tr>
<tr>
<td>9.33</td>
<td>Transitions on the main recursion level for the short independent threads benchmark.</td>
<td>255</td>
</tr>
<tr>
<td>9.35</td>
<td>Transitions on the main recursion level for the long independent threads benchmark.</td>
<td>257</td>
</tr>
<tr>
<td>9.38</td>
<td>Transitions on the main recursion level for the branching heavy benchmark.</td>
<td>259</td>
</tr>
<tr>
<td>9.39</td>
<td>Transitions on recursion levels other than the main recursion level for the branching heavy benchmark.</td>
<td>260</td>
</tr>
<tr>
<td>9.46</td>
<td>Transitions on the main recursion level for the communication benchmarks.</td>
<td>265</td>
</tr>
<tr>
<td>9.47</td>
<td>Transitions on recursion levels other than the main recursion level for the communication benchmarks.</td>
<td>266</td>
</tr>
<tr>
<td>9.51</td>
<td>Transitions on the main recursion level for the synchronization benchmarks.</td>
<td>268</td>
</tr>
<tr>
<td>9.58</td>
<td>Transitions on the main recursion level for the communication and synchronization benchmarks.</td>
<td>272</td>
</tr>
<tr>
<td>9.59</td>
<td>Transitions on recursion levels other than the main recursion level for the communication and synchronization benchmarks.</td>
<td>273</td>
</tr>
<tr>
<td>9.62</td>
<td>Transitions on the main recursion level for the well-structured data parallel benchmark.</td>
<td>275</td>
</tr>
<tr>
<td>9.63</td>
<td>Transitions on recursion levels other than the main recursion level for the well-structured data parallel benchmark.</td>
<td>276</td>
</tr>
<tr>
<td>9.66</td>
<td>Derived execution time bounds for the short independent threads benchmark.</td>
<td>279</td>
</tr>
<tr>
<td>9.69</td>
<td>Derived execution time bounds for the long independent threads benchmark.</td>
<td>281</td>
</tr>
<tr>
<td>9.72</td>
<td>Derived execution time bounds for the branching heavy benchmark.</td>
<td>283</td>
</tr>
</tbody>
</table>
9.79 Derived execution time bounds for the communication benchmarks. .......................................................... 288
9.86 Derived execution time bounds for the synchronization benchmarks. ......................................................... 291
9.93 Derived execution time bounds for the communication and synchronization benchmarks. ........................ 295
9.96 Derived execution time bounds for the well-structured data parallel benchmark. ....................................... 297

10.1 Lock owner assignments based on \( \bar{c} \in \text{Conf} \) resulting in one valid and one invalid configuration. .......... 302
10.2 Lock owner assignments based on \( \bar{c} \in \text{Conf} \) resulting in two valid and two invalid (i.e., falsely deadlocked) configurations. .................................................. 303
Appendix E

List of Tables

4.1 The syntax of PPL. ............................................. 49
4.2 Semantics of concrete axiom transitions. .................... 53
4.3 Semantics of concrete evaluation of arithmetic expressions. . 54
4.4 Semantics of concrete evaluation of boolean expressions. . . 54
4.5 Semantics of concrete program transitions. .................... 55
4.6 Definition of STM and LABELS. ............................ 56
4.7 Definition of STT, OWN, DL, POWN and REL. ............. 56

5.1 PPL operators defined for interval arguments. ............... 67
5.1 Cont. PPL operators defined for interval arguments. ......... 68
5.2 The abstract function evaluating arithmetic expressions. .... 70
5.3 Boolean restriction for intervals. .............................. 71
5.4 Arithmetic restriction for intervals. ............................ 73
5.5 Multiplication operator for inverting interval division expres-
sions. .......................................................... 75
5.6 Division operator for inverting interval multiplication expres-
sions. .......................................................... 77
5.7 Division operator for inverting interval division expressions. . 79
5.11 Definition of STT, OWN, DL, POWN and REL. .......... 101
5.12 Semantics of abstract axiom transitions. ...................... 111
5.13 Semantics of abstract program transitions. .................... 115

7.1 Communication – Program. ................................. 193
7.2 Communication – Timing model. ................................ 194
<table>
<thead>
<tr>
<th>Table Number</th>
<th>Table Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.3</td>
<td>Communication – Configurations (first half)</td>
<td>195</td>
</tr>
<tr>
<td>7.4</td>
<td>Communication – Configurations (second half)</td>
<td>196</td>
</tr>
<tr>
<td>7.6</td>
<td>Synchronization (Deadlock) – Program</td>
<td>198</td>
</tr>
<tr>
<td>7.7</td>
<td>Synchronization (Deadlock) – Timing model</td>
<td>198</td>
</tr>
<tr>
<td>7.8</td>
<td>Synchronization (Deadlock) – Configurations</td>
<td>198</td>
</tr>
<tr>
<td>7.10</td>
<td>Synchronization (Deadline miss) – Program</td>
<td>201</td>
</tr>
<tr>
<td>7.11</td>
<td>Synchronization (Deadline miss) – Timing model</td>
<td>201</td>
</tr>
<tr>
<td>7.12</td>
<td>Synchronization (Deadline miss) – Configurations</td>
<td>202</td>
</tr>
<tr>
<td>7.14</td>
<td>Data parallel loop – Program</td>
<td>203</td>
</tr>
<tr>
<td>7.15</td>
<td>Data parallel loop – Timing model</td>
<td>203</td>
</tr>
<tr>
<td>7.16</td>
<td>Data parallel loop – Configurations (thread-local states)</td>
<td>205</td>
</tr>
<tr>
<td>7.17</td>
<td>Data parallel loop – Configurations (variable states)</td>
<td>206</td>
</tr>
<tr>
<td>7.18</td>
<td>Data parallel loop – Configurations (lock states)</td>
<td>207</td>
</tr>
<tr>
<td>8.1</td>
<td>UPPL: a user-friendly version of PPL</td>
<td>211</td>
</tr>
<tr>
<td>8.2</td>
<td>Example UPPL program</td>
<td>212</td>
</tr>
<tr>
<td>8.3</td>
<td>Interface for changing initial configurations</td>
<td>214</td>
</tr>
<tr>
<td>8.4</td>
<td>Example specification for changing an initial configuration</td>
<td>215</td>
</tr>
<tr>
<td>8.5</td>
<td>Invocation and list of controlling options</td>
<td>217</td>
</tr>
<tr>
<td>9.1</td>
<td>Short independent thread</td>
<td>226</td>
</tr>
<tr>
<td>9.2</td>
<td>Long independent thread</td>
<td>227</td>
</tr>
<tr>
<td>9.3</td>
<td>Branching heavy thread</td>
<td>227</td>
</tr>
<tr>
<td>9.4</td>
<td>Communication thread</td>
<td>228</td>
</tr>
<tr>
<td>9.5</td>
<td>Synchronization thread</td>
<td>229</td>
</tr>
<tr>
<td>9.6</td>
<td>Communication and synchronization thread</td>
<td>229</td>
</tr>
<tr>
<td>9.7</td>
<td>Well-structured data parallel thread</td>
<td>230</td>
</tr>
<tr>
<td>9.8</td>
<td>Short range and large separation timing model</td>
<td>232</td>
</tr>
<tr>
<td>9.9</td>
<td>Medium range and medium separation timing model</td>
<td>232</td>
</tr>
<tr>
<td>9.10</td>
<td>Long range and low separation timing model</td>
<td>233</td>
</tr>
<tr>
<td>9.11</td>
<td>Heterogeneous timing model</td>
<td>234</td>
</tr>
<tr>
<td>9.12</td>
<td>Analysis running time in milliseconds for the short independent threads benchmark</td>
<td>236</td>
</tr>
<tr>
<td>9.14</td>
<td>Analysis running time in milliseconds for the long independent threads benchmark</td>
<td>238</td>
</tr>
<tr>
<td>9.16</td>
<td>Analysis running time in milliseconds for the branching heavy benchmark</td>
<td>240</td>
</tr>
<tr>
<td>9.18</td>
<td>Analysis running time in milliseconds for the communication light (1 loop iteration) benchmark</td>
<td>242</td>
</tr>
</tbody>
</table>
9.19 Analysis running time in milliseconds for the communication
medium heavy (5 loop iterations) benchmark. 243
9.20 Analysis running time in milliseconds for the communication
heavy (10 loop iterations) benchmark. 243
9.22 Analysis running time in milliseconds for the synchronization
light (1 loop iteration) benchmark. 245
9.23 Analysis running time in milliseconds for the synchronization
medium heavy (5 loop iterations) benchmark. 245
9.24 Analysis running time in milliseconds for the synchronization
heavy (10 loop iterations) benchmark. 246
9.26 Analysis running time in milliseconds for the communication
and synchronization light (1 loop iteration) benchmark. 248
9.27 Analysis running time in milliseconds for the communication
and synchronization medium heavy (5 loop iterations) benchmark. 248
9.28 Analysis running time in milliseconds for the communication
and synchronization heavy (10 loop iterations) benchmark. 249
9.30 Analysis running time in milliseconds for the well-structured
data parallel benchmark. 251
9.32 Transitions on the main recursion level for the short independent
threads benchmark. 254
9.34 Transitions on the main recursion level for the long independent
threads benchmark. 256
9.36 Transitions on the main recursion level for the branching heavy
benchmark. 258
9.37 Transitions on recursion levels other than the main recursion
level for the branching heavy benchmark. 258
9.40 Transitions on the main recursion level for the communication
light (1 loop iteration) benchmark. 262
9.41 Transitions on recursion levels other than the main recursion
level for the communication light (1 loop iteration) benchmark. 262
9.42 Transitions on the main recursion level for the communication
medium heavy (5 loop iterations) benchmark. 263
9.43 Transitions on recursion levels other than the main recursion
level for the communication medium heavy (5 loop iterations)
benchmark. 263
9.44 Transitions on the main recursion level for the communication
heavy (10 loop iterations) benchmark. 264
9.45 Transitions on recursion levels other than the main recursion level for the communication heavy (10 loop iterations) benchmark. ................................................... 264
9.48 Transitions on the main recursion level for the synchronization light (1 loop iteration) benchmark. .......................................................... 267
9.49 Transitions on the main recursion level for the synchronization medium heavy (5 loop iterations) benchmark. .............................. 267
9.50 Transitions on the main recursion level for the synchronization heavy (10 loop iterations) benchmark. ............................................. 267
9.52 Transitions on the main recursion level for the communication and synchronization light (1 loop iteration) benchmark. ....................... 270
9.53 Transitions on recursion levels other than the main recursion level for the communication and synchronization light (1 loop iteration) benchmark. ................................................... 270
9.54 Transitions on the main recursion level for the communication and synchronization medium heavy (5 loop iterations) benchmark. ................................................... 270
9.55 Transitions on recursion levels other than the main recursion level for the communication and synchronization medium heavy (5 loop iterations) benchmark. ................................................... 271
9.56 Transitions on the main recursion level for the communication and synchronization heavy (10 loop iterations) benchmark. .............................. 271
9.57 Transitions on recursion levels other than the main recursion level for the communication and synchronization heavy (10 loop iterations) benchmark. ................................................... 271
9.60 Transitions on the main recursion level for the well-structured data parallel benchmark. .......................................................... 274
9.61 Transitions on recursion levels other than the main recursion level for the well-structured data parallel benchmark. ................................................... 274
9.64 Derived bounds on the BCET for the short independent threads benchmark. ............................................................................. 278
9.65 Derived bounds on the WCET for the short independent threads benchmark. ............................................................................. 278
9.67 Derived bounds on the BCET for the long independent threads benchmark. ............................................................................. 280
9.68 Derived bounds on the WCET for the long independent threads benchmark. ............................................................................. 280
9.70 Derived bounds on the BCET for the branching heavy benchmark. ............................................................................. 282
9.71 Derived bounds on the WCET for the branching heavy benchmark. ................................. 282
9.73 Derived bounds on the BCET for the communication light (1 loop iteration) benchmark. ................. 285
9.74 Derived bounds on the WCET for the communication light (1 loop iteration) benchmark. ................. 285
9.75 Derived bounds on the BCET for the communication medium heavy (5 loop iterations) benchmark. ................. 286
9.76 Derived bounds on the WCET for the communication medium heavy (5 loop iterations) benchmark. ................. 286
9.77 Derived bounds on the BCET for the communication heavy (10 loop iterations) benchmark. .................. 287
9.78 Derived bounds on the WCET for the communication heavy (10 loop iterations) benchmark. .................. 287
9.80 Derived bounds on the BCET for the synchronization light (1 loop iteration) benchmark. .................. 289
9.81 Derived bounds on the WCET for the synchronization light (1 loop iteration) benchmark. .................. 289
9.82 Derived bounds on the BCET for the synchronization medium heavy (5 loop iterations) benchmark. ................. 289
9.83 Derived bounds on the WCET for the synchronization medium heavy (5 loop iterations) benchmark. ................. 290
9.84 Derived bounds on the BCET for the synchronization heavy (10 loop iterations) benchmark. .................. 290
9.85 Derived bounds on the WCET for the synchronization heavy (10 loop iterations) benchmark. .................. 290
9.87 Derived bounds on the BCET for the communication and synchronization light (1 loop iteration) benchmark. ................. 293
9.88 Derived bounds on the WCET for the communication and synchronization light (1 loop iteration) benchmark. ................. 293
9.89 Derived bounds on the BCET for the communication and synchronization medium heavy (5 loop iterations) benchmark. ................. 293
9.90 Derived bounds on the WCET for the communication and synchronization medium heavy (5 loop iterations) benchmark. ................. 294
9.91 Derived bounds on the BCET for the communication and synchronization heavy (10 loop iterations) benchmark. ................. 294
9.92 Derived bounds on the WCET for the communication and synchronization heavy (10 loop iterations) benchmark. ................. 294
9.94 Derived bounds on the BCET for the well-structured data parallel benchmark ........................................ 296
9.95 Derived bounds on the WCET for the well-structured data parallel benchmark ........................................ 296
Appendix F

List of Algorithms

5.1 Partial order of abstract variable states .................. 89
5.2 Earliest write for a thread ............................... 90
5.3 Lower bounding two abstract variable states ............. 91
5.4 Upper bounding two abstract variable states ............. 92
5.5 Write to variable ........................................... 94
5.6 Read from variable .......................................... 95
5.7 Time of most recent write ................................. 95
5.8 Time of most recent write in thread ...................... 95
5.9 Trim variable state ......................................... 98
5.10 Split set of writes ......................................... 99
5.11 Determine deadline for lock owner assignment .......... 116
5.12 Determine accumulated execution time ................... 116
5.12 Cont. Determine accumulated execution time .......... 117

6.1 Abstract execution .......................................... 165
6.2 Choose an element .......................................... 167
6.3 Determine if graph has cycles ............................. 168
6.4 Threads to execute in an abstract configuration .......... 169
6.5 Global variables in an abstract configuration .......... 169
6.6 Threads executing a possibly unsafe load-statement .... 171
6.7 Final abstract configuration ............................... 171
6.8 Deadlocked abstract configuration ........................ 171
6.9 Timed-out abstract configuration .......................... 173
### Appendix F. List of Algorithms

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.10</td>
<td>Valid abstract configuration</td>
<td>174</td>
</tr>
<tr>
<td>6.11</td>
<td>Get variable in <code>load</code>-statement</td>
<td>177</td>
</tr>
<tr>
<td>6.12</td>
<td>Get register in <code>load</code>-statement</td>
<td>177</td>
</tr>
<tr>
<td>6.13</td>
<td>BCET/WCET analysis</td>
<td>188</td>
</tr>
</tbody>
</table>
### Appendix G

#### List of Lemmas

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.4</td>
<td>Completely multiplicative functions</td>
<td>27</td>
</tr>
<tr>
<td>3.14</td>
<td>Relation between $\alpha$ and $\gamma$</td>
<td>32</td>
</tr>
<tr>
<td>3.15</td>
<td>Galois connection – Existence</td>
<td>33</td>
</tr>
<tr>
<td>3.18</td>
<td>Monotonicity of $\alpha_p$</td>
<td>35</td>
</tr>
<tr>
<td>3.19</td>
<td>Monotonicity of $\gamma_p$</td>
<td>36</td>
</tr>
<tr>
<td>3.23</td>
<td>Monotonicity of $\gamma_s$</td>
<td>39</td>
</tr>
<tr>
<td>3.37</td>
<td>Monotonicity of $\gamma_{int}$</td>
<td>45</td>
</tr>
<tr>
<td>3.38</td>
<td>Monotonicity of $\alpha_{int}$</td>
<td>45</td>
</tr>
<tr>
<td>4.2</td>
<td>Time only moves forward</td>
<td>57</td>
</tr>
<tr>
<td>4.5</td>
<td>$\rightarrow_{pg}$ preserves lock state validity</td>
<td>61</td>
</tr>
<tr>
<td>4.6</td>
<td>Properties of $\llbracket \cdot \rrbracket'$</td>
<td>62</td>
</tr>
<tr>
<td>5.24</td>
<td>Soundness of WRITE</td>
<td>93</td>
</tr>
<tr>
<td>5.25</td>
<td>Soundness of MOSTRECENTWRITETIMETHREAD</td>
<td>94</td>
</tr>
<tr>
<td>5.26</td>
<td>Soundness of MOSTRECENTWRITETIME</td>
<td>96</td>
</tr>
<tr>
<td>5.27</td>
<td>Soundness of READ</td>
<td>96</td>
</tr>
<tr>
<td>5.28</td>
<td>Soundness of TRIM</td>
<td>98</td>
</tr>
<tr>
<td>5.34</td>
<td>Monotonicity of $\gamma_{lock}$</td>
<td>103</td>
</tr>
<tr>
<td>5.38</td>
<td>Monotonicity of $\gamma_{conf}$</td>
<td>105</td>
</tr>
<tr>
<td>5.50</td>
<td>Soundness of $\rightarrow_{ax}$</td>
<td>110</td>
</tr>
<tr>
<td>5.52</td>
<td>Time accumulation</td>
<td>120</td>
</tr>
<tr>
<td>5.53</td>
<td>Thread isolation</td>
<td>121</td>
</tr>
</tbody>
</table>
Appendix G. List of Lemmas

5.54 Soundness of DLLOCK ........................................ 122
5.55 Partial soundness of ACCTIME ........................... 128
5.56 Properties of owner assignment for lock-transitions ...... 132
5.57 Soundness of $\xrightarrow{prg}$, no frozen thread .......... 136
5.58 Soundness of $\xrightarrow{prg}$, possibly frozen thread ......... 146
5.59 Soundness of $\xrightarrow{prg}$, final state .................... 157

6.1 Soundness of CYCLE ........................................... 167
6.2 Soundness of EXETHRD ........................................ 168
6.3 Soundness of GLOBALVAR ................................. 170
6.4 Soundness of EXELOADTHRD ............................... 170
6.5 Soundness of ISDEADLOCK ................................. 172
6.6 Soundness of ISTIMEOUT ..................................... 173
6.7 Soundness of ISVALID .......................................... 174
### Appendix H

**List of Theorems**

<table>
<thead>
<tr>
<th>Section</th>
<th>Theorem Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.5</td>
<td>Complete lattice – Lifting</td>
<td>28</td>
</tr>
<tr>
<td>3.6</td>
<td>Complete lattice – Cartesian product</td>
<td>28</td>
</tr>
<tr>
<td>3.7</td>
<td>Complete lattice – Total function space</td>
<td>29</td>
</tr>
<tr>
<td>3.8</td>
<td>Complete lattice – Monotone function space</td>
<td>30</td>
</tr>
<tr>
<td>3.13</td>
<td>Adjunctions and Galois connections</td>
<td>32</td>
</tr>
<tr>
<td>3.16</td>
<td>Galois connection – Independent attribute method</td>
<td>34</td>
</tr>
<tr>
<td>3.17</td>
<td>Galois connection – Lifted independent attribute method</td>
<td>35</td>
</tr>
<tr>
<td>3.20</td>
<td>Galois connection – Double lifting</td>
<td>37</td>
</tr>
<tr>
<td>3.21</td>
<td>Not a Galois connection – Double lifting</td>
<td>37</td>
</tr>
<tr>
<td>3.22</td>
<td>Galois connection – Function space</td>
<td>38</td>
</tr>
<tr>
<td>3.24</td>
<td>Galois connection – Lifted function space</td>
<td>39</td>
</tr>
<tr>
<td>3.25</td>
<td>Galois connection – Indexing</td>
<td>40</td>
</tr>
<tr>
<td>3.39</td>
<td>Galois insertion – Intervals</td>
<td>45</td>
</tr>
<tr>
<td>5.6</td>
<td>Galois connection – Register states</td>
<td>69</td>
</tr>
<tr>
<td>5.11</td>
<td>Galois connection – Variable states</td>
<td>81</td>
</tr>
<tr>
<td>5.35</td>
<td>Galois connection – Lock states</td>
<td>103</td>
</tr>
<tr>
<td>5.42</td>
<td>Galois connection – Configurations</td>
<td>107</td>
</tr>
<tr>
<td>5.47</td>
<td>Galois connection – Axiom input configurations</td>
<td>109</td>
</tr>
<tr>
<td>5.48</td>
<td>Galois connection – Axiom output configurations</td>
<td>110</td>
</tr>
<tr>
<td>6.8</td>
<td>Soundness of <code>ABSEXE</code></td>
<td>177</td>
</tr>
<tr>
<td>6.11</td>
<td>Soundness of <code>ANALYSIS</code></td>
<td>189</td>
</tr>
</tbody>
</table>
Index

ERLANG, 14, 210, 211, 215, 216, 218, 240, 261, 284
TIMES, 10
abstract domain, 11, 30
abstract execution, 16, 17, 20, 305, 306
abstract interpretation, 11, 20, 22, 25, 51, 305
abstraction, 11, 15, 17, 30, 59, 65, 69, 163, 299, 301, 324
anti-symmetric relation, see relation
BCET, BCET, 3–5, 7, 13, 14, 16, 20, 187, 190, 231, 277, 326
Best-Case Execution Time, see BCET, BCET
bottom element, 26
bounds
  lower, 26
  greatest, 26
  upper, 26
  least, 26
calculation, 6
completely additive function, see function
completely multiplicative function, see function
concrete domain, 31
COST Action, 308
COTS, 233
dynamic analysis, 5
eMBEDDED SYSTEM, 1
estimation
  safe, 3, 7
  tight, 4, 7, 13
fixed-point calculation, 118
flow analysis, 5
function
  completely additive, 27
  completely multiplicative, 27
  monotone, 27
  partial, 26
  total, 26
Galois connection, 11, 31
global memory, see variable
greatest lower bound, see bounds
halting-problem, 7
high-level analysis, 5
hybrid analysis, 7
least upper bound, see bounds
local memory, see register
lock, 47
low-level analysis, 5
lower bound, see bounds
Mälardalen WCET Benchmark suite, 308
model checking, 1, 10, 11, 20, 22, 23, 254
   bounded, 23
   symbolic, 22
monotone function, see function
multi-core CPU, 2, 3, 5, 14, 20, 21, 23, 24, 47, 65, 233, 299, 303
Note, 25, 26, 43, 48, 50, 66, 83, 93, 127, 163, 187
partial function, see function
partial ordering, 26
probabilistic analysis, 5
processor-behavior analysis, 5
real-time system, 1–3, 10, 11, 14
   hard, 2, 3
   soft, 2
reflexive relation, see relation
register, 47
relation, 26
   anti-symmetric, 26
   reflexive, 26
   transitive, 26
safe estimation, see estimation
shared memory, 5, 7, 14, 21, 24, 47, 300
single-core CPU, 5, 10, 23, 24
static analysis, 5
symbolic execution, 13, 20, 23

TACLe, see COST Action
tight estimation, see estimation
top element, 26
total function, see function
transitive relation, see relation
UPPAAL, 10, 11, 20
upper bound, see bounds
variable, 47
WCET, WCET, 3–5, 7, 8, 10, 13–16, 19–24, 119, 187, 190, 231, 277, 308, 326
Worst-Case Execution Time, see WCET, WCET
Mälardalen WCET Benchmark suite, 308

model checking, 1, 10, 11, 20, 22, 23, 254

bounded, 23

symbolic, 22

monotone function, see function

multi-core CPU, 2, 3, 5, 14, 20, 21, 23, 24, 47, 65, 233, 299, 303

Note, 25, 26, 43, 48, 50, 66, 83, 93, 127, 163, 187

partial function, see function

partial ordering, 26

probabilistic analysis, 5

processor-behavior analysis, 5

real-time system, 1–3, 10, 11, 14

hard, 2, 3

soft, 2

reflexive relation, see relation

register, 47

relation, 26

anti-symmetric, 26

reflexive, 26

transitive, 26

safe estimation, see estimation

shared memory, 5, 7, 14, 21, 24, 47, 300

single-core CPU, 5, 10, 23, 24

static analysis, 5

symbolic execution, 13, 20, 23

TACLe, see COST Action

tight estimation, see estimation

top element, 26

total function, see function

transitive relation, see relation

UPPAAL, 10, 11, 20

upper bound, see bounds

variable, 47

WCET, Worst-Case Execution Time, 3–5, 7, 8, 10, 13–16, 19–24, 119, 187, 190, 231, 277, 308, 326