STATIC TIMING ANALYSIS OF PARALLEL SYSTEMS USING ABSTRACT EXECUTION

Andreas Gustavsson

2014

School of Innovation, Design and Engineering
Abstract

The Power Wall has stopped the past trend of increasing processor throughput by increasing the clock frequency and the instruction level parallelism. Therefore, the current trend in computer hardware design is to expose explicit parallelism to the software level. This is most often done using multiple 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), and 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 output. For some real-time systems, a failure to meet the temporal requirements can have catastrophic consequences. Therefore, it is of utmost importance that methods to analyze and derive safe estimations on the timing properties of parallel computer systems are developed.

This thesis presents an analysis that derives safe (lower and upper) bounds on the execution time of a given parallel system. The interface to the analysis is a small concurrent programming language, based on communicating and synchronizing threads, that is formally (syntactically and semantically) defined in the thesis. The analysis 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. Basically, abstract execution simulates the execution of several real executions of the analyzed program in one go. The thesis also proves the soundness of the presented analysis (i.e. that the estimated timing bounds are indeed safe) and includes some examples, each showing different features or characteristics of the analysis.
Abstract

The Power Wall has stopped the past trend of increasing processor throughput by increasing the clock frequency and the instruction level parallelism. Therefore, the current trend in computer hardware design is to expose explicit parallelism to the software level. This is most often done using multiple 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), and 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 output. For some real-time systems, a failure to meet the temporal requirements can have catastrophic consequences. Therefore, it is of utmost importance that methods to analyze and derive safe estimates on the timing properties of parallel computer systems are developed.

This thesis presents an analysis that derives safe (lower and upper) bounds on the execution time of a given parallel system. The interface to the analysis is a small concurrent programming language, based on communicating and synchronizing threads, that is formally (syntactically and semantically) defined in the thesis. The analysis 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. Basically, abstract execution simulates the execution of several real executions of the analyzed program in one go. The thesis also proves the soundness of the presented analysis (i.e. that the estimated timing bounds are indeed safe) and includes some examples, each showing different features or characteristics of the analysis.
I would like to express my deepest gratitude to 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 so far. Without you, this thesis would not exist. A special thanks goes to Vesa Hirvisalo for putting a lot of energy and time into getting acquainted with, and suggesting improvements on, my research. Last, but far from least, I would like to thank everybody with whom I have shared many laughs and experiences during coffee breaks, trips, parties, after works and other activities. Thank you all!

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".

Andreas Gustavsson
Västerås, October, 2014
Acknowledgments

I would like to express my deepest gratitude to 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 so far. Without you, this thesis would not exist. A special thanks goes to Vesa Hirvisalo for putting a lot of energy and time into getting acquainted with, and suggesting improvements on, my research. Last, but far from least, I would like to thank everybody with whom I have shared many laughs and experiences during coffee breaks, trips, parties, after works and other activities. Thank you all!

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”.

Andreas Gustavsson
Västerås, October, 2014
# Contents

1 Introduction .............................................. 1
1.1 Real-Time Systems ................................. 1
1.2 Execution Time Analysis ......................... 3
1.3 Research Questions ............................... 7
1.4 Pilot Study ........................................ 8
1.5 Approach ........................................... 9
1.6 Contribution ....................................... 11
1.7 Included Publications .......................... 12
1.8 Thesis Outline ................................... 13

2 Related Work ........................................... 15
2.1 Static WCET Analysis ............................ 15
2.2 Static WCET Analysis for Multi-Processors .......... 16
2.3 WCET Analysis Using Model Checking ............ 18
2.4 Multi-Core Analyzability ........................ 19

3 Preliminaries .......................................... 21
3.1 Partially Ordered Sets & Complete Lattices .......... 22
3.2 Constructing Complete Lattices .................. 24
3.3 Galois Connections & Galois Insertions ........... 26
3.4 Constructing Galois Connections .................. 30
3.5 Constructing Galois Insertions ................... 37
3.6 The Interval Domain ............................... 39

4 PPL: A Concurrent Programming Language ........... 43
4.1 States & Configurations .......................... 45
4.2 Semantics ........................................ 47
4.3 Collecting Semantics ........................................ 57

5 Abstractly Interpreting PPL .................................. 59
  5.1 Arithmetical Operators for Intervals ....................... 60
  5.2 Abstract Register States ..................................... 60
  5.3 Abstract Evaluation of Arithmetic Expressions .............. 63
  5.4 Boolean Restriction for Intervals ............................ 63
  5.5 Abstract Variable States ..................................... 73
  5.6 Abstract Lock States .......................................... 91
  5.7 Abstract Configurations ..................................... 95
  5.8 Abstract Semantics ........................................... 101

6 Safe Execution Time Analysis by Abstract Execution .......... 155
  6.1 Abstract Execution ........................................... 155
  6.2 Execution Time Analysis ..................................... 177

7 Examples ......................................................... 181
  7.1 Communication ............................................... 181
  7.2 Synchronization – Deadlock .................................. 186
  7.3 Synchronization – Deadline Miss ............................. 189
  7.4 Parallel Loop .................................................. 190

8 Conclusions ....................................................... 197
  8.1 The Underlying Architecture ................................. 197
  8.2 Algorithmic Structure & Complexity ........................ 198
  8.3 Nonterminating Transition Sequences ....................... 202
  8.4 The Research Questions ..................................... 203
  8.5 Other Applications of the Analysis ......................... 204
  8.6 Future Work .................................................... 205

Bibliography ....................................................... 207

A Notation & Nomenclature ..................................... 221

B List of Assumptions ............................................. 225

C List of Definitions ............................................. 227

D List of Figures ................................................. 229
Contents

E List of Tables 231
F List of Algorithms 233
G List of Lemmas 235
H List of Theorems 237
Index 239
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 [15].

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 [54].

"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 [15].

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 [54].

“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.”
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 (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 (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 [89].

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 par-
allelized 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 software libraries for explicitly parallelizing sequential code automatically. One example of such a library available for C/C++ and Fortran code running on shared-memory machines is OpenMP [83]. 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 [5, 34, 70]. 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 execution time for the given program and set of initial states. The shortest execution time 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 execution time for the given program and set of initial states. The longest execution time is referred to as the Worst-Case Execution Time (WCET). Note that both the BCET and the WCET could
Chapter 1. Introduction

The actual WCET must be found or upper bounded.

When considering simple-enough (most often sequential) hardware, i.e. hardware that is free from timing anomalies [72], research on \textit{execution time analysis} can result in very efficient methods for \textit{tight} (i.e. not too over-approximate) estimation of the (BCET or) WCET. This is because tight estimations of the best-case and worst-case execution times for each single instruction, or a block of instructions, can be derived in isolation from other statements. However, when introducing multi-core architectures with

\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 deadlocks.}
shared memory, the hardware does most likely suffer from timing anomalies regardless of how simple the processor cores are [2, 72, 97]. Practically, this means that an execution time for a statement that lies in-between the statement’s BCET and WCET could result in the global (BCET and) WCET. The consequence is that the only safe option is to take all the possible execution scenarios into account when estimating the global timing bounds.

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 [30, 113], Bound-T [49, 113], Chronos [65, 113], Heptane [113], OTAWA [8], RapiTime [96, 113], SWEET [27, 113], SymTA/P [113] and TuBound [91, 113]. 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. RapiTime is heavily based on dynamic techniques.

In dynamic WCET analysis, measurements of the actual execution time of the software running on the target hardware are performed. This method is not guaranteed to execute the program’s worst-case path, though, 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 (longer) execution times than the worst execution time detected by the measurements.

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 executions of code sequences based on the considered hardware; and the calculation, where the most time-consuming path is found, using information derived in the first two phases. This is illustrated in Figure 1.2.

The flow analysis phase takes as input some form of representation of the analyzed program’s control flow structure (e.g. a Control Flow Graph, CFG [104]), and possibly additional information such as input data ranges and
bounds on the number of iterations for some loops. The additional information is often either provided manually by annotating the code, or derived using a preceding value analysis which statically finds information about the values of the processor registers and program variables etc. at every program point. The flow analysis outputs constraints on the dynamic behavior of the analyzed program, such as bounds on the number of loop iterations, (in)feasible paths in the control flow structure, dependencies between conditional statements, and which functions may be called.

The processor-behavior analysis phase takes as input the compiled and linked program binary and uses a model of the processor, memory subsystem, buses and all peripherals etc. to derive safe timing information for the execution of the different instructions found in the program binary. The execution time of a given instruction is most often dependent on the occupancy state of the different hardware components; i.e. on the execution context. To derive tight timing information, it is therefore necessary to derive the possible execution contexts for a given instruction; i.e. the possible hardware states in which the instruction can be executed. The processor-behavior analysis outputs such information.

For the calculation phase, there exist several possible strategies for combining the information retrieved from the flow analysis and processor-behavior analysis to derive a safe estimation of the WCET. These classes 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.

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 (cf. [59]). 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 must be derived. For many of the traditional methods for analyzing sequential programs, there are ways to guarantee termination using widening/narrowing techniques [81]. These techniques are not directly applicable to the method presented in this thesis, though. Therefore, other techniques will be presented.

## 1.3 Research Questions

This thesis mainly tries to answer the following questions. 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: “How can safe and tight bounds on the execution time of a concurrent program consisting of dependent tasks be derived?”

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

Question 3: “How can programs suffering from deadlocks and other types of nonterminating programs be handled?”

Question 4: “How can the timing of communicating tasks be safely and tightly estimated?”

1.4 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 [24, 52, 78].

Timed automata² can be used to model real-time systems [4]. An automaton can be viewed as a state machine with locations and edges [57]. 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 [57]. (Continuous) time is expressed as a set of real-valued variables called clocks. UPPAAL³ [9, 63, 111] is a tool used to model, simulate and verify networks of timed automata [9, 10, 57].

Preceding the work presented in this thesis, an initial study [42] in which UPPAAL was used to model, and derive high precision estimates on the timing bounds of, a small parallel real-time system was performed. The paper shows that timing analysis of parallel real-time systems can be performed using the model checking techniques available in for example UPPAAL. However, the proposed method (i.e. the way the system was modeled and analyzed) did not scale very well, for example with respect to the number of threads in the analyzed program. Therefore, it was decided not to continue on the pure model checking path (although, there might be other ways to model the system that would succeed better).

²The formal syntax and semantics of timed automata can be found in [3] and [57].
³An introduction to UPPAAL and the formal semantics of networks of timed automata are given in [9] and [57], respectively.
1.5 Approach

Abstract interpretation [23, 35, 81] 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 1.3. 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 \textbf{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 \textbf{Intv} is completely defined, and that a Galois connection is established between \textbf{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 domain, the value of \( x \) could safely be represented by \([1, 8]\). This is an overapproximation 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) [35, 40] was originally designed as a method to derive program flow constraints [113] on imperative sequential programs, like bounds on the number of iterations in loops and infeasible program path constraints. This information can be used by a subsequent execution time (WCET) analysis [113] to compute a safe WCET bound. AE is based on abstract interpretation, and is basically a very context-sensitive value analysis [81, 113] which can be seen as a form of symbolic execution [35] (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 [35]. 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 [28].

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. 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 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 [115].

1.6 Contribution

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 synchronization on locks. The semantics of PPL includes timing behavior and is defined based on the familiar notation of operational semantics (cf. [82]).

2. An abstraction of the PPL semantics where values and concrete points in time are abstracted using intervals.

3. A safe timing analysis based on the abstract semantics of PPL. A complete correctness/soundness proof is provided.

1.7 Included Publications

This thesis includes 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 [41].

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 [43].

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.
Chapter 1. Introduction

1. PPL: a formally defined, rudimentary, concurrent programming language for real-time systems, including shared memory and synchronization on locks. The semantics of PPL includes timing behavior and is defined based on the familiar notation of operational semantics (cf. [82]).

2. An abstraction of the PPL semantics where values and concrete points in time are abstracted using intervals.

3. A safe timing analysis based on the abstract semantics of PPL. A complete correctness/soundness proof is provided.

1.7 Included Publications

This thesis includes 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 [41].

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 [43].

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.

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 [45].

This paper addresses contributions 1, 2 and 3 and summarizes the 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 [42].

This paper does not address any of the main contributions of this thesis. However, this paper contains the pilot study discussed in Section 1.4.

1.8 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.

Chapter 5 presents a semi-safe 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 discusses the research questions and the analysis presented in Chapter 6. The chapter also gives pointers to future work.

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

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

Appendices B-H present listings of the assumptions, definitions, figures, tables, algorithms, lemmas and theorems defined in this thesis, respectively.
Chapter 2

Related Work

WCET-related research started with the introduction of timing schemas by Shaw in 1989 [104]. 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 and 2008 can be found in [92] and [113] respectively.

2.1 Static WCET Analysis

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.

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 [26, 47, 68, 105, 110], caches [66, 68, 110, 112], branch predictors [21], and super-scalar CPUs [67, 101].

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 pos-
sible 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 [40, 48, 49, 71, 110].

Three main methods exist for the WCET calculation: The tree-based method [20, 21, 68], originating from Park’s timing schemas [87]; the path-based method [47, 106]; and the Implicit Path Enumeration Technique (IPET) [27, 49, 66, 93, 110], 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 [93]. A comparison of the graph-based and IPET approaches is performed in [52]. The graph-based approach is conducted using model checking in UPPAAL [9, 63, 111]. It is shown that IPET outperforms the model checking-based approach, but that model checking allows for 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. Another motivation to why model checking could be useful in WCET analysis can be found in [78].

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 [28, 40]. This thesis uses basically the same approach, but applies it to explicitly concurrent programs.

2.2 Static WCET Analysis for Multi-Processors

Some research has been conducted within the field of static WCET analysis for multi-core and other types of multi-processor systems. This research is very young and in some cases, the level of evaluation is lacking depth. Note that not all of the material referred to here focus on analysis of concurrent programs. 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.

In this thesis, focus is put on analyzing concurrent programs with synchronizing and communicating threads executing on some arbitrary (i.e. sequen-
tial 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.

A static analysis method for analyzing concurrent programs executing on a multi-core processor with a shared L2 instruction cache has been presented [117]. 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 [118].

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 [18].

Staschulat et al. [107] 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 [79] use a graph based approach and Kronecker algebra to calculate an estimation of the WCET of a concurrent program. The graph is referred to as CPG (Concurrent Program Graph) and plays a role similar to the CFG for sequential programs.

Ozaktas et al. [84] focus on analyzing synchronization delays experienced by POSIX threads in a concurrent program executing on time-predictable shared-memory multi-core architectures.

Potop-Butucaru and Puaut [90] target static timing analysis of a concurrent program 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 of this approach is to enable the use of the traditional three-phase WCET analysis when analyzing parallel systems.

There is also some research on data flow analysis for concurrent programs [25, 33, 60], which is of relevance to WCET analysis. Constant propagation has also been considered [64]. Using its Mthread plugin [1], the Framework for Modular Analysis of C programs, Frama-C, can perform a safe value analysis of concurrent C programs using abstract interpretation. A survey of analyses for concurrent and concurrent programs is found in [98].
2.3 WCET Analysis Using Model Checking

Lv et al. [73] and Wu and Zhang [116] 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. [74] have also combined abstract interpretation with model checking to avoid the scalability problems found in, for example, [42]. 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 [76]. This approach is similar to abstract 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 [12, 13]. This approach reduces model checking to a propositional satisfiability problem and has been primarily used to find hardware bugs but similar approaches have also been used for very accurate WCET analysis [14].

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 [31, 94, 95]. 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.
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 to the method found in [117], memory bits for each instruction are used to determine whether the instruction should be cached or not [46]. 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.

Arbiters (hardware circuits) can be added to a shared memory multi-core processor to synchronize the memory accesses from different cores in order to increase the timing predictability of the system [85]. The result is a multi-core architecture that can be analyzed with existing single-core (and single-task) WCET analysis tools.

GAMC [86] 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 [5, 100]. 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. [58] 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 [77, 99] strives towards providing a timing analyzable multi-core CPU with a system level software (cf. operating system). A case study [99] 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 [88] is a continuation of the MERASA
The PROARTIS project [17] 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 [61].
Chapter 3

Preliminaries

In general, basing a timing analysis on the concrete semantics of a program is infeasible due to the enormous number of states that must be explored. As discussed in Section 1.5, abstract interpretation [23, 35, 81] 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 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 [35, 81]. 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.

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 \((A, \mathcal{R})\) 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, \( V = \langle V, \sqsubseteq, \bigcup, \bigcap, \bot, \top \rangle \), 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 \( \top = \bigcup V = \bigcap \emptyset \).

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:

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

\[\Box\]

---

1 Extensive introductions to complete lattices can be found in many textbooks, e.g. [81].
3.1 Partially Ordered Sets & Complete Lattices

**Definition 3.2 (Completely additive function):**
A function, \( f : V_1 \to V_2 \), between the partially ordered sets \( V_1 = (V_1, \sqsubseteq_1) \) and \( V_2 = (V_2, \sqsubseteq_2) \) is completely additive if for all \( V'_1 \subseteq V_1 \)
\[
f(\bigcup_{1} V'_1) = \bigcup_{2}\{f(v) \mid v \in V'_1\}
\]
whenever \( \bigcup_{1} V'_1 \) and \( \bigcup_{2}\{f(v) \mid v \in V'_1\} \) exist.

**Definition 3.3 (Completely multiplicative function):**
A function, \( f : V_1 \to V_2 \), between the partially ordered sets \( V_1 = (V_1, \sqsubseteq_1) \) and \( V_2 = (V_2, \sqsubseteq_2) \) is completely multiplicative if for all \( V'_1 \subseteq V_1 \)
\[
f(\bigcap_{1} V'_1) = \bigcap_{2}\{f(v) \mid v \in V'_1\}
\]
whenever \( \bigcap_{1} V'_1 \) and \( \bigcap_{2}\{f(v) \mid v \in V'_1\} \) exist.

**Lemma 3.4 (Completely multiplicative functions):**
If \( V = (V, \sqsubseteq, \sqcup, \sqcap, \bot, \top) \) and \( \bar{V} = (\bar{V}, \bar{\sqsubseteq}, \bar{\sqcup}, \bar{\sqcap}, \bar{\bot}, \bar{\top}) \) are complete lattices and \( \bar{V} \) is finite, then the three conditions
1. \( \gamma : \bar{V} \to V \) is monotone,
2. \( \gamma(\bar{\top}) = \top \), and
3. \( \gamma(\bar{v} \sqcap \bar{v}') = \gamma(\bar{v}) \sqcap \gamma(\bar{v}') \) whenever \( \bar{v} \sqsubseteq \bar{v}' \land \bar{v}' \sqsubseteq \bar{v} \), where \( \bar{v}, \bar{v}' \in \bar{V} \)
are jointly equivalent to \( \gamma : \bar{V} \to V \) being completely multiplicative.

**Proof (cf. [81]).** Assume that \( V = (V, \sqsubseteq, \sqcup, \sqcap, \bot, \top) \) and \( \bar{V} = (\bar{V}, \bar{\sqsubseteq}, \bar{\sqcup}, \bar{\sqcap}, \bar{\bot}, \bar{\top}) \) are complete lattices and that \( \bar{V} \) is finite.

First note that if \( \gamma : \bar{V} \to V \) is completely multiplicative, then the three conditions trivially hold. Next, assuming that the three conditions are fulfilled, it will be proven that
\[
\gamma(\bar{v} \cap \bar{v}') = \bigcap\{\gamma(\bar{v}) \mid \bar{v} \in \bar{V}'\}
\]
where \( \bar{V}' \subseteq \bar{V} \), using induction on the finite cardinality of \( \bar{V}' \subseteq \bar{V} \).

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

If the cardinality of \( \bar{V}' \) is larger than 0, then \( \bar{V}' = \bar{V}'' \cup \{\bar{v}''\} \) where \( \bar{v}'' \not\in \bar{V}'' \); which ensures that the cardinality of \( \bar{V}'' \) is strictly less than that of \( \bar{V}' \). Note that
by condition 1, $\gamma(\bar{v} \cap \bar{v'}) = \gamma(\bar{v}) \cap \gamma(\bar{v'})$ also when $\bar{v} \subseteq \bar{v}' \lor \bar{v}' \subseteq \bar{v}$. Hence, by assuming that $\gamma(\bigcap \bar{v}'') = \bigcap\{\gamma(\bar{v}) \mid \bar{v} \in \bar{v}''\}$ (this is the induction assumption),

$$
\gamma(\bigcap \bar{v}') = \gamma((\bigcap \bar{v}'') \cap \gamma(\bar{v}')) \\
\overset{\text{calc.}}{=} \gamma((\bigcap \bar{v}'') \cap \gamma(\bar{v}')) \\
\overset{\text{cond. 1 and 3}}{=} \gamma(\bigcap \bar{v}'') \cap \gamma(\bar{v}') \\
\overset{\text{ind. ass.}}{=} (\bigcap\{\gamma(\bar{v}) \mid \bar{v} \in \bar{v}''\}) \cap \gamma(\bar{v}') \\
\overset{\text{calc.}}{=} \bigcap\{\gamma(\bar{v}) \mid \bar{v} \in \bar{v}'\}
$$

which proves the lemma. 

\[
\]

### 3.2 Constructing Complete Lattices

There are several different ways to construct complete lattices. Any given set can be lifted into a complete lattice (Theorem 3.5).

**Theorem 3.5 (Complete lattice – Lifting):**

If $S$ is a set, then $(\mathcal{P}(S), \subseteq, \cup, \cap, \emptyset, S)$ is a complete lattice.

**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 $\top = S$ if $\subseteq = \subseteq$ (note that $\subseteq$ is reflexive, transitive and anti-symmetric by definition).

The Cartesian product of two complete lattices is a complete lattice (Theorem 3.6).

**Theorem 3.6 (Complete lattice – Cartesian product):**

If $(V_1, \subseteq_1, \cup_1, \cap_1, \bot_1, \top_1)$ and $(V_2, \subseteq_2, \cup_2, \cap_2, \bot_2, \top_2)$ are complete lattices, then so is $(V, \subseteq, \cup, \cap, \bot, \top)$ where (let $V' \subseteq V$):

\[
V = V_1 \times V_2 = \{(v_1, v_2) \mid v_1 \in V_1 \land v_2 \in V_2\} \\
(v_1, v_2) \subseteq (v'_1, v'_2) \iff v_1 \subseteq_1 v'_1 \land v_2 \subseteq_2 v'_2 \text{ where } v_1, v'_1 \in V_1 \text{ and } v_2, v'_2 \in V_2 \\
\bigcup V' = \bigcup_1 \{v_1 \in V_1 \mid \exists v_2 \in V_2 : (v_1, v_2) \in V'\}, \bigcup_2 \{v_2 \in V_2 \mid \exists v_1 \in V_1 : (v_1, v_2) \in V'\} \\
\bigcap V' = (\bigcap_1 \{v_1 \in V_1 \mid \exists v_2 \in V_2 : (v_1, v_2) \in V'\}, \bigcap_2 \{v_2 \in V_2 \mid \exists v_1 \in V_1 : (v_1, v_2) \in V'\}) \\
\bot = (\bot_1, \bot_2) \\
\top = (\top_1, \top_2)
\]
3.2 Constructing Complete Lattices

PROOF. Assume that \( \langle V_1, \sqsubseteq_1, \sqcup_1, \sqcap_1, \bot_1, \top_1 \rangle \) and \( \langle V_2, \sqsubseteq_2, \sqcup_2, \sqcap_2, \bot_2, \top_2 \rangle \) are complete lattices and let \( V = \{(v_1, v_2) \mid v_1 \in V_1 \land v_2 \in V_2\} \) and \( (v_1, v_2) \sqsubseteq (v'_1, v'_2) \iff v_1 \sqsubseteq_1 v'_1 \land v_2 \sqsupseteq_2 v'_2 \) where \( v_1, v'_1 \in V_1 \) and \( v_2, v'_2 \in V_2 \). (Note that it is straightforward to verify that \( (V, \sqsubseteq) \) is a partially ordered set since \( \sqsubseteq_1 \) and \( \sqsubseteq_2 \) are partial orders.) Also assume that \( V' \subseteq V \).

Since \( \bigcup_1 \{v_1 \in V_1 \mid \exists v_2 \in V_2 : (v_1, v_2) \in V'\} \sqsubseteq_1 v''_1 \) 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 \( \bigcup_2 \{v_2 \in V_2 \mid \exists v_1 \in V_1 : (v_1, v_2) \in V'\} \sqsubseteq_2 v''_2 \) 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 \( \bigcup V' = \bigcup_1 \{v_1 \in V_1 \mid \exists v_2 \in V_2 : (v_1, v_2) \in V'\} \sqcup \bigcup_2 \{v_2 \in V_2 \mid \exists v_1 \in V_1 : (v_1, v_2) \in V'\} \subseteq (v''_1, v''_2) \) (cf. the definition of \( \sqsubseteq \) above). \( \sqcap V' \) 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. \( \square \)

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 \( \langle V_1, \sqsubseteq_1, \sqcup_1, \sqcap_1, \bot_1, \top_1 \rangle \) is a complete lattice, then \( \langle V, \sqsubseteq, \sqcup, \sqcap, \bot, \top \rangle \) where \( \text{let } V' \subseteq V \)

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

\( f \sqsubseteq f' \iff \forall s \in S : f(s) \sqsubseteq_1 f'(s) \text{ where } f, f' \in V \)

\[
\bigcup V' = \lambda s \in S. \bigcup_1 \{f(s) \mid f \in V'\},
\]

\[
\sqcap V' = \lambda s \in S. \sqcap_1 \{f(s) \mid f \in V'\},
\]

\( \bot = \lambda s \in S. \bot_1 \)

\( \top = \lambda s \in S. \top_1 \)

is also a complete lattice. \( \square \)

PROOF. Assume that \( S \) is a set and \( \langle V_1, \sqsubseteq_1, \sqcup_1, \sqcap_1, \bot_1, \top_1 \rangle \) is a complete lattice, \( V = S \to V_1 = \{f : S \to V_1 \mid f \text{ is a total function}\} \) and \( f \sqsubseteq f' \iff \forall s \in S : f(s) \sqsubseteq_1 f'(s) \text{ where } f, f' \in V \). (Note that it is straightforward to verify that \( (V, \sqsubseteq) \) 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 \( \forall s \in S : \forall f' \in V' : f'(s) \sqsubseteq_1 \bigcup_1 \{f(s) \mid f \in V'\} \) and that \( \forall s \in S : \bigcup_1 \{f(s) \mid f \in V'\} \sqsubseteq_1 f'(s) \) for any \( f' \in V \) such that \( \forall s \in S : \forall f' \in V' : f'(s) \sqsubseteq_1 f'(s) \) since \( \langle V_1, \sqsubseteq_1, \sqcup_1, \sqcap_1, \bot_1, \top_1 \rangle \) 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 \( \sqsubseteq \) above). \( \sqcap V' \) is shown in a similar manner.
Since $\bot_1 = \bigcup_1 \emptyset$, it is easy to see that $\bot = \lambda s \in S. \bigcup_1 \emptyset = \lambda s \in S. \bot_1$. $\top$ is shown in a similar manner. □

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 $\langle V_1, \subseteq_1, \bigcup_1, \prod_1, \bot_1, \top_1 \rangle$ and $\langle V_2, \subseteq_2, \bigcup_2, \prod_2, \bot_2, \top_2 \rangle$ are complete lattices, then so is $\langle V, \subseteq, \bigcup, \prod, \bot, \top \rangle$ where (let $V' \subseteq V$):

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

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

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

$$\prod V' = \lambda v_1 \in V_1. \prod_2 \{ f(v_1) \mid f \in V' \},$$

$$\bot = \lambda v_1 \in V_1. \bot_2$$

$$\top = \lambda v_1 \in V_1. \top_2$$

**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).

□

### 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 [81]. 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):**
$\langle \alpha : V \rightarrow D, \gamma : D \rightarrow V \rangle$ is a Galois connection iff $\alpha$ and $\gamma$ are monotone functions that fulfill

$$\left\{ \begin{array}{l}
\alpha \circ \gamma \subseteq_D \lambda d. d \\
\gamma \circ \alpha \subseteq_V \lambda v. v
\end{array} \right.$$
3.3 Galois Connections & Galois Insertions

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

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

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 $\forall d \in D : f(\gamma(d)) \sqsubseteq (\tilde{f}(d))$. The best approximation is achieved by inducing $f$ along $\alpha$ [81]; 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 $D \to D$ 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):**

$\langle \alpha : V \to D , \gamma : D \to V \rangle$ is said to be an adjunction between the complete lattices $V = \langle V, \sqsubseteq_V, \sqcup_V, \sqcap_V, V \rangle$ and $D = \langle D, \sqsubseteq_D, \sqcup_D, \sqcap_D, D \rangle$ 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 $d \in D$.

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

**Theorem 3.13 (Adjunctions and Galois connections):**

$\langle \alpha : V \to D , \gamma : D \to V \rangle$ is an adjunction iff it is a Galois connection.
PROOF (cf. [81]). First assume that \( \langle \alpha : V \to D, \gamma : D \to V \rangle \) is an adjunction. It will be proven that it also is a Galois connection by showing that \( \gamma \circ \alpha \cong \lambda v. v \) and \( \alpha \circ \gamma \subseteq \lambda d. d \). For any \( v \in V \), trivially \( \alpha(v) \subseteq D \alpha(v) \). Using that \( \alpha(v) \subseteq D d \Rightarrow v \subseteq V \gamma(d) \), it can be established that \( v \subseteq V \gamma(\alpha(v)) \). Similarly, for any \( d \in D \), trivially \( \gamma(d) \subseteq V \gamma(d) \). Using that \( v \subseteq V \gamma(d) \Rightarrow \alpha(v) \subseteq D d \), it can be established that \( \alpha(\gamma(d)) \subseteq D d \). Thus, \( \langle \alpha : V \to D, \gamma : D \to V \rangle \) is a Galois connection.

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

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

**Lemma 3.14 (Relation between \( \alpha \) and \( \gamma \)):**

If \( V = \langle V, \subseteq, \cup, \cap, \bot, \top \rangle \) and \( \hat{V} = \langle \hat{V}, \subseteq, \cup, \cap, \hat{\bot}, \hat{\top} \rangle \) are complete lattices, and \( \langle \alpha : V \to \hat{V}, \gamma : \hat{V} \to V \rangle \) is a Galois connection between these lattices, then (let \( v \in V \) and \( \hat{v} \in \hat{V} \)):

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

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

In particular, \( \alpha(\bot) = \hat{\bot} \) and \( \gamma(\hat{\top}) = \top \).

**Proof** (cf. [81]). Assume that \( V = \langle V, \subseteq, \cup, \cap, \bot, \top \rangle \) and \( \hat{V} = \langle \hat{V}, \subseteq, \cup, \cap, \hat{\bot}, \hat{\top} \rangle \) are complete lattices, \( \langle \alpha : V \to \hat{V}, \gamma : \hat{V} \to V \rangle \) is a Galois connection between these lattices, \( v \in V \) and \( \hat{v} \in \hat{V} \).

To show 1, it will first be shown that \( \gamma \) is determined by \( \alpha \). Since \( \langle \alpha : V \to \hat{V}, \gamma : \hat{V} \to V \rangle \) is an adjunction (Theorem 3.13), it must be that \( \gamma(\hat{v}) = \bigcup \{ v \mid v \subseteq \gamma(\hat{v}) \} = \bigcup \{ v \mid \alpha(v) \subseteq \hat{v} \} \). Assume that both \( \langle \alpha, \gamma_1 \rangle \) and \( \langle \alpha, \gamma_2 \rangle \) are Galois connections, then \( \gamma_1(\hat{v}) = \bigcup \{ v \mid v \subseteq \gamma_1(\hat{v}) \} = \bigcup \{ v \mid \alpha(v) \subseteq \hat{v} \} = \bigcup \{ v \mid v \subseteq \gamma_2(\hat{v}) \} = \gamma_2(\hat{v}) \), and thus, \( \gamma_1 = \gamma_2 \). This shows that \( \alpha \) uniquely determines \( \gamma \). Similarly, it must be that \( \alpha(v) = \bigcap \{ \hat{v} \mid \alpha(v) \subseteq \hat{v} \} = \bigcap \{ v \mid v \subseteq \gamma(\hat{v}) \} \). This shows that \( \gamma \) uniquely determines \( \alpha \).
To show 2, consider \( V' \subseteq V \), then
\[
\alpha(\bigcup V') \sqsubseteq \tilde{v} \quad \text{Th. 3.13}\quad \bigcup V' \subseteq \gamma(\tilde{v})
\]
\[
\text{calc:} \quad \forall v \in V' : v \sqsubseteq \gamma(\tilde{v})
\]
\[
\text{Th. 3.13} \quad \forall v \in V' : \alpha(v) \sqsubseteq \tilde{v}
\]
\[
\text{calc:} \quad \bigcup \{ \alpha(v) \mid v \in V' \} \subseteq \tilde{v}
\]
and it follows that \( \alpha(\bigcup V') = \bigcup \{ \alpha(v) \mid v \in V' \} \).

The proof that \( \gamma(\bigcap V') = \bigcap \{ \gamma(\tilde{v}) \mid \tilde{v} \in V' \} \) 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 \( V = \langle V, \sqsubseteq, \sqcup, \sqcap, \bot, \top \rangle \) and \( \tilde{V} = \langle \tilde{V}, \sqsubseteq, \sqcup, \sqcap, \tilde{\bot}, \tilde{\top} \rangle \) 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 (cf. [81]).** Assume that \( V = \langle V, \sqsubseteq, \sqcup, \sqcap, \bot, \top \rangle \) and \( \tilde{V} = \langle \tilde{V}, \sqsubseteq, \sqcup, \sqcap, \tilde{\bot}, \tilde{\top} \rangle \) 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:
\[
\gamma(\tilde{v}) = \bigcup \{ v' \mid \alpha(v') \sqsubseteq \tilde{v} \}
\]
Then it must be that \( \alpha(v) \sqsubseteq \tilde{v} \Rightarrow v \in \{ v' \mid \alpha(v') \sqsubseteq \tilde{v} \} \Rightarrow v \subseteq \gamma(\tilde{v}) \), 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,
\[
\alpha(\gamma(\tilde{v})) = \alpha(\bigcup \{ v' \mid \alpha(v') \sqsubseteq \tilde{v} \})
\]
\[
= \bigcup \{ \alpha(v') \mid \alpha(v') \sqsubseteq \tilde{v} \}
\]
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.
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 (cf. [81]). 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.
\[
\alpha((v_1, v_2)) \sqsubseteq_D (d_1, d_2) \overset{\text{Def. } \alpha}{=} (\alpha_1(v_1), \alpha_2(v_2)) \sqsubseteq_D (d_1, d_2)
\]
\[
\overset{\text{calc.}}{=} \alpha_1(v_1) \sqsubseteq_{D_1} d_1 \land \alpha_2(v_2) \sqsubseteq_{D_2} d_2
\]
\[
\overset{\text{Th. 3.13}}{=} \forall v_1 \in V_1 \land \forall v_2 \in V_2 \land \forall d_1 \in D_1 \land \forall d_2 \in D_2 \land v_1 \sqsubseteq_{V_1} \gamma_1(d_1) \land v_2 \sqsubseteq_{V_2} \gamma_2(d_2)
\]
\[
\overset{\text{calc.}}{=} (v_1, v_2) \sqsubseteq_{V_1 \times V_2} (\gamma_1(d_1), \gamma_2(d_2))
\]
\[
\overset{\text{Def. } \gamma}{=} (v_1, v_2) \sqsubseteq_{V_1 \times V_2} \gamma((d_1, d_2))
\]

Then, using Theorem 3.13, the result follows.

The Cartesian product can also be used on lifted sets (Theorem 3.17).

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
\[
\begin{align*}
\alpha(V) &= (\alpha_1(V) \cup \{v_1 \in V_1 \mid \exists v_2 \in V_2 : (v_1, v_2) \in V\}) \\
\alpha_2(V) &= (\{v_2 \in V_2 \mid \exists v_1 \in V_1 : (v_1, v_2) \in V\})
\end{align*}
\]
\[
\gamma((d_1, d_2)) = \gamma_1(d_1) \times \gamma_2(d_2)
\]
and \( V \subseteq V_1 \times V_2 \) and \((d_1, d_2) \in D_1 \times D_2\).
3.4 Constructing Galois Connections

**Proof.** Assume that \( \langle \alpha_1 : \mathcal{P}(V_1) \to D_1, \gamma_1 : D_1 \to \mathcal{P}(V_1) \rangle \) and \( \langle \alpha_2 : \mathcal{P}(V_2) \to D_2, \gamma_2 : D_2 \to \mathcal{P}(V_2) \rangle \) 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

\[
\alpha(V) \subseteq (d_1, d_2) \quad \text{Def.} \quad \alpha \quad (\alpha_1(V'_1), \alpha_2(V'_2)) \subseteq (d_1, d_2) \\
\Longleftrightarrow \quad \text{calc.} \quad \alpha_1(V'_1) \subseteq_1 d_1 \land \alpha_2(V'_2) \subseteq_2 d_2 \\
\Longleftrightarrow \quad \text{Th. 3.13} \quad V'_1 \subseteq \gamma_1(d_1) \land V'_2 \subseteq \gamma_2(d_2) \\
\Longleftrightarrow \quad \text{calc.} \quad V'_1 \times V'_2 \subseteq \gamma_1(d_1) \times \gamma_2(d_2) \\
\Longleftrightarrow \quad \text{Def.} \quad \gamma \quad V'_1 \times V'_2 \subseteq V_2 \subseteq \gamma((d_1, d_2))
\]

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 \( \alpha_{\mathcal{P}} \)):**

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

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

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

**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:

\[
\alpha_{\mathcal{P}}(V'') \quad \text{Def.} \quad \alpha_{\mathcal{P}}(V) = \{ \alpha(v) \mid v \in V'' \} \\
\quad \text{calc.} \quad \{ \alpha(v) \mid v \in V' \cup (V'' \setminus V') \} \\
\quad \text{calc.} \quad \{ \alpha(v) \mid v \in V' \} \cup \{ \alpha(v) \mid v \in V'' \setminus V' \} \\
\quad \text{calc.} \quad \{ \alpha(v) \mid v \in V' \} \\
\quad \text{Def.} \quad \alpha_{\mathcal{P}}(V') = \alpha_{\mathcal{P}}(V')
\]
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 $\gamma_\mathcal{D}$ is monotone. ■

**Lemma 3.19 (Monotonicity of $\gamma_\mathcal{D}$):**
The function $\gamma_\mathcal{D} : \mathcal{P}(D) \rightarrow \mathcal{P}(V)$, defined as

$$\gamma_\mathcal{D}(D') = \{ v \in V \mid \alpha(v) \in D' \}$$

where $D' \subseteq D$, $\alpha$ is monotone and $\gamma : D \rightarrow V$, is monotone.

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

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

$$\gamma_\mathcal{D}(D'') \overset{\text{Def.}}{=} \gamma_\mathcal{D} \{ v \in V \mid \alpha(v) \in D'' \}$$

$$\overset{\text{calc.}}{=} \{ v \in V \mid \alpha(v) \in D' \cup (D'' \setminus D') \}$$

$$\overset{\text{calc.}}{=} \{ v \in V \mid \alpha(v) \in D' \} \cup \{ v \in V \mid \alpha(v) \in D'' \setminus D' \}$$

$$\overset{\text{calc.}}{=} \{ v \in V \mid \alpha(v) \in D' \}\cup \{ v \in V \mid \alpha(v) \in D'' \setminus D' \}$$

$$\overset{\text{Def.}}{=} \gamma_\mathcal{D}(D')$$

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

Thus, $\gamma_\mathcal{D}(D') \subseteq \gamma_\mathcal{D}(D'')$, and hence it has been shown that $\gamma_\mathcal{D}$ is monotone. ■

**Theorem 3.20 (Galois connection – Double lifting):**
If $\langle \alpha : V \rightarrow D, \gamma : D \rightarrow V \rangle$ is a Galois connection, then so is $\langle \alpha_\mathcal{D} : \mathcal{P}(V) \rightarrow \mathcal{P}(D), \gamma_\mathcal{D} : \mathcal{P}(D) \rightarrow \mathcal{P}(V) \rangle$, where

$$\begin{cases} 
\alpha_\mathcal{D}(V') = \{ \alpha(v) \mid v \in V' \} \\
\gamma_\mathcal{D}(D') = \{ v \in V \mid \alpha(v) \in D' \}
\end{cases}$$

and $V' \subseteq V$ and $D' \subseteq D$.

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

Since $\alpha_\mathcal{D}$ and $\gamma_\mathcal{D}$ 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:

$$\gamma_\mathcal{P}(\alpha_\mathcal{P}(V')) = \{v \in V | \alpha(v) \in \{\alpha'(v') | v' \in V'\}\}$$

Assume that $v'' \in V'$, then it must be that $\alpha(v'') \in \{\alpha'(v') | v' \in V'\}$ 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:

$$\alpha_\mathcal{P}(\gamma_\mathcal{P}(D')) = \{\alpha(v) | v \in \{v' \in V | \alpha(v') \in D'\}\}$$

Assume that $d \in \alpha_\mathcal{P}(\gamma_\mathcal{P}(D'))$. Then it must be that $\exists v \in \{v' \in V | \alpha(v') \in D'\}: d = \alpha(v)$. 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 $\langle \alpha_\mathcal{P}' : \mathcal{P}(V) \to \mathcal{P}(D), \gamma_\mathcal{P}' : \mathcal{P}(D) \to \mathcal{P}(V) \rangle$ is not a Galois connection, where

$$\begin{align*}
\alpha_\mathcal{P}'(V') &= \{\alpha(v) | v \in V'\} \\
\gamma_\mathcal{P}'(D') &= \{\gamma(d) | d \in D'\}
\end{align*}$$

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 $\alpha_\mathcal{P}$ and $\gamma_\mathcal{P}$, it clearly follows that they are monotone since $\alpha$ and $\gamma$ are (cf. Lemma 3.18).

By way of contradiction, assume that $\langle \alpha_\mathcal{P}, \gamma_\mathcal{P} \rangle$ is a Galois connection. Then, by Definition 3.9, $\gamma_\mathcal{P}(\alpha_\mathcal{P}(V')) \supseteq V'$. A closer look at $\gamma_\mathcal{P}(\alpha_\mathcal{P}(V'))$ reveals that:

$$\gamma_\mathcal{P}(\alpha_\mathcal{P}(V')) = \{\gamma(d) | d \in \{\alpha(v) | v \in V'\}\}$$
Assume that $v' \in V'$, then $v' \in \gamma'((V'))$ since $\gamma'((V')) \supseteq V'$. This means that $\exists d' \in \{\alpha(v) \mid v \in V'\} : d' = \alpha(v')$ and hence, for this $d'$, $\exists v'' \in \{\gamma(d) \mid d \in \{\alpha(v) \mid v \in V'\}\} : v' = v'' = \gamma(d') = \gamma(\alpha(v'))$.

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

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 $\langle\alpha : V \rightarrow D, \gamma : D \rightarrow V\rangle$ is a Galois connection, then so is $\langle\alpha' : (S \rightarrow V) \rightarrow (S \rightarrow D), \gamma' : (S \rightarrow D) \rightarrow (S \rightarrow V)\rangle$ for some set, $S$, where:

$$
\begin{cases}
\alpha'(f) = \alpha \circ f \\
\gamma'(g) = \gamma \circ g
\end{cases}
$$

**Proof.** Assume that $\langle\alpha : V \rightarrow D, \gamma : D \rightarrow V\rangle$ is a Galois connection and that $S$ is a set. Note that $S \rightarrow V$ and $S \rightarrow D$ 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.

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 (otrwp is short for otherwise).

**Lemma 3.23 (Monotonicity of $\gamma_s$):**
The function $\gamma_s : (S \rightarrow D) \rightarrow \mathcal{P}(S \rightarrow V)$, defined as

$$
\gamma_s(d) = \begin{cases}
S \rightarrow V & \text{if } d = \top \\
\emptyset & \text{if } d = \bot \\
\{\lambda s \in S \mid v \in \gamma(d) s\} & \text{otrwp}
\end{cases}
$$
for some set $S$ and complete lattices $V$ and $D$, is monotone, given that $\gamma : D \to \mathcal{P}(V)$ is a monotone function and $d \in S \to D$.

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

Assume that $\gamma : D \to \mathcal{P}(V)$ is a monotone function, $d', d'' \in S \to D$ and that $d' \not\subseteq d'' \wedge d' \neq \bot \wedge d'' \neq \top$. Then, by definition:

\[
\begin{cases}
\gamma_s(d') = \{ \lambda s \in S.v \mid v \in \gamma(d's) \} \\
\gamma_s(d'') = \{ \lambda s \in S.v \mid v \in \gamma(d''s) \}
\end{cases}
\]

Since $\gamma$ is monotone, it must be that $\forall s \in S : \gamma(d's) \subseteq \gamma(d''s)$. This means that

\[
\gamma_s(d'') = \{ \lambda s \in S.v \mid v \in \gamma(d's) \cup (\gamma(d''s) \setminus \gamma(d's)) \} \\
\supseteq \{ \lambda s \in S.v \mid v \in \gamma(d's) \} \cup \{ \lambda s \in S.v \mid v \in (\gamma(d''s) \setminus \gamma(d's)) \} \\
= \gamma_s(d') \cup \{ \lambda s \in S.v \mid v \in (\gamma(d''s) \setminus \gamma(d's)) \}
\]

and thus, trivially, $\gamma_s(d') \subseteq \gamma_s(d'')$. $\blacklozenge$

**Theorem 3.24 (Galois connection – Lifted function space):**

If $\langle \alpha : \mathcal{P}(V) \to D, \gamma : D \to \mathcal{P}(V) \rangle$ is a Galois connection, then so is $\langle \alpha_s : \mathcal{P}(S \to V) \to (S \to D), \gamma_s : (S \to D) \to \mathcal{P}(S \to V) \rangle$, for some set $S$, where

\[
\alpha_s(V') = \begin{cases} 
\top & \text{if } V' = S \to V \\
\bot & \text{if } V' = \emptyset \\
\{ \lambda s \in S.\alpha(\{ v' s \mid v' \in V' \}) \} & \text{otrsw}
\end{cases}
\]

\[
\gamma_s(d) = \begin{cases} 
S \to V & \text{if } d = \top \\
\emptyset & \text{if } d = \bot \\
\{ \lambda s \in S.v \mid v \in \gamma(d s) \} & \text{otrsw}
\end{cases}
\]

and $V' \subseteq S \to V$ and $d \in S \to D$. $\blacklozenge$

**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.

$\alpha_s(V') \subseteq d$  

$\lambda s \in S. \alpha(\{v' s \mid v' \in V'\}) \subseteq d$  

$\gamma_s(\lambda s \in S. \alpha(\{v' s \mid v' \in V'\})) \subseteq \gamma_s(d)$  

$\{\lambda s \in S. v \mid v \in \gamma(\lambda s \in S. \alpha(\{v' s \mid v' \in V'\}))\} \subseteq \gamma_s(d)$  

$\lambda s \in S. v \in \gamma(\alpha(\{v' s \mid v' \in V'\})) \subseteq \gamma_s(d)$  

$\lambda s \in S. v \in \{v' s \mid v' \in V'\} \subseteq \gamma_s(d)$  

$\lambda s \in S. v \in \{v' s \mid v' \in V'\} \subseteq \gamma_s(d)$  

$\{v' \mid v' \in V'\} \subseteq \{\lambda s \in S. (v' s) \mid v' \in V'\} \subseteq \gamma_s(d)$  

$\{v' \mid v' \in V'\} \subseteq \gamma_s(d)$  

$\gamma_s(d)$  

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 $\langle \alpha : V \rightarrow D, \gamma : D \rightarrow V \rangle$ is a Galois connection, then so is $\langle \alpha' : (S \times V) \rightarrow (S \times D), \gamma' : (S \times D) \rightarrow (S \times V) \rangle$, for some set $S \ni s$ (with the partial order $\preceq$), where

$$
\begin{align*}
\alpha'((s, v)) &= (s, \alpha(v)) \\
\gamma'((s', d)) &= (s', \gamma(d))
\end{align*}
$$

and $(s, v) \in S \times V$ and $(s', d) \in S \times D$. The top elements, $\top_V$ and $\top_D$, 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 $\langle \alpha' : (V \times S) \rightarrow (D \times S), \gamma' : (D \times S) \rightarrow (V \times S) \rangle$ are defined similarly.

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

First note that:
\[
\begin{align*}
\gamma'((\alpha'(\top')) &= \gamma'(\top') = \top' \sqsubseteq \top' \\
\gamma'((\alpha'(\bot')) &= \gamma'(\bot') = \bot' \sqsubseteq \bot'
\end{align*}
\]
Then, using Theorem 3.13, the result follows.

Now, using Theorem 3.13, the result follows.

The proof for \( \langle \alpha', (V \times S) \rightarrow (D \times S), \gamma' : (D \times S) \rightarrow (V \times S) \rangle \) 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 [35].

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 \( \gamma : \tilde{D} \rightarrow D \).

3. Define the partial order \( \sqsubseteq \) for \( \tilde{D} \).

4. The greatest lower bound \( \sqcap \) and the least upper bound \( \sqcup \) must exist for all subsets of \( \tilde{D} \). Then, by definition, \( \langle \tilde{D}, \sqsubseteq, \sqcup, \sqcap, \bot, \top \rangle \) is a complete lattice.

5. Define the abstraction function \( \alpha : D \rightarrow \tilde{D} \), which must be monotone.

Assuming that the domains \( D \) and \( \tilde{D} \) and the monotone concretization function, \( \gamma \), are defined, the partial ordering \( \sqsubseteq \) can easily be defined as given by Definition 3.26 [35].
Definition 3.26 (Partial order):
\( \subseteq \) is a partial order for the domain \( \bar{D} \) iff \( \forall d_1, d_2 \in \bar{D} : (d_1 \subseteq d_2 \iff \gamma(d_1) \subseteq \gamma(d_2)) \).

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 [35].

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

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

The abstraction function \( \alpha \) can be defined based on the definition of the greatest lower bound operator as given by Definition 3.29 [35].

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:

\[
\alpha(d) = \bigcap \{ \tilde{\bar{d}} \mid d \subseteq \gamma(\tilde{\bar{d}}) \}
\]

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

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 [35].

Definition 3.30 (Alternative definition – Concretization function, \( \gamma \)):
Given two domains \( D \) and \( \bar{D} \) and a monotone abstraction function \( \alpha : D \rightarrow \bar{D} \), the concretization function \( \gamma : \bar{D} \rightarrow D \) is defined by:

\[
\gamma(\tilde{\bar{d}}) = \bigcup \{ d \mid \alpha(d) \subseteq \tilde{\bar{d}} \}
\]

where \( d \in D \) and \( \tilde{\bar{d}} \in \bar{D} \).
3.6 The Interval Domain

One example of an abstract domain for values is the interval domain \([29, 35, 81]\). 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 \(\text{Intv} = \{\perp_{\text{int}}, \top_{\text{int}}\} \cup \{[n_1, n_2] \mid n_1 \leq n_2 \land n_1, n_2 \in \text{Val}\}\), 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 \(\gamma_{\text{int}} : \text{Intv} \rightarrow \mathcal{P}(\text{Val})\) is given by Definition 3.32.

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

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

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

**Definition 3.33 (Partial order for intervals):**

\[
\begin{cases} 
i \sqsubseteq_{\text{int}} \top_{\text{int}} \\
\perp_{\text{int}} \sqsubseteq_{\text{int}} i \\
[n_1, n_2] \sqsubseteq_{\text{int}} [n'_1, n'_2] \iff n'_1 \leq n_1 \land n_2 \leq n'_2
\end{cases}
\]

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

**Definition 3.34 (Greatest lower bound for intervals):**

\[
\begin{cases} 
i \sqcap_{\text{int}} \top_{\text{int}} = \top_{\text{int}} \sqcap_{\text{int}} i = i \\
i \sqcap_{\text{int}} \perp_{\text{int}} = \perp_{\text{int}} \sqcap_{\text{int}} i = \perp_{\text{int}} \\
[n_1, n_2] \sqcap_{\text{int}} [n'_1, n'_2] = \\
\begin{cases} 
\left[\max\{\{n_1, n'_1\}\}, \min\{\{n_2, n'_2\}\}\right] & \text{if } \max\{\{n_1, n'_1\}\} \leq \min\{\{n_2, n'_2\}\} \\
\perp_{\text{int}} & \text{otrw}
\end{cases}
\end{cases}
\]

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

**Definition 3.35 (Least upper bound for intervals):**

$$
\begin{cases}
\bigcup_{\text{int}} \top_{\text{int}} = \top_{\text{int}} \\
\bigcap_{\text{int}} \bot_{\text{int}} = \bot_{\text{int}} \\
[n_1, n_2] \sqcup_{\text{int}} [n_1', n_2'] = [\min\{n_1, n_1'\}, \max\{n_2, n_2'\}]
\end{cases}
$$

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

**Definition 3.36 (Abstraction to interval):**

$$\alpha_{\text{int}}(V) = \begin{cases}
\top_{\text{int}} & \text{if } V = \mathbb{Z} \cup \{-\infty, \infty\} \\
\bot_{\text{int}} & \text{if } V = \emptyset \\
[\min(V), \max(V)] & \text{otrw}
\end{cases}$$

To show that $(\alpha_{\text{int}}, \gamma_{\text{int}})$ 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 $i = \bot_{\text{int}}$ 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 $\forall V, V' \in \mathcal{P}(\text{Val}) : (V \subseteq V' \Rightarrow \alpha_{\text{int}}(V) \subseteq_{\text{int}} \alpha_{\text{int}}(V'))$. 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, \( [n_1, n_2] \subseteq \text{int} [n'_1, n'_2] \) (Definition 3.33), which means that \( \alpha_{\text{int}}(V) \subseteq \text{int} \alpha_{\text{int}}(V') \).

**Theorem 3.39 (Galois insertion – Intervals):**

\( \langle \alpha_{\text{int}} : \mathcal{P}(\text{Val}) \rightarrow \text{Intv}, \gamma_{\text{int}} : \text{Intv} \rightarrow \mathcal{P}(\text{Val}) \rangle \) 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 \cite{81}.

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 \( i = \bot_{\text{int}} \), then \( \gamma_{\text{int}}(i) = \emptyset \). Thus, \( \alpha_{\text{int}}(\gamma_{\text{int}}(i)) = \alpha_{\text{int}}(\emptyset) = \bot_{\text{int}} = i \).

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

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

- 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, \( \gamma_{\text{int}}(\alpha_{\text{int}}(V)) = \gamma_{\text{int}}([\min(V), \max(V)]) = \{n \in \text{Val} \mid \min(V) \leq n \leq \max(V)\} \supseteq V \).  

\[\blacksquare\]
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 4.3). 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
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 4.3). 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. $\Pi \in \text{Prg}$ denotes a program, which simply is a (static) set of threads, i.e. $\Pi = \text{Thrd} \in \mathcal{P} (\text{ThrdID} \times \text{Stm}) = \text{Prg}$, 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, $l \in \text{Lbl}_T$, and stored in an array-like fashion in ascending order of their labels. $a \in \text{Aexp}$ and $b \in \text{Bexp}$ denote an arithmetic 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 \texttt{lock} and re-
leased using unlock. Values can be transferred between variables and registers using load and store. Conditional branching is performed using if. A register is assigned a value using :=. A no-operation is performed using skip. And, halt stops the execution of the issuing thread. 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.

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, there is an instance of each of the following states.

\( \text{pc}_T : \text{Lbl}_T \rightarrow \text{Val} \) – a program counter that keeps track of which label (i.e. statement) within the thread \( T \) that is active. Note that \( \text{Lbl}_T = \mathbb{N}^+ \) (i.e. the positive integers).

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

\( t^*_T : \text{Time} \) – 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.

\( \mathfrak{X} : \text{Var} \rightarrow \text{Thrd} \rightarrow \mathcal{P}(\text{Val} \times \text{Time}) \) – 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.

\( \mathfrak{I} : \text{Lck} \rightarrow (\text{Lck}_{\text{att}} \times \text{Thrd}_{\perp} \times \text{Time} \times \text{Thrd}_{\perp} \times \text{Time}) \) – a mapping from locks to their values; i.e. a state \( (\text{Lck}_{\text{att}} = \{\text{unlocked}, \text{locked}\}) \), 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 $\bot_{\text{thrld}}$.

**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 4.3). 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 $\exists$ and $\forall$ 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 included in the concrete case as well.

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, $\text{ax}c_{\text{in}}^{\text{T}} \in \text{axConf}_{\text{T}}^{\text{in}}$ and $\text{ax}c_{\text{T}}^{\text{out}} \in \text{axConf}_{\text{T}}^{\text{out}}$, are defined for $T \in \text{Thrld}$. $\text{ax}c_{\text{T}}^{\text{in}}$ is used as input to the semantic rules for axiom
The above listed states, together with the threads of the program, will be referred to as a program state or configuration, denoted by $\text{Conf}$. The concrete case is the current owner of each lock (cf. Tables 4.2 and 4.3). Note that the functions $\text{STM}$ and $\text{STT}$, $\text{OWN}$, $\text{DL}$, $\text{POWN}$ and $\text{REL}$ 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 all other threads. The semantic rule for a set of threads (i.e. the program) is described by the relation $\rightarrow_{\text{prg}} : \text{Conf} \times \text{Conf} \rightarrow \{\text{true}, \text{false}\}$, which is formally defined in Table 4.3. $\rightarrow_{\text{prg}}$ describes the semantics of a single statement within a thread when considered in isolation from all other threads.

The semantic rules for individual statements within a thread, the language axioms, are described by the relation $\rightarrow_{\text{ax}} : \text{axConf}_{\text{in}} \times \text{axConf}_{\text{out}} \rightarrow \{\text{true}, \text{false}\}$, which is formally defined in Table 4.2. $\rightarrow_{\text{ax}}$ describes the semantics of a single statement within a thread.

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 $c \xrightarrow{\text{prg}} c'$. Then, a succeeding configuration, $c'' \in \text{Conf}$, to $c'$ is given by $c' \xrightarrow{\text{prg}} c''$, and so on.

Note that $\rightarrow_{\text{ax}}$ and $\xrightarrow{\text{prg}}$ are relations, not functions, because one single input configuration can have several outputs; e.g. if two or more threads execute
Table 4.2: $\langle T, pc, τ, x, \|=l, t \rangle \rightarrow_{ax} \langle pc', \tau', x', \|=l' \rangle$, the semantics of concrete axiom transitions.

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

where $A(r, \tau, x, \|=l) = \tau[r \mapsto v]$ for some $v$, such that

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

\[
v \in \gamma_{\text{int}}([-\infty, \infty])
\]
Table 4.3: \( c \rightarrow_{prg} c' \), the semantics of concrete program transitions.

\[
\begin{align*}
\text{Thrd}_{exe} \neq \emptyset \land \forall T \in \text{Thrd}_{exe} : (T, pc_T, t_T, x, x', t'_T) & \rightarrow (pc'_T, t'_T, x'_T, t''_T) \\
c @ ([T, pc_T, t_T, t''_T] \in \text{Thrd}, x, x') & \rightarrow c' @ ([T, (T \in \text{Thrd}_{exe} \wedge T'' \neq pc_T), (T \in \text{Thrd}_{exe} \wedge T'' = pc_T), \text{Thrd}_{exe} @ \text{Thrd}_{exe}] pc'_T, t'_T, x'_T, t''_T)
\end{align*}
\]

where

\[
t = \min(\{t''_T + \text{TIME}(c, T) \mid T \in \text{Thrd} \land \text{STM}(T, pc_T) \neq [\text{halt}]^{pc_T}\})
\]

\[
\text{Thrd}_{exe} = \{T \in \text{Thrd} \mid T = \text{TIME} + \text{TIME}(c, T) \wedge \text{STM}(T, pc_T) \neq [\text{halt}]^{pc_T}\}
\]

\[
t''_T = \begin{cases} t''_T + \text{TIME}(c, T) & \text{if } T \in \text{Thrd}_{exe} \\ t''_T & \text{otherwise} \end{cases}
\]

\[
x' = \begin{cases} x & \text{if } \text{Thrd}_{exe} = \emptyset \\ \lambda T' \in \text{Thrd}.(T = T' \wedge (x', x) = \emptyset) & \text{otherwise} \end{cases}
\]

where \( T' \) is one of the threads in \( \text{Thrd}_{exe} = \{T \in \text{Thrd}_{exe} \mid \exists r \in \text{Reg}_T : \text{STM}(T, pc_T) = [\text{store} \ r \ \text{to} \ x]\} \)

\[
\ll'' \ lck = \begin{cases} \ll \ lck \ \\ \ll \ lck \ \\ \ll \ lck \ \\ \ll \ lck \ \\ \ll \ lck \ \\ \ll \ lck \ \end{cases}
\]

\[
\ll' \ lck = \begin{cases} \ll'' \ lck \ \\ \ll'' \ lck \ \\ \ll'' \ lck \ \\ \ll'' \ lck \ \\ \ll'' \ lck \ \\ \ll'' \ lck \ \end{cases}
\]

4.2 Semantics
Table 4.4: Definition of STM and LABELS.

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

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

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

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

Table 4.5: Definition of STT, OWN, DL, POWN and REL.

\[
\text{STT} : (\text{Lck}_{\text{stt}} \times \text{Thrd}_\bot \times \text{Time} \times \text{Thrd}_\bot \times \text{Time}) \rightarrow \text{Lck}_{\text{stt}}
\]

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

\[
\text{OWN} : (\text{Lck}_{\text{stt}} \times \text{Thrd}_\bot \times \text{Time} \times \text{Thrd}_\bot \times \text{Time}) \rightarrow \text{Thrd}_\bot
\]

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

\[
\text{DL} : (\text{Lck}_{\text{stt}} \times \text{Thrd}_\bot \times \text{Time} \times \text{Thrd}_\bot \times \text{Time}) \rightarrow \text{Time}
\]

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

\[
\text{POWN} : (\text{Lck}_{\text{stt}} \times \text{Thrd}_\bot \times \text{Time} \times \text{Thrd}_\bot \times \text{Time}) \rightarrow \text{Thrd}_\bot
\]

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

\[
\text{REL} : (\text{Lck}_{\text{stt}} \times \text{Thrd}_\bot \times \text{Time} \times \text{Thrd}_\bot \times \text{Time}) \rightarrow \text{Time}
\]

\[
\text{REL}((u,T,t,T',t')) = t'
\]
lock lck, where \( \text{STT}([]) = \text{unlocked} \), then \( lck \) is assigned to one of these threads by \( \text{prg} \) in a non-deterministic fashion. \( \text{ax} \), \( \text{prg} \) and the semantic behavior of each statement in PPL are further described below. Since not all parts of Tables 4.2 and 4.3 are relevant for all statements, the \( \text{ax} \) and \( \text{prg} \) relations (i.e. the semantics of PPL) will be explained based on what statement is considered.

**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 \( \text{prg} \) 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 \( c \in \text{Conf} \) and \( c' \in \text{Conf} \), where \( c \rightarrow c' \) and \( \forall T \in \text{Thrd} : t'^T \leq t^T \).

**PROOF.** Assume that \( c \in \text{Conf} \) and \( c' \in \text{Conf} \) are such that \( c \rightarrow c' \). From Table 4.3, it is apparent that there are two possibilities for the value of \( t'_T \).

If \( t'^T + \text{TIME}(c, T) = \min\{t'^T, T \in \text{Thrd} \} \neq [\text{halt}][pc_T] \), then \( t'^T = t'^T + \text{TIME}(c, T) \). Thus, \( t'^T \geq t'^T \) (Assumption 4.1).

If \( t'^T + \text{TIME}(c, T) \neq \min\{t'^T, T \in \text{Thrd} \} \neq [\text{halt}][pc_T] \), then \( t'^T = t^T \).

Thus, it must be that \( \forall T \in \text{Thrd} : t'^T \leq t'^T \).
Assumption 4.3 (TIME is non-zero when spin-locking):
It is assumed that $\forall c \in (\mathit{Thr}_T \times \mathit{Reg} \times x, \emptyset) \in \mathit{Conf} : \forall T \in \mathit{Thr}_c :
(\exists \mathit{lck} \in \mathit{Lck} : \mathit{STM}(T, pc_T) = \mathit{lock}\mathit{lck}[^{pc_T} \land \mathit{OWN}(\mathit{lck}) \not\in \{\perp_{\mathit{thr}_T}, T\}) \Rightarrow
0 < \mathit{TIME}(c, T)).$ □

The set of threads to execute, $\mathit{Thr}_{\mathit{exe}}$, (i.e. the threads whose active statements will take effect) on a transition from a given configuration, $c\mathit{Thr}_T \times \mathit{Reg} \times x, \emptyset) \in \mathit{Conf}$, is determined based on $t^d$ and $\mathit{TIME}(c, T)$ for each thread $T \in \mathit{Thr}_c$. It simply consists of the threads that will execute their active statements at the nearest point in time, denoted by $t$ in Table 4.3. Only threads in $\mathit{Thr}_{\mathit{exe}}$ affect the system state upon a transition between configurations.

An illustration of how $\mathit{Thr}_{\mathit{exe}}$ is determined is given in Figure 4.6. For $c_1$ in Figure 4.6a, $t = t^d_{T_2} + \mathit{TIME}(c_1, T_2) = t^d_{T_3} + \mathit{TIME}(c_1, T_3) = 6$ and $t^d_{T_1} + \mathit{TIME}(c_1, T_1) = 10$. Thus, $\mathit{Thr}_{\mathit{exe}} = \{T_2, T_3\}$ and $t^d_{T_2} = t^d_{T_3} = 6$, while $t^d_{T_1} = t^d_{T_1} = 7$. For $c_2$ in Figure 4.6b, $\mathit{Thr}_{\mathit{exe}}$ is determined in a similar manner (note that $c_1 \not\rightarrow c_2$).

The statements $\mathit{halt}$ and $\mathit{skip}$

As previously discussed, $\mathit{halt}$ stops the execution of a thread and $\mathit{skip}$ is a no-operation. This is implemented by letting the semantic rule for $\mathit{halt}$ return the thread’s input state without modifying it, which means that the issuing thread will still be executing the same $\mathit{halt}$-statement in the next iterative step; thus the thread halts. Note that threads issuing a $\mathit{halt}$-statement are not included in $\mathit{Thr}_{\mathit{exe}}$, however. The rule for the $\mathit{skip}$-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 $\phi : \mathit{Aexp} \rightarrow (\mathit{Reg} \rightarrow \mathit{Val}) \rightarrow \mathit{Val}$. $\phi$ evaluates arithmetic expressions based on a given register state as defined in Table 4.7. It should be noted that the calculations $\infty - \infty$, $\infty/0$, $0 \cdot \infty$ and $0/0$ are undefined and hence not supported by $\phi$. The operation $x/0$, where $x \in \mathit{Val}$, results in $\infty$ if $0 < x$ and $-\infty$ if $x < 0$. 
dependent on the register values in the input register state and is determined by the value of the arithmetic expression $a$. This is determined in a similar manner (note the $-\text{statement}$ in the next iterative step; $c_1 \overset{\text{reg}}{\rightarrow} c_2$).

**Figure 4.6:** Illustration of how $\text{Thrd}_{\text{exe}}$ is determined ($c_1 \overset{\text{reg}}{\rightarrow} c_2$).

**Table 4.7:** Semantics of concrete evaluation of arithmetic expressions.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n$</td>
<td>$\mathcal{S}[n] = n$</td>
</tr>
<tr>
<td>$a_1 - a_2$</td>
<td>$\mathcal{S}[a_1 - a_2] = \mathcal{S}[a_1] - \mathcal{S}[a_2]$</td>
</tr>
<tr>
<td>$a_1 * a_2$</td>
<td>$\mathcal{S}[a_1 * a_2] = \mathcal{S}[a_1] * \mathcal{S}[a_2]$</td>
</tr>
<tr>
<td>$a_1 + a_2$</td>
<td>$\mathcal{S}[a_1 + a_2] = \mathcal{S}[a_1] + \mathcal{S}[a_2]$</td>
</tr>
<tr>
<td>$a_1 / a_2$</td>
<td>$\mathcal{S}[a_1 / a_2] = \frac{\mathcal{S}[a_1]}{\mathcal{S}[a_2]}$</td>
</tr>
</tbody>
</table>
Table 4.8: Semantics of concrete evaluation of boolean expressions.

<table>
<thead>
<tr>
<th>Boolean Expression</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>( B[\text{true}] \equiv \text{true} )</td>
</tr>
<tr>
<td>( b_1 \land b_2 )</td>
<td>( B[b_1 \land b_2] \equiv B[b_1] \land B[b_2] )</td>
</tr>
<tr>
<td>false</td>
<td>( B[\text{false}] \equiv \text{false} )</td>
</tr>
<tr>
<td>( a_1 = a_2 )</td>
<td>( B[a_1 = a_2] \equiv A[a_1] = A[a_2] )</td>
</tr>
<tr>
<td>( !b )</td>
<td>( B[!b] \equiv -B[b] )</td>
</tr>
<tr>
<td>( a_1 &lt; a_2 )</td>
<td>( B[a_1 &lt; a_2] \equiv A[a_1] \leq A[a_2] )</td>
</tr>
</tbody>
</table>

The conditional statement if

The statement if \( b \) goto \( l \) performs conditional branching. If the boolean expression \( b \) evaluates to true, the issuing thread’s \( pc \) is set to \( l \). If \( b \) evaluates to 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 \( B : \text{Bexp} \rightarrow (\text{Reg} \rightarrow \text{Val}) \rightarrow \text{Bool} \). \( B \) evaluates boolean expressions based on a given register state as defined in Table 4.8.

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, \( \chi \), has to be defined accordingly. This is why the definition of \( \chi \) 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 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 \( \tilde{\chi}' \) in Table 4.3).

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 $\mathcal{R}$ 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, $\llbracket$, 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 $\llbracket$ 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 4.3. Note that a consequence of this is that, in a given configuration, $c \preceq \langle [T, pc_T, s_T, t^a_T]_{T \in \text{Thrd}}, \llbracket, ] \rangle$, $\text{STT}(\llbracket lck) = \text{locked}$ whenever $\text{OWN}(\llbracket lck) \neq \bot_{\text{thrd}}$ (cf. Lemma 4.5). Also note that $(\text{STT}(\llbracket lck) = \text{locked} \land \text{OWN}(\llbracket lck) = \bot_{\text{thrd}}) \lor \exists T \in \text{Thrd} : \dot{t}^a_T + \text{TIME}(c, T) < \text{REL}(\llbracket lck)$ implies that the given configuration is actually not valid (cf. Definition 4.4).

The lock-statement has the same behavior as the halt-statement as long as the issuing thread is not assigned the given lock; i.e. the issuing thread will wait for its turn to acquire the lock. If the issuing thread is assigned the given lock (which is done within $\xrightarrow{prg}$, using the intermediate lock state $\llbracket''$), lock is defined to basically take the lock (i.e. set its state to locked) and advance the thread’s $pc$. Only one single thread can be assigned a given lock at any point in time.

Note that $\llbracket''$ 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. $\llbracket''$ is only used as an input to the $\xrightarrow{ax}$-relation. The resulting lock state, $\llbracket'$, is in turn based on the output lock states from $\xrightarrow{ax}$, $\llbracket''_T$, given that $T$ is the owner of the considered lock in $\llbracket''$.

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, $c \preceq \langle [T, pc_T, s_T, t^a_T]_{T \in \text{Thrd}}, \llbracket, ] \rangle \in \text{Conf}$, is valid with
Definition 4.4), then so is $c$ result in a valid configuration since STM
\[ \exists \text{ respecting to the lock state}, \bot, \text{ iff} \]

\[
\forall \text{lck} \in \text{Lck} : ((\text{STT}(\bot \text{lck}) = \text{locked} \Leftrightarrow \text{OWN}(\bot \text{lck}) \neq \bot_{\text{thr}}) \land \\
(\text{STT}(\bot \text{lck}) = \text{unlocked} \Leftrightarrow \text{OWN}(\bot \text{lck}) = \bot_{\text{thr}}) \land \\
\forall \text{T} \in \text{Thrd} : \text{REL}(\bot \text{lck}) \leq \text{TIME}(c, T)) \]

\[\Box\]

Lemma 4.5 ($\rightarrow_{\text{prg}}$ preserves lock state validity): Given that the configuration $c @ \langle [T, pc_T, z_T, t_T^T]_{T \in \text{Thrd}}, x, \bot \rangle \in \text{Conf}$ is valid (cf. Definition 4.4), then so is $c' @ \langle [T, pc'_T, z'_T, t'_T^T]_{T \in \text{Thrd}}, x', \bot' \rangle \in \text{Conf}$, whenever $c \rightarrow_{\text{prg}} c'$.

PROOF. From Table 4.2, it is apparent that the possible axiom output lock states, called $\bot_1$ in Table 4.3, given an input lock state, called $\bot''$ in Table 4.3, are

1. $\bot''[\text{lck}] \rightarrow \langle \text{locked}, T, \text{DL}(\bot'' \text{lck}), \text{OWN}(\bot'' \text{lck}), \text{REL}(\bot'' \text{lck}) \rangle$, whenever $\text{STM}(T, pc_T) = [\text{lock lck}]^{pc_T} \land \text{OWN}(\bot'' \text{lck}) = T$,

2. $\bot''[\text{lck}] \rightarrow \langle \text{unlocked}, \bot_{\text{thr}}, \text{DL}(\bot'' \text{lck}), T, t_T^T \rangle$, whenever $\text{STM}(T, pc_T) = [\text{unlock lck}]^{pc_T} \land \text{OWN}(\bot'' \text{lck}) = T$, and

3. $\bot''$, otherwise.

Assume that the configurations $c @ \langle [T, pc_T, z_T, t_T^T]_{T \in \text{Thrd}}, x, \bot \rangle \in \text{Conf}$ and $c' @ \langle [T, pc'_T, z'_T, t'_T^T]_{T \in \text{Thrd}}, x', \bot' \rangle \in \text{Conf}$ are such that $c$ is valid and $c \rightarrow_{\text{prg}} c'$.

From Table 4.3, it is apparent that $\rightarrow_{\text{ax}}$ is only applied to axiom input configurations in which $\bot''$ is such that

1. $\bot'' \text{lck} = \bot \text{lck}$, or

2. $\bot'' \text{lck} = \langle \text{unlocked}, T, t_T^T, \text{OWN}(\bot \text{lck}), \text{REL}(\bot \text{lck}) \rangle$.

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 $\exists T' \in \text{Thrd}_{\text{exe}} : (\text{STM}(T', pc_{T'}) = \bot_{\text{lck}})_{\bot_{\text{thr}}} \land \text{OWN}(\bot \text{lck}) = \bot_{\text{thr}})$. Note that the assigned owner, $T \in \text{Thrd}_{\text{exe}}$, is one of the threads executing $\text{lck}$. For thread $T$, the output lock state is $\bot''[\text{lck}] \rightarrow \langle \text{locked}, T, \text{DL}(\bot'' \text{lck}), \text{OWN}(\bot'' \text{lck}), \text{REL}(\bot'' \text{lck}) \rangle$, since $\text{STM}(T, pc_T) = [\text{lock lck}]^{pc_T} \land \text{OWN}(\bot'' \text{lck}) = T$. Hence, $\bot' \text{lck} = \bot[\text{lck}] \rightarrow \langle \text{locked}, T, t_T^T, \text{OWN}(\bot \text{lck}), \text{REL}(\bot \text{lck}) \rangle$.

Since time moves forward for each thread (Lemma 4.2), it is easy to see that $c'$ is valid. \[\Box\]
Lemma 4.6 gives some important properties of the “intermediate” lock state, \(\mathbb{I}''\), defined in Table 4.3, 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 5.10 and 5.11, presented on pages 102 and 106, respectively. To increase the readability of the upcoming proofs in Chapter 5, \(\preceq\) is used here instead of \(\geq\). This is to better correlate with the \(\leq_t\) operator, defined in Definition 5.14.

**Lemma 4.6 (Properties of \(\mathbb{I}''\)):**

If for some valid configuration, \(c \circ \langle [T, pc_T, t_T, r_T]_{T \in \text{Thrd}} \rangle, \mathbb{I} \rangle \in \text{Conf}\), and lock, \(lck \in \text{Lck}\), \(\text{OWN}(\mathbb{I} lck) = \bot_{\text{thrd}} \land \exists T' \in \{T \in \text{Thrd}_{\text{exe}} | \text{STM}(T, pc_T) = [\text{lck} lck]^{pc_T}\} : \text{OWN}(\mathbb{I}' lck) = T'\), where \(\mathbb{I}''\) and \(\text{Thrd}_{\text{exe}}\) are as defined in Table 4.3, then

1. \(\text{STM}(\mathbb{I}'' lck) = \text{unlocked}\),
2. \(\text{DL}(\mathbb{I}'' lck) \not\prec t_{\mathbb{I}'}, \text{and}\)
3. \(t_{\mathbb{I}'} \not\prec \text{REL}(\mathbb{I}'' lck)\).

**PROOF.** For this proof, each of the properties above will be shown based on the definition of \(\rightarrow_{ax}\) and \(\rightarrow_{prg}\), defined in Tables 4.2 and 4.3, respectively.

Assume that for the valid configuration \(c \circ \langle [T, pc_T, t_T, r_T]_{T \in \text{Thrd}} \rangle, \mathbb{I}\rangle \in \text{Conf}\) (cf. Definition 4.4) and some lock, \(lck \in \text{Lck}\), \(\text{OWN}(\mathbb{I} lck) = \bot_{\text{thrd}} \land \exists T' \in \{T \in \text{Thrd}_{\text{exe}} | \text{STM}(T, pc_T) = [\text{lck} lck]^{pc_T}\} : \text{OWN}(\mathbb{I}' lck) = T'\).

1 follows directly from the definition of \(\mathbb{I}''\) in Table 4.3.

Table 4.3 also gives that \(\text{DL}(\mathbb{I}'' lck) = t\) and that \(t_{\mathbb{I}'}, t\), since \(T' \in \text{Thrd}_{\text{exe}}\land \text{STM}(T, pc_T) = [\text{lck} lck]^{pc_T}\). Thus, \(\text{DL}(\mathbb{I}'' lck) = t_{\mathbb{I}'}, \text{and hence}, 2\) has been shown.

For 3, Assumption 4.1 gives that time moves forward when using \(\rightarrow_{prg}\) (Lemma 4.2). Thus, it must be that \(t_{\mathbb{I}'} = t \geq \text{REL}(\mathbb{I} lck) = \text{REL}(\mathbb{I}'' lck)\) (cf. Table 4.3), which concludes the proof.

### 4.3 Collecting Semantics

This section defines the collecting semantics, \(\mathcal{C}(C)\), of a program \([81]\); i.e. the set of all possible semantic configurations given an initial set of configurations, \(C\), (cf. Definition 4.7).
Definition 4.7 (Collecting semantics):

The collecting semantics, \( \mathcal{C}(C) \), of an initial set of configurations, \( C \), is defined as [35]:

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

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 in the case of a nonterminating program; i.e. the accumulated time, \( t^a_T \), for some thread, \( T \in \text{Thrd} \), could increase indefinitely.
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. $\text{Time} = \text{Intv}$. This approach is also taken by Chattopadhyay et al. [18] 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. $\text{Val} = \text{Intv}$, 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, 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 $\mathfrak{r}, \mathfrak{x}, t^a, t^b$ and $c$. The abstraction of these will be referred to as $\tilde{r}, \tilde{x}, \tilde{t}^a, \tilde{t}^b$ and $\tilde{c}$, respectively.

Note that since Time = Intv and Val = 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. $\alpha_t = \alpha_{\text{val}} = \alpha_{\text{int}}$ 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 (Section 5.7) and the abstract semantics (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 $\infty - \infty, \infty/\infty, 0 \cdot \infty$ 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, $\oplus_t$ and $\oplus_{\text{val}}$ both refer to $+_\text{int}$, 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, $(\alpha_{\text{reg}}, \gamma_{\text{reg}})$, between the concrete domain $\mathcal{P}(\text{Reg} \rightarrow \text{Val})$ and the abstract domain $(\text{Reg} \rightarrow \text{Val}) \cup \{\bot_{\text{reg}}, \top_{\text{reg}}\}$ (Theorem 5.6). The concretization function, $\gamma_{\text{reg}}$, partial order, $\sqsubseteq_{\text{reg}}$, greatest lower bound, $\sqcap_{\text{reg}}$, least upper bound, $\sqcup_{\text{reg}}$, and abstraction function, $\alpha_{\text{reg}}$, 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 [35]. $\bar{\xi}$ is the bottom element, $\bot_{\text{reg}}$, if $\forall r \in \text{Reg}: \bar{\xi} r = \bot_{\text{val}}$; i.e. if $\bar{\xi}$ maps all registers to $\bot_{\text{val}}$. The top element, $\top_{\text{reg}}$, corresponds to an abstract mapping for which all registers map to $\top_{\text{val}}$.

Definition 5.1 (Concretization of an abstract register state):

$$
\gamma_{\text{reg}}(\bar{\xi}) = \begin{cases} 
\text{Reg} \rightarrow \text{Val} & \text{if } \bar{\xi} = \top_{\text{reg}} \\
\emptyset & \text{if } \bar{\xi} = \bot_{\text{reg}} \\
\{ \lambda r \in \text{Reg}.v \mid v \in \gamma_{\text{val}}(\bar{\xi} r) \} & \text{otherwise}
\end{cases}
$$
Using Theorems 3.24 and 3.39, it is easy to see that there is indeed a Galois connection,\textsuperscript{60}  

Table 5.1: PPL operators defined for interval arguments.

\[
\begin{align*}
\bot\text{\tiny int} \circ \text{op}_{\text{int}} \ i &= i \text{\tiny int} \circ \bot\text{\tiny int} = \bot\text{\tiny int} & \text{where } \text{op}_{\text{int}} \subseteq \{+\text{int}, -\text{int}, \ast\text{int}, /\text{int}\}
\end{align*}
\]

\[
[l_1, u_1] +_{\text{int}} [l_2, u_2] = \begin{cases}
[l_1 + l_2, u_1 + u_2] & \text{if } -\infty < l_1, l_2, u_1, u_2 < \infty \\
[l_1 + l_2, \infty] & \text{if } (u_1 = \infty \lor u_2 = \infty) \\
[-\infty, u_1 + u_2] & \text{if } (l_1 = -\infty \land l_2 = \infty) \lor (l_1 = \infty \land l_2 = -\infty) \\
[-\infty, \infty] & \text{others}
\end{cases}
\]

\[
[l_1, u_1] -_{\text{int}} [l_2, u_2] = \begin{cases}
[l_1 - u_2, u_1 - l_2] & \text{if } -\infty < l_1, l_2, u_1, u_2 < \infty \\
[l_1 - u_2, \infty] & \text{if } (u_1 = \infty \lor l_2 = -\infty) \lor (l_1 = -\infty \land u_2 = \infty) \lor (l_1 = \infty \land u_2 = -\infty) \\
[-\infty, u_1 - l_2] & \text{if } (l_1 = -\infty \lor u_2 = \infty) \lor (l_1 = -\infty \land l_2 = \infty) \\
[-\infty, \infty] & \text{others}
\end{cases}
\]

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

where $V = \{l_1 l_2, l_1 u_2, u_1 l_2, u_1 u_2\}$
Table 5.1: *Cont.* PPL operators defined for interval arguments.

\[
[l_1, u_1] / \text{int} [l_2, u_2] = \begin{cases}
[l_1/l_2, l_1/u_2, u_1/l_2, u_1/u_2] & \text{if } l_2 \leq 0 \land u_2 > 0 \\
[-\infty, -l_1] & \text{if } l_2 \leq 0 \land u_2 < 0 \\
[-u_1, \infty] & \text{if } l_2 > 0 \land u_2 > 0 \\
[-\infty, \infty] & \text{otherwise}
\end{cases}
\]

where \( V = \{ l_1 / l_2, l_1 / u_2, u_1 / l_2, u_1 / u_2 \} \)

\[
\text{Definition 5.2 (Partial order for abstract register states)}:
\begin{align*}
\text{if } & \exists \, \exists' \text{ s.t. } \exists \subseteq \exists' \subseteq \top \text{ reg} \\
\hspace{1cm} & \exists \subseteq \exists' \subseteq \top \text{ reg} \\
\hspace{1cm} & \exists \subseteq \exists' \subseteq \top \text{ reg}
\end{align*}
\]

\[
\text{Definition 5.3 (Greatest lower bound of abstract register states)}:
\begin{align*}
\top \text{ reg} \cap \exists = \exists \cap \top \text{ reg} = \exists \\
\bot \text{ reg} \cap \exists = \exists \cap \bot \text{ reg} = \exists \\
(\exists \cap \top \text{ reg}) \cap \exists' = (\exists \cap \exists') \cap \top \text{ reg}
\end{align*}
\]

\[
\text{Definition 5.4 (Least upper bound of abstract register states)}:
\begin{align*}
\top \text{ reg} \cup \exists = \exists \cup \top \text{ reg} = \top \text{ reg} \\
\bot \text{ reg} \cup \exists = \exists \cup \bot \text{ reg} = \exists \\
(\exists \cup \top \text{ reg}) \cup \exists' = (\exists \cup \exists') \cup \top \text{ reg}
\end{align*}
\]

\[
\text{Definition 5.5 (Abstraction of a set of register states)}:
\begin{align*}
\alpha_{\text{reg}}(\exists) = \begin{cases}
\top \text{ reg} & \text{if } \exists = \text{ Reg } \rightarrow \text{ Val } \\
\bot \text{ reg} & \text{if } \exists = \emptyset \\
\lambda r \in \text{ Reg} . \alpha_{\text{val}}(\{ r | r \in \exists \}) & \text{ otherwise}
\end{cases}
\end{align*}
\]
Theorem 5.6 (Galois connection – Register states):
\(\langle \alpha_{\text{reg}}, \gamma_{\text{reg}} \rangle\), 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 \(\mathcal{V}_{\text{al}}\) (Theorem 3.39).

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

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 \(\mathcal{A} : \text{Aexp} \rightarrow (\text{Reg} \rightarrow \mathcal{V}_{\text{al}}) \rightarrow \mathcal{V}_{\text{al}}\), 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:

\[
\mathcal{A}^\pi[a]R = \{\mathcal{A}[a]r \mid r \in R\}
\]

The abstract evaluation function is then given by Definition 5.7.

Definition 5.7 (Abstract evaluation of arithmetic expressions):

\[
\mathcal{A}[a] = \alpha_{\text{val}} \circ \mathcal{A}^\pi[a] \circ \gamma_{\text{reg}} = \alpha_{\text{val}} \circ \lambda R.\{\mathcal{A}[a]r \mid r \in R\} \circ \gamma_{\text{reg}} \quad \Box
\]

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 [35, 81].

5.4 Boolean Restriction for Intervals

The function \(\mathcal{B}_{\text{if}} : \text{Bexp} \rightarrow (\text{Reg} \rightarrow \mathcal{V}_{\text{al}}) \rightarrow (\text{Reg} \rightarrow \mathcal{V}_{\text{al}})\) is ideally defined as \(\mathcal{B}_{\text{if}}^{\text{ind}}\), given in Definition 5.8, and is applied when the if-statement is evaluated with the purpose to restrict the register states for the subsequent analysis.

Definition 5.8 (Boolean restriction):

\[
\mathcal{B}_{\text{if}}^{\text{ind}}[b]x = \alpha_{\text{reg}}(\{r \in \gamma_{\text{reg}}(x) \mid \mathcal{B}[b]r\}) \quad \Box
\]
Table 5.2: The abstract function evaluating arithmetic expressions.

\[
\begin{align*}
\bar{A}[n] \bar{r} &= \alpha_{val}(\{n\}) \\
\bar{A}[r] \bar{r} &= \bar{r} \ r \\
\bar{A}[a_1 + a_2] \bar{r} &= \bar{A}[a_1] \bar{r} +_{val} \bar{A}[a_2] \bar{r} \\
\bar{A}[a_1 - a_2] \bar{r} &= \bar{A}[a_1] \bar{r} -_{val} \bar{A}[a_2] \bar{r} \\
\bar{A}[a_1 * a_2] \bar{r} &= \bar{A}[a_1] \bar{r} \cdot_{val} \bar{A}[a_2] \bar{r} \\
\bar{A}[a_1 / a_2] \bar{r} &= \bar{A}[a_1] \bar{r} /_{val} \bar{A}[a_2] \bar{r}
\end{align*}
\]

\(\bar{B}^{ind}\) is safely induced from \(\bar{B}\), using Definition 3.11, so that the concretization of \(\bar{B}^{ind}[b] \bar{T}\), where \(b \in \text{Bexp}\), always contains (at least) the concrete register states, derived from \(\bar{T} \in (\text{Reg} \rightarrow \text{Val})\), for which \(b\) evaluates to \text{true} [35]. For example, evaluating the statement \(if\ b\ \text{goto}\ l\) in some register state, \(\bar{T}_T\), 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 \(\bar{B}^{ind}_\bar{R}[] b] \bar{T} \neq \perp_{reg}\) and the latter when \(\bar{B}^{ind}_\bar{R}[] b] \bar{T} = \perp_{reg}\). Note that both cases can occur for a given pair of \(b\) and \(\bar{T}_T\). Also note that Definition 5.8 is generic and does not necessarily target the interval domain specifically. The over-approximation \(\bar{R}_R\) of \(\bar{B}^{ind}\) for the interval domain will be used in the abstract axiom transition rules (see Table 5.10 on page 102).

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 \(\bar{R}_R\) as defined in Table 5.3 is strictly induced from \(\bar{B}\) (i.e. is as tight as \(\bar{B}^{ind}\)), a safe approximation will rather be mathematically derived.

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

\[
\begin{align*}
\tilde{\mathcal{R}}_{\tilde{\mathcal{F}}}[b] &\bar{1}_{\text{reg}} = \bar{1}_{\text{reg}} \\
\tilde{\mathcal{R}}_{\tilde{\mathcal{F}}}[\text{true}] &\tilde{\beta} = \tilde{\beta} \\
\tilde{\mathcal{R}}_{\tilde{\mathcal{F}}}[\text{false}] &\tilde{\beta} = \bar{1}_{\text{reg}} \\
\tilde{\mathcal{R}}_{\tilde{\mathcal{F}}}[b_1 \& b_2] &\tilde{\beta} = (\tilde{\mathcal{R}}_{\tilde{\mathcal{F}}}[b_1] \tilde{\beta}) \cap_{\text{reg}} (\tilde{\mathcal{R}}_{\tilde{\mathcal{F}}}[b_2] \tilde{\beta}) \\
\tilde{\mathcal{R}}_{\tilde{\mathcal{F}}}[a_1 == a_2] &\tilde{\beta} = (\tilde{\mathcal{A}}_{\tilde{\mathcal{F}}}[a_1] (\tilde{\mathcal{A}}[a_2] \tilde{\beta})) \cap_{\text{reg}} \tilde{\mathcal{A}}_{\tilde{\mathcal{F}}}[a_2] (\tilde{\mathcal{A}}[a_1] \tilde{\beta}) \\
\tilde{\mathcal{R}}_{\tilde{\mathcal{F}}}[a_1 <= a_2] &\tilde{\beta} = (\tilde{\mathcal{A}}_{\tilde{\mathcal{F}}}[a_1] (\tilde{\mathcal{A}}[a_2] \tilde{\beta} \cap_{\text{int}} \alpha_{\text{int}}(\{-\infty\}) \tilde{\beta}) \cap_{\text{reg}} \tilde{\mathcal{A}}_{\tilde{\mathcal{F}}}[a_2] (\tilde{\mathcal{A}}[a_1] \tilde{\beta} \cap_{\text{int}} \alpha_{\text{int}}(\{\infty\}) \tilde{\beta}) \\
\tilde{\mathcal{R}}_{\tilde{\mathcal{F}}}[\text{!true}] &\tilde{\beta} = \bar{1}_{\text{reg}} \\
\tilde{\mathcal{R}}_{\tilde{\mathcal{F}}}[\text{!false}] &\tilde{\beta} = \tilde{\beta} \\
\tilde{\mathcal{R}}_{\tilde{\mathcal{F}}}[\text{!}(b_1 \& b_2)] &\tilde{\beta} = (\tilde{\mathcal{R}}_{\tilde{\mathcal{F}}}[\text{!}b_1] \tilde{\beta}) \sqcup_{\text{reg}} (\tilde{\mathcal{R}}_{\tilde{\mathcal{F}}}[\text{!}b_2] \tilde{\beta}) \\
\tilde{\mathcal{R}}_{\tilde{\mathcal{F}}}[\text{!}a_1 == a_2] &\tilde{\beta} = ((\tilde{\mathcal{A}}_{\tilde{\mathcal{F}}}[a_1] (\tilde{\mathcal{A}}[a_2] \tilde{\beta} \cap_{\text{int}} \alpha_{\text{int}}(\{-\infty\}) \tilde{\beta}) \cap_{\text{reg}} \tilde{\mathcal{A}}_{\tilde{\mathcal{F}}}[a_2] (\tilde{\mathcal{A}}[a_1] \tilde{\beta} \cap_{\text{int}} \alpha_{\text{int}}(\{\infty\}) \tilde{\beta})) \\
&\quad \cap_{\text{int}} \alpha_{\text{int}}(\{1\}) \cap_{\text{int}} \alpha_{\text{int}}(\{\infty\}) \tilde{\beta}) \\
&\quad \sqcup_{\text{reg}} ((\tilde{\mathcal{A}}_{\tilde{\mathcal{F}}}[a_1] (\tilde{\mathcal{A}}[a_2] \tilde{\beta} \cap_{\text{int}} \alpha_{\text{int}}(\{1\}) \cap_{\text{int}} \alpha_{\text{int}}(\{\infty\}) \tilde{\beta}) \cap_{\text{reg}} \tilde{\mathcal{A}}_{\tilde{\mathcal{F}}}[a_2] (\tilde{\mathcal{A}}[a_1] \tilde{\beta} \cap_{\text{int}} \alpha_{\text{int}}(\{-\infty\}) \tilde{\beta}) \\
&\quad \cap_{\text{int}} \alpha_{\text{int}}(\{1\}) \cap_{\text{int}} \alpha_{\text{int}}(\{-\infty\}) \tilde{\beta})) \\
\tilde{\mathcal{R}}_{\tilde{\mathcal{F}}}[\text{!}a_1 <= a_2] &\tilde{\beta} = (\tilde{\mathcal{A}}_{\tilde{\mathcal{F}}}[a_1] (\tilde{\mathcal{A}}[a_2] \tilde{\beta} \cap_{\text{int}} \alpha_{\text{int}}(\{1\}) \cap_{\text{int}} \alpha_{\text{int}}(\{\infty\}) \tilde{\beta}) \cap_{\text{reg}} \tilde{\mathcal{A}}_{\tilde{\mathcal{F}}}[a_2] (\tilde{\mathcal{A}}[a_1] \tilde{\beta} \cap_{\text{int}} \alpha_{\text{int}}(\{-\infty\}) \tilde{\beta}) \\
&\quad \cap_{\text{int}} \alpha_{\text{int}}(\{1\}) \cap_{\text{int}} \alpha_{\text{int}}(\{-\infty\}) \tilde{\beta})) \\
&= \tilde{\mathcal{R}}_{\tilde{\mathcal{F}}}[\text{!true}] \quad \text{(since the restriction is already satisfied)}
\end{align*}
\]
behind this notation is that all the values for \( a \) that must be taken into account are found in the restricting interval \( i_{\mathcal{R}} \). This basically renders the equation \( a = i_{\mathcal{R}} \) 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 \( i_{\mathcal{R}} \). If the resulting value for \( r \) is \( \bot_{\text{int}} \), then the equation \( a = i_{\mathcal{R}} \) 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.
Solving the equation $a = i_{\mathcal{I}}$ 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 $a = a_1 \cdot a_2$ and $a = a_1 / a_2$, special care must be taken as the operators $*_{\text{int}}$ 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 $r_1 / r_2 \leq 1$, where $r_1 = [-100, 100]$ and $r_2 = [4, 6]$, using the $*_{\text{int}}$ 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 $[-\infty, 1] *_{\text{int}} [4, 6] = [-\infty, 6]$. 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 $r_1 / r_2 \leq 1$ since $[11/6] = 1$ which means that $*_{\text{int}}$ 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 $r_1 * r_2 \leq -7$, 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 $[-6, \infty]$. This means that the value of $r_1$ would be restricted by $[-6, \infty] /_{\text{int}} [8, 40] = [-1, \infty]$. Thus, the resulting value of $r_1$ would be $[-1, 100]$. However, the interval $[0, 100]$ is also a safe restricted value for $r_1$ since $-1 \cdot 8 \leq -7$ and $-1 \cdot 40 \leq -7$ (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 $\sim_{\text{int}}$ which is used in the definition of $\mathcal{I}_{\text{int}}$, found in Table 5.4. This operator is derived based on the following equation solution in the infinite equation integer domain, $\mathbb{Z} \cup \{-\infty, \infty\}$, and it safely calculates the restricting interval, $i_{\mathcal{I}}^{a_1} \in \text{Intv}$, for $a_1$ in the expression $a_1 / a_2$ based on a safe restricting interval, $i_{\mathcal{I}} \in \text{Intv}$, for that expression and a known value, $i^{a_2} \in \text{Intv}$, for $a_2$ (i.e. $i_{\mathcal{I}}^{a_1} = i_{\mathcal{I}} \sim_{\text{int}} i^{a_2}$). Note that in Table 5.4, the operator is used on a strictly negative interval (or $\bot_{\text{int}}$), a strictly positive interval (or...
and the $[0,0]$ interval (or $\mathcal{I}_{\text{int}}$) for $i^{a_2}$. 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 $\oplus_{\text{int}}^*$.

$\oplus_{\text{int}}^*$: 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:

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

Table 5.6 defines the operator $\oplus_{\text{int}}^*$ which is used in the definition of $\oplus_{\mathcal{I}}$, 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, $i_{\mathcal{I}}^{a_1} \in \mathbf{Intv}$, for $a_1$ in the expression $a_1 * a_2$ based on a safe restricting interval, $i_{\mathcal{I}} \in \mathbf{Intv}$, for that expression and a known value, $i^{a_2} \in \mathbf{Intv}$, for $a_2$ (i.e. $i_{\mathcal{I}}^{a_1} = i_{\mathcal{I}} \oplus_{\text{int}}^* i^{a_2}$). Remember that if $i_{\mathcal{I}}^{a_1} @ [l,u]$ is such that $u < l$, then $i_{\mathcal{I}}^{a_1} = \mathcal{I}_{\text{int}}$ (Definition 3.31). Note that in Table 5.4, the operator is used on a strictly negative interval (or $\mathcal{I}_{\text{int}}$), a strictly positive interval (or $\mathcal{I}_{\text{int}}$) and the $[0,0]$ interval (or $\mathcal{I}_{\text{int}}$) for $i^{a_2}$. 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: $i_{\mathcal{I}}^{a_2} = i_{\mathcal{I}} \oplus_{\text{int}}^* i^{a_1}$. Further note that any operator fulfilling (i.e. over-approximating) the below result qualifies as a substitute for $\oplus_{\text{int}}^*$. 


Table 5.5: Multiplication operator for determining the restricting interval for the numerator in an interval division expression.

\[ \bot_{\text{int}} \otimes_{\text{int}} i = i \otimes_{\text{int}} \bot_{\text{int}} = \bot_{\text{int}} \quad \text{where } i \in \text{Intv} \]

\[ [l_1, u_1] \otimes_{\text{int}} [l_2, u_2] = \]

\[ \begin{aligned}
\bot_{\text{int}} & \quad \text{if } [l_2, u_2] = [0, 0] \wedge \{-\infty, \infty\} \cap \gamma_{\text{int}}([l_1, u_1]) = \emptyset \\
\bot_{\text{int}} & \quad \text{if } [l_2, u_2] \in \{[-\infty, -\infty], [\infty, \infty]\} \wedge 0 \notin \gamma_{\text{int}}([l_1, u_1]) \\
\min(V_1 \cup V_2 \cup V_3 \cup V_4 \cup V_5 \cup V_6 \cup V_7 \cup V_8 \cup V_9 \cup V_{10}), & \quad \text{otherwise}
\end{aligned} \]

where

\[ V_1 = \{-\infty, \infty\} \quad \text{if } \{-\infty, \infty\} \cap \gamma_{\text{int}}([l_2, u_2]) \neq \emptyset \wedge 0 \notin \gamma_{\text{int}}([l_1, u_1]) \]

\[ V_2 = \{0\} \quad \text{if } 0 \notin \gamma_{\text{int}}([l_1, u_1]) \]

\[ V_3^{\text{min}} = \{l_1 l_2\} \quad \text{if } 0 < l_1 \wedge 0 < l_2 \]

\[ V_3^{\max} = \{u_1 u_2 + u_2 - 1\} \quad \text{if } 0 \leq u_1 \wedge 0 < u_2 < \infty \]

\[ V_4^{\text{min}} = \{l_1 u_2\} \quad \text{if } l_1 < 0 \wedge 0 < u_2 \]

\[ V_4^{\max} = \{u_1 l_2 + l_2 - 1\} \quad \text{if } u_1 < 0 \wedge 0 < l_2 \]

\[ V_5^{\text{min}} = \{u_1 l_2 + l_2 + 1\} \quad \text{if } 0 \leq u_1 \wedge -\infty < l_2 < 0 \]

\[ V_5^{\max} = \{l_1 u_2\} \quad \text{if } 0 < l_1 \wedge u_2 < 0 \]

\[ V_6^{\text{min}} = \{u_1 u_2 + u_2 + 1\} \quad \text{if } u_1 < 0 \wedge u_2 < 0 \]

\[ V_6^{\max} = \{l_1 l_2\} \quad \text{if } l_1 < 0 \wedge l_2 < 0 \]

\[ V_7 = \{\infty\} \quad \text{if } (\infty \in \gamma_{\text{int}}([l_1, u_1]) \wedge 0 < u_2) \vee \\
(\infty \in \gamma_{\text{int}}([l_1, u_1]) \wedge l_2 < 0) \]

\[ V_8 = \{-\infty\} \quad \text{if } (-\infty \in \gamma_{\text{int}}([l_1, u_1]) \wedge 0 < u_2) \vee \\
(\infty \in \gamma_{\text{int}}([l_1, u_1]) \wedge l_2 < 0) \]

\[ V_9 = \{1, \infty\} \quad \text{if } \infty \in \gamma_{\text{int}}([l_1, u_1]) \wedge 0 \in \gamma_{\text{int}}([l_2, u_2]) \]

\[ V_{10} = \{-\infty, -1\} \quad \text{if } -\infty \in \gamma_{\text{int}}([l_1, u_1]) \wedge 0 \in \gamma_{\text{int}}([l_2, u_2]) \]

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.
Chapter 5. Abstractly Interpreting PPL

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

\[
\bot_{\text{int}} \otimes_{\text{int}}^* i = i \otimes_{\text{int}}^* \bot_{\text{int}} = \bot_{\text{int}} \quad \text{where } i \in \text{Intv}
\]

\[
[l_1, u_1] \otimes_{\text{int}}^* [l_2, u_2] =
\begin{cases}
\bot_{\text{int}} & \text{if } 0 \not\in \gamma_{\text{int}}([l_1, u_1]) \land [l_2, u_2] = [0, 0] \\
\bot_{\text{int}} & \text{if } \{-\infty, \infty\} \cap \gamma_{\text{int}}([l_1, u_1]) = \emptyset \land [l_2, u_2] \in \{[-\infty, -\infty], [\infty, \infty]\} \\
\bot_{\text{int}} & \text{if } 0 \not\in \gamma_{\text{int}}([l_1, u_1]) \land \max(\{|l_1|, |u_1|\}) < \min(\{|l_2|, |u_2|\}) \\
[-\infty, \infty] & \text{if } 0 \in \gamma_{\text{int}}([l_1, u_1]) \land 0 \in \gamma_{\text{int}}([l_2, u_2]) \\
\min(V_0 \cup V_{1}^{\text{min}} \cup V_{2}^{\text{min}} \cup V_3 \cup V_4 \cup V_{5}^{\text{min}} \cup V_{6}^{\text{min}} \cup V_{7}^{\text{min}} \cup V_{8}^{\text{min}}, \\
\max(V_0 \cup V_{1}^{\text{max}} \cup V_{2}^{\text{max}} \cup V_3 \cup V_4 \cup V_{5}^{\text{max}} \cup V_{6}^{\text{max}} \cup V_{7}^{\text{max}} \cup V_{8}^{\text{max}})) & \text{otrsw}
\end{cases}
\]

where

\[
\begin{align*}
V_0 &= \{0\} & \text{if } & 0 \in \gamma_{\text{int}}([l_1, u_1]) \land u_2 \neq -\infty \land l_2 \neq \infty \\
V_{1}^{\text{min}} &= \{1\} & \text{if } & l_1 = l_2 = -\infty \lor u_1 = u_2 = \infty \\
V_{1}^{\text{max}} &= \{\infty\} & \text{if } & l_1 = l_2 = -\infty \lor u_1 = u_2 = \infty \\
V_{2}^{\text{min}} &= \{-\infty\} & \text{if } & l_1 = -u_2 = -\infty \lor u_1 = -l_2 = \infty \\
V_{2}^{\text{max}} &= \{-1\} & \text{if } & l_1 = -u_2 = -\infty \lor u_1 = -l_2 = \infty \\
V_3 &= \{-\infty\} & \text{if } & (u_1 = \infty \land l_2 < 0) \lor (l_1 = -\infty \land 0 < u_2) \\
V_4 &= \{\infty\} & \text{if } & (u_1 = \infty \land 0 < u_2) \lor (l_1 = -\infty \land l_2 < 0) \\
V_{5}^{\text{min}} &= \{|l_1/u_2| + (u_2 = \infty \land 1 : 0)\} & \text{if } & 0 < l_1 < \infty \land 0 < u_2 \\
V_{5}^{\text{max}} &= \{|u_1/l_2| - (l_2 = \infty \land 1 : 0)\} & \text{if } & 0 < u_1 < \infty \land 0 < l_2 \\
V_{6}^{\text{min}} &= \{|l_1/l_2| + (l_2 = \infty \land 1 : 0)\} & \text{if } & -\infty < l_1 < 0 \land 0 < l_2 \\
V_{6}^{\text{max}} &= \{|u_1/u_2| - (u_2 = \infty \land 1 : 0)\} & \text{if } & -\infty < u_1 < 0 \land 0 < u_2 \\
V_{7}^{\text{min}} &= \{|u_1/l_2| + (u_2 = -\infty \land 1 : 0)\} & \text{if } & 0 < u_1 < \infty \land u_2 < 0 \\
V_{7}^{\text{max}} &= \{|l_1/l_2| - (l_2 = -\infty \land 1 : 0)\} & \text{if } & 0 < l_1 < \infty \land l_2 < 0 \\
V_{8}^{\text{min}} &= \{|u_1/l_2| + (l_2 = -\infty \land 1 : 0)\} & \text{if } & -\infty < u_1 < 0 \land l_2 < 0 \\
V_{8}^{\text{max}} &= \{|l_1/u_2| - (u_2 = -\infty \land 1 : 0)\} & \text{if } & -\infty < l_1 < 0 \land u_2 < 0
\end{align*}
\]

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

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

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

Table 5.7 defines the operator \( \Box_{\text{int}}^{/} \) which is used in the definition of \( \Box_{\text{int}}^* \), 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, \( i_{\text{int}}^{a_2} \in \text{Intv} \), for \( a_2 \) in the expression \( a_1 / a_2 \) based on a safe restricting interval, \( i_{\text{int}} \in \text{Intv} \), for that expression and a known value, \( i_{\text{int}} \in \text{Intv} \), for \( a_1 \) (i.e. \( i_{\text{int}}^{a_1} = i_{\text{int}} \Box_{\text{int}}^{/} i_{\text{int}}^{a_2} \)). Remember that if \( i_{\text{int}}^{a_2} \in \Box_{\text{int}}^{/} i_{\text{int}}^{a_1} \) is such that \( u < l \), then \( i_{\text{int}}^{a_2} = \bot_{\text{int}} \) (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 \( \Box_{\text{int}}^{/} \).
Table 5.7: Division operator for determining the restricting interval for the denominator in an interval division expression.

\[ \bot_{int} \odot_{int} i = i \odot_{int} \bot_{int} = \bot_{int} \text{ where } i \in \text{Intv} \]

\[ [l_1, u_1] \odot_{int} [l_2, u_2] = \]

\[ \begin{cases} \left[ -\infty, \infty \right] & \text{if } l_1 \neq \infty \land u_1 \neq -\infty \land 0 \notin \gamma_{int}([l_2, u_2]) \\ \bot_{int} & \text{if } \left[ l_1, u_1 \right] = [0, 0] \land 0 \notin \gamma_{int}([l_2, u_2]) \\ \bot_{int} & \text{if } (l_1 = \infty \lor u_1 = -\infty) \land l_2 \neq -\infty \land u_2 \neq \infty \\ \bot_{int} & \text{if } 0 \leq l_1 \land u_1 < -\infty \land u_2 = -\infty \\ \bot_{int} & \text{if } -\infty < l_1 \land u_1 \leq 0 \land l_2 = \infty \\ \bot_{int} & \text{if } 0 \notin \gamma_{int}([l_2, u_2]) \land \max(\{|l_1|, |u_1|\}) < \min(\{|l_2|, |u_2|\}) \land \\
( l_2 = -\infty \Rightarrow 0 \leq l_1 ) \land (u_2 = \infty \Rightarrow u_1 \leq 0 ) \end{cases} \]

\[ \begin{align*} &\{ \min(V_0 \cup V_1 \cup V_2 \cup V_3^{\text{min}} \cup V_4^{\text{min}} \cup V_5^{\text{min}} \cup V_6^{\text{min}} \cup V_7^{\text{min}} \cup V_8^{\text{min}}), \\
&\max(V_0 \cup V_1 \cup V_2 \cup V_3^{\text{max}} \cup V_4^{\text{max}} \cup V_5^{\text{max}} \cup V_6^{\text{max}} \cup V_7^{\text{max}} \cup V_8^{\text{max}}) \} \end{align*} \text{ otrw} \]

where

\[ V_0 = \{ 0 \} \text{ if } (0 < u_1 \land u_2 = \infty) \lor (l_1 < 0 \land l_2 = -\infty) \]

\[ V_1 = \{ 0, \infty \} \text{ if } l_1 = l_2 = -\infty \lor u_1 = u_2 = \infty \]

\[ V_2 = \{ -\infty, -1 \} \text{ if } l_1 = -u_2 = -\infty \lor u_1 = -l_2 = \infty \]

\[ V_3^{\text{min}} = \{ \lfloor l_1/(u_2 + 1) \rfloor + 1 \} \text{ if } 0 < l_1 \land 0 < u_2 \]

\[ V_3^{\text{max}} = \{ \lfloor u_1/l_2 \rfloor \} \text{ if } 0 < u_1 \land 0 < l_2 \]

\[ V_4^{\text{min}} = \{ \lfloor u_1/(u_2 + 1) \rfloor + 1 \} \text{ if } 0 < u_1 \land u_2 < -1 \]

\[ V_4^{\text{max}} = \{ \lfloor l_1/l_2 \rfloor \} \text{ if } 0 < l_1 \land l_2 < -1 \]

\[ V_5^{\text{min}} = \{ -\infty \} \text{ if } 0 < u_1 \land -1 \notin \gamma_{int}([l_2, u_2]) \]

\[ V_5^{\text{max}} = \{ -l_1 \} \text{ if } 0 < l_1 \land -1 \notin \gamma_{int}([l_2, u_2]) \]

\[ V_6^{\text{min}} = \{ \lfloor l_1/l_2 \rfloor \} \text{ if } l_1 < 0 \land 0 < l_2 \]

\[ V_6^{\text{max}} = \{ \lfloor u_1/(u_2 + 1) \rfloor - 1 \} \text{ if } u_1 < 0 \land 0 < u_2 \]

\[ V_7^{\text{min}} = \{ \lfloor u_1/l_2 \rfloor \} \text{ if } u_1 < 0 \land l_2 < -1 \]

\[ V_7^{\text{max}} = \{ \lfloor l_1/(u_2 + 1) \rfloor - 1 \} \text{ if } l_1 < 0 \land u_2 < -1 \]

\[ V_8^{\text{min}} = \{ -u_1 \} \text{ if } u_1 < 0 \land -1 \notin \gamma_{int}([l_2, u_2]) \]

\[ V_8^{\text{max}} = \{ \infty \} \text{ if } l_1 < 0 \land -1 \notin \gamma_{int}([l_2, u_2]) \]

The sets \( V_i \), where \( i \in \{ 0, 1, 2 \} \), and \( V_i^m \), where \( i \in \{ 3, 4, 5, 6, 7, 8 \} \) and \( m \in \{ \text{min}, \text{max} \} \), 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, \( \langle \alpha_{\text{var}}, \gamma_{\text{var}} \rangle \), between the concrete domain \( \mathcal{P}(\text{Var} \to \text{Thrd} \to \mathcal{P}(\text{Val} \times \text{Time})) \) and the abstract domain \( (\text{Var} \to \text{Thrd} \to \mathcal{P}(\text{Val} \times \text{Time})) \cup \{\bot_{\text{var}}, \top_{\text{var}}\} \ni x \) can be established.

\( \otimes \) is the bottom element, \( \bot_{\text{var}} \), if \( \exists x \in \text{Var} : \exists T \in \text{Thrd} : ((\otimes x) T) = \emptyset \); 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 (\( \gamma_{\text{var}}(\bot_{\text{var}}) = \emptyset \)). Therefore, an abstract variable state, \( \otimes \), that actually contains no history for thread \( T \) on variable \( x \), should have \( ((\otimes x) T) = \{((\bot_{\text{val}}, \bot_{\text{t}})\} \) to make \( \otimes \neq \bot_{\text{var}} \). Note that \( \gamma_{\text{var}}(\otimes) \), where

\[ \otimes_{\text{int}}: \text{Solving } a_1 / a_2 = v \iff [a_1 / a_2] = v \iff v \leq a_1 / a_2 < v + 1 \text{ for } a_2 \text{ gives:} \]

\[ a_1 = 0 \land v = 0 \quad \Rightarrow \quad -\infty \leq a_2 < 0 \lor 0 < a_2 \leq \infty \]
\[ a_1 = 0 \land v \neq 0 \quad \Rightarrow \quad \text{no solution} \]
\[ a_1 \in \{-\infty, \infty\} \land v = 0 \quad \Rightarrow \quad \text{no solution} \]
\[ a_1 \in \{-\infty, \infty\} \land v = a_1 \quad \Rightarrow \quad 0 \leq a_2 < \infty \]
\[ a_1 \in \{-\infty, \infty\} \land v = -a_1 \quad \Rightarrow \quad -\infty < a_2 < 0 \]
\[ a_1 \in \{-\infty, \infty\} \land v \notin \{-\infty, 0, \infty\} \quad \Rightarrow \quad \text{no solution} \]
\[ a_1 \notin \{-\infty, 0, \infty\} \land v / a_1 = -\infty \quad \Rightarrow \quad \text{no solution} \]
\[ a_1 \notin \{-\infty, 0, \infty\} \land v / a_1 = \infty \quad \Rightarrow \quad a_2 = 0 \]
\[ 0 < a_1 < \infty \land 0 < v \leq a_1 \quad \Rightarrow \quad 0 < a_1 / (v + 1) < a_2 \leq a_1 / v \]
\[ 0 < a_1 < \infty \land a_1 < v < \infty \quad \Rightarrow \quad \text{no solution} \]
\[ 0 < a_1 < \infty \land v = 0 \quad \Rightarrow \quad a_2 \in \{-\infty, \infty\} \]
\[ 0 < a_1 < \infty \land v = -1 \quad \Rightarrow \quad -\infty < a_2 \leq -a_1 < 0 \]
\[ 0 < a_1 < \infty \land -a_1 \leq v < -1 \quad \Rightarrow \quad a_1 / (v + 1) < a_2 \leq a_1 / v < 0 \]
\[ 0 < a_1 < \infty \land -\infty < v < -a_1 \quad \Rightarrow \quad \text{no solution} \]
\[ -\infty < a_1 < 0 \land 0 < v \leq -a_1 \quad \Rightarrow \quad a_1 / v \leq a_2 < a_1 / (v + 1) < 0 \]
\[ -\infty < a_1 < 0 \land -a_1 < v < \infty \quad \Rightarrow \quad \text{no solution} \]
\[ -\infty < a_1 < 0 \land v = 0 \quad \Rightarrow \quad a_2 \in \{-\infty, \infty\} \]
\[ -\infty < a_1 < 0 \land v = -1 \quad \Rightarrow \quad 0 < -a_1 \leq a_2 < \infty \]
\[ -\infty < a_1 < 0 \land a_1 \leq v < -1 \quad \Rightarrow \quad 0 < a_1 / v \leq a_2 < a_1 / (v + 1) \]
\[ -\infty < a_1 < 0 \land -\infty < v < a_1 \quad \Rightarrow \quad \text{no solution} \]
((\exists x)\; T) = \{(\darr_{\var}, \darr_{\gamma})\} \text{ for some variable, } x, \text{ and thread, } T, \text{ is a set of concrete states, } \mathbb{X}, \text{ for which all } \exists x \in \mathbb{X} \text{ are such that } ((\exists x)\; T) = \emptyset.

The top element, \(\darr_{\var}\), corresponds to a state where all variable and thread combinations are mapped to \(\text{Val} \times \text{Time}\).

The concretization function, \(\gamma_{\var}\), and abstraction function, \(\alpha_{\var}\), are given by Definitions 5.9 and 5.10, respectively.

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

\[
\begin{align*}
\gamma_{\var}(\darr_{\var}) &= \text{Var} \rightarrow \text{Thrd} \rightarrow (\text{Val} \times \text{Time}) \\
\gamma_{\var}(\darr_{\gamma}) &= \emptyset \\
\gamma_{\var}(\darr_x) &= \{\lambda x \in \text{Var}.f \mid f \in \{\lambda T \in \text{Thrd}.W \mid W \in \\
& \quad \{W' \mid (\alpha_{\var}(\{v \in \text{Val} \mid \exists t \in \text{Time} : (v, t) \in W\prime\}), \\
& \quad \alpha_{\var}(\{t \in \text{Time} \mid \exists v \in \text{Val} : (v, t) \in W\prime\})\}) \in \\
& \quad ((\exists x)\; T)\})\}
\end{align*}
\]

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

\[
\begin{align*}
\alpha_{\var}(\text{Var} \rightarrow \text{Thrd} \rightarrow (\text{Val} \times \text{Time})) &= \darr_{\var} \\
\alpha_{\var}(\emptyset) &= \darr_{\gamma} \\
\alpha_{\var}(\darr_x) &= \lambda x \in \text{Var}.\lambda T \in \text{Thrd}. \\
& \quad \{((\alpha_{\var}(\{v \in \text{Val} \mid \exists t \in \text{Time} : (v, t) \in W\}, \\
& \quad \alpha_{\var}(\{t \in \text{Time} \mid \exists v \in \text{Val} : (v, t) \in W\})\}) \mid \\
& \quad W \in ((\exists x)\; T \mid x \in \mathbb{X})\}
\end{align*}
\]

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

**Theorem 5.11 (Galois connection – Variable states):**

\(\langle \alpha_{\var}, \gamma_{\var} \rangle\), where \(\gamma_{\var}\) and \(\alpha_{\var}\) are defined as in Definitions 5.9 and 5.10, respectively, defines a Galois connection.

**Proof.** Since \(\langle \alpha_{\int}, \gamma_{\int} \rangle\) is a Galois insertion (Theorem 3.39) and thus a Galois connection, so are \(\langle \alpha_{\var}, \gamma_{\var} \rangle\) and \(\langle \alpha_{\gamma}, \gamma_{\gamma} \rangle\) (since \(\alpha_{\var} = \alpha_{\gamma} = \alpha_{\int}\) and \(\gamma_{\var} = \gamma_{\gamma} = \gamma_{\int}\)). Using Theorems 3.17, 3.20 and 3.24 to derive \(\alpha_{\var}\) and \(\gamma_{\var}\), the result follows (note that the cases \(\gamma_{\var}(\darr_{\var}), \gamma_{\var}(\darr_{\gamma}), \alpha_{\var}(\text{Var} \rightarrow \text{Thrd} \rightarrow (\text{Val} \times \text{Time}))\) and \(\alpha_{\var}(\emptyset)\) 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:

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

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

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

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

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

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

\[
\begin{align*}
\alpha_{\text{T}}(V') &= \lambda T \in \text{Thrd}. \alpha_{\mathcal{P}}(\{v' T \mid v' \in V'\}) \\
\gamma_{\text{T}}(d) &= \{\lambda T \in \text{Thrd}. W \mid W \in \gamma_{\mathcal{P}}(d T)\}
\end{align*}
\]

and $V' \in \mathcal{P}((\text{Thrd} \rightarrow \mathcal{P}(\text{Val} \times \text{Time})))$ and $d \in \text{Thrd} \rightarrow \mathcal{P}(\tilde{\text{Val}} \times \tilde{\text{Time}})$.

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

\[
\begin{align*}
\alpha_{\text{var}}(\bar{x}) &= \lambda x \in \text{Var}. \alpha_{\text{T}}(\{\bar{x} x \mid x \in \bar{x}\}) \\
\gamma_{\text{var}}(\bar{x}) &= \{\lambda x \in \text{Var}. f \mid f \in \gamma_{\text{T}}(\bar{x} x)\}
\end{align*}
\]

and $\bar{x} \in \mathcal{P}(\text{Var} \rightarrow \text{Thrd} \rightarrow \mathcal{P}(\text{Val} \times \text{Time}))$ and $d \in \text{Var} \rightarrow \text{Thrd} \rightarrow \mathcal{P}(\tilde{\text{Val}} \times \tilde{\text{Time}})$.

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

\[
\begin{align*}
\gamma_{\text{var}}(\bar{x}) &\overset{\text{Th. 3.24}}{=} \{\lambda x \in \text{Var}. f \mid f \in \gamma_{\text{T}}(\bar{x} x)\} \\
&\overset{\text{Th. 3.24}}{=} \{\lambda x \in \text{Var}. f \mid f \in \{\lambda T \in \text{Thrd}. W \mid W \in \gamma_{\mathcal{P}}((\bar{x} x) T)\}\} \\
&\overset{\text{Th. 3.20}}{=} \{\lambda x \in \text{Var}. f \mid f \in \{\lambda T \in \text{Thrd}. W \mid W \in \{W' \mid \alpha_{\omega}(W') \in ((\bar{x} x) T)\}\}\} \\
&\overset{\text{Th. 3.17}}{=} \{\lambda x \in \text{Var}. f \mid f \in \{\lambda T \in \text{Thrd}. W \mid W \in \{W' \mid (\alpha_{\text{val}}(\{v \in \text{Val} \mid \exists t \in \text{Time} : (v, t) \in W'\}), \\
\alpha_t(\{t \in \text{Time} \mid \exists v \in \text{Val} : (v, t) \in W'\})) \in ((\bar{x} x) T)\}\}\}
\end{align*}
\]
\( \alpha_{\text{var}}(X) \) 
\[ \begin{align*}
\text{Th. 3.24} & : \lambda x \in \text{Var} \Rightarrow \alpha_T(\{x \mid x \in X\}) \\
\text{Th. 3.24} & : \lambda x \in \text{Var} \Rightarrow \alpha_T(\{f \mid f \in \{x \mid x \in X\}\})
\end{align*} \]
\[ \begin{align*}
\text{calc.} & : \lambda x \in \text{Var} \Rightarrow \alpha_T(\{(x) \mid x \in X\}) \\
\text{Th. 3.20} & : \lambda x \in \text{Var} \Rightarrow \alpha_T(\{(\alpha) \mid \alpha \in X\}) \\
\text{Th. 3.17} & : \lambda x \in \text{Var} \Rightarrow \alpha_T(\{(\alpha) \mid \alpha \in X\}) \\
\end{align*} \]

The state \( \tilde{x} \in (\text{Var} \rightarrow \text{Thrd} \rightarrow \mathcal{P}(\tilde{\text{Val}} \times \tilde{\text{Time}})) \cup \{\tilde{\text{var}}, \tilde{\text{var}}\} \) can save any number (i.e. history) of abstract writes, \( \tilde{w} \in \tilde{\text{Val}} \times \tilde{\text{Time}} \), 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 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, \( \sqsubseteq_w \), and least upper bound operator, \( \sqcup_w \), for writes follow naturally (cf. Definitions 3.26 and 3.28) from the partial orders and least upper bound operators for values, \( \sqsubseteq_{\text{val}} \) and \( \sqcup_{\text{val}} \), and time, \( \sqsubseteq_t \) and \( \sqcup_t \). \( \sqsubseteq_w \) and \( \sqcup_w \) are given by Definitions 5.12 and 5.13, respectively.

**Note.** The notations “\( \tilde{w} \)” and “\((\tilde{v}, \tilde{t})\)” for abstract writes will from here on be used interchangeably.

**Definition 5.12 (Partial order of writes):**
\( \begin{align*}
\tilde{w} & \sqsubseteq_w \tilde{v} \\
\tilde{w} & \sqsubseteq_w \tilde{v} \\
(\tilde{v}_1, \tilde{t}_1) & \sqsubseteq_w (\tilde{v}_2, \tilde{t}_2) \iff \tilde{v}_1 \sqsubseteq_{\text{val}} \tilde{v}_2 \wedge \tilde{t}_1 \sqsubseteq_t \tilde{t}_2
\end{align*} \)

**Definition 5.13 (Least upper bound of writes):**
\( \begin{align*}
\tilde{w} \sqcap_w \tilde{v} & = \tilde{t}_w \sqcap_w \tilde{v} = \tilde{t}_w \\
\tilde{w} \sqcup_w \tilde{v} & = \tilde{t}_w \sqcup_w \tilde{v} = \tilde{t}_w \\
(\tilde{v}_1, \tilde{t}_1) \sqcup_w (\tilde{v}_2, \tilde{t}_2) & = (\tilde{v}_1 \sqcup_{\text{val}} \tilde{v}_2, \tilde{t}_1 \sqcup_t \tilde{t}_2)
\end{align*} \)
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 (Time precedence):**

\[
\begin{align*}
\hat{t} &\preceq_t \hat{t}' & \text{if } \hat{t} \neq \hat{t}' \\
\mathbb{I}_t &\preceq_t \hat{t} & \text{if } \hat{t} \neq \mathbb{I}_t \\
\hat{t}_1 &\preceq_t \hat{t}_2 \iff \max(\gamma_t(\hat{t}_1)) < \min(\gamma_t(\hat{t}_2)) & \text{if } \hat{t}_1, \hat{t}_2 \notin \{\mathbb{I}_t, \hat{t}\}
\end{align*}
\]

The definitions of the partial order relation, \( \sqsubseteq_{\text{var}} \), 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. 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 (see Definition 5.20; see Figure 5.8 on page 88 and its accompanying explanation if Definition 5.20 is not found completely clear) at all times. Note that Definition 5.18 defines the uniquely most recent write in a set of writes. This definition defines the most recent write both among several threads (globally) and for single threads (locally). These definitions 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 105).

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

\[
\begin{align*}
\overline{x} &\sqsubseteq_{\text{var}} \overline{\overline{x}} & \overline{\overline{x}} &\sqcap_{\text{var}} \overline{x} \\
\overline{x} &\sqsubseteq_{\text{var}} \overline{x} & \overline{\overline{x}} \sqcup_{\text{var}} \overline{x} \\
\overline{x} &\sqsubseteq_{\text{var}} \overline{x'} \iff \forall x \in \mathbf{Var} : \forall T \in \mathbf{Thrd} : (\overline{x} x) T \subseteq (\overline{x'} x) T
\end{align*}
\]
Definition 5.16 (Greatest lower bound of abstract variable states):

\[
\begin{align*}
\tilde{T}_{\text{var}} \cap_{\text{var}} \tilde{x} &= \tilde{x} \cap_{\text{var}} \tilde{T}_{\text{var}} = \tilde{x} \\
\tilde{I}_{\text{var}} \cap_{\text{var}} \tilde{x} &= \tilde{x} \cap_{\text{var}} \tilde{I}_{\text{var}} = \tilde{x} \\
((\tilde{x} \cap_{\text{var}} \tilde{x}') \times T) &= ((\tilde{x} \times T) \cap ((\tilde{x}' \times T))
\end{align*}
\]

\[\square\]

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

\[
\begin{align*}
\tilde{T}_{\text{var}} \cup_{\text{var}} \tilde{x} &= \tilde{x} \cup_{\text{var}} \tilde{T}_{\text{var}} = \tilde{T}_{\text{var}} \\
\tilde{I}_{\text{var}} \cup_{\text{var}} \tilde{x} &= \tilde{x} \cup_{\text{var}} \tilde{I}_{\text{var}} = \tilde{x} \\
((\tilde{x} \cup_{\text{var}} \tilde{x}') \times T) &= ((\tilde{x} \times T) \cup ((\tilde{x}' \times T))
\end{align*}
\]

\[\square\]

Definition 5.18 (Time of most recent write):

The most recent write(s), \((\tilde{v}, \tilde{t})\), in a set of 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, \((\tilde{v}', \tilde{t}')\), are such that \(\min(\gamma_t(\tilde{t})) = \min(\gamma_{t'}(\tilde{t}'))\), the time of the most recent write, \(\tilde{t}\), is uniquely determined from the write(s) \(\max(\gamma_t(\tilde{t})) = \max(\{\max(\gamma_{t'}(\tilde{t}')) | \tilde{t}' \text{ ranges over the timestamps of the writes such that } \min(\gamma_t(\tilde{t})) = \min(\gamma_{t'}(\tilde{t}'))\})\). \(\square\)

Definition 5.19 (Safe write history):

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

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

1. all writes, \((\tilde{v}, \tilde{t}')\), by \(T\) on \(x\), such that \(\tilde{t}' \preceq \tilde{t} \wedge \tilde{t} \preceq \tilde{t}'\), and

2. the latest (most recent) write(s), \((\tilde{v}, \tilde{t}')\), by \(T\) on \(x\), such that \(\tilde{t}' \preceq \tilde{t}\), if \(\tilde{t}' \cap t^{-\text{mnrw}} \neq \underline{I}_t\), where \(t^{-\text{mnrw}}\) is the time of the globally most recent write, such that \(t^{-\text{mnrw}} \preceq \tilde{t}\),

or,

3. \((\underline{I}_{\text{val}}, \underline{I}_t)\), 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 \(\tilde{x}\) is a safe approximation of \(x\) (i.e. \(\tilde{x}\) contains safe write history) iff \(\exists \tilde{x}' \in \gamma_{\text{var}}(\tilde{x}) : \forall x \in \text{Var} : \forall T \in \text{Thrd} : ((\tilde{x} \times T) \subseteq ((\tilde{x}' \times T))\). \(\square\)
Definition 5.20 (Safe value of \( x \) as seen by thread \( T \)):
Assuming that \( \hat{\gamma} \) contains 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 time \( \hat{t} \) is the least upper bound, \( \bigcup_{\text{val}} \), 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, \( \hat{w}_{T'} = (\hat{v}_{T'}', \hat{t}_{T'}) \), for thread \( T' \in \text{Thrd} \setminus \{T\} \) on \( x \) such that \( \hat{t}_{T'} \not< \hat{t} \land \hat{t} \not< \hat{t}_{T'} \).

2. The most recent write(s) in \( \{ (\hat{v}_{T}, \hat{t}_{T}) \in (\hat{\gamma} x) T' \mid \hat{t}_{T'} \not< \hat{t} \land \hat{t} \not< \hat{t}_{T'} \cap \hat{t} \not\in \hat{\tau}_{T'} \} \) for each thread \( T' \in \text{Thrd} \setminus \{T\} \), and the most recent write(s), \((\hat{v}_{T}, \hat{t}_{T}) \in (\hat{\gamma} x) T \), such that \( \min(\gamma_t(\hat{t}_{T})) \leq \min(\gamma_t(\hat{t})) \), if \( \hat{t} \cap \hat{t}_{T} \cap \hat{\tau}_{T'} \neq \perp_{T} \), where \( \hat{\tau}_{T'} \) is the time of the (globally) most recent write in \( \{ (\hat{v}_{T}, \hat{t}_{T}) \in (\hat{\gamma} x) T \mid \min(\gamma_t(\hat{t}_{T})) \leq \min(\gamma_t(\hat{t})) \} \cup \bigcup_{T' \in \text{Thrd} \setminus \{T\}} \{ (\hat{v}_{T'}, \hat{t}_{T'}) \in (\hat{\gamma} x) T' \mid \hat{t}_{T'} \not< \hat{t} \} \).

The partial order for abstract variable states to be used within the analysis, \( \preceq_{\text{var}} \), is given by Definition 5.21 based on PARTIALORDERVAR, defined in Algorithm 5.1, taking the safety of write history (Definition 5.19) into account. Note that EARIESTWRITETHREAD, as defined in Algorithm 5.2, returns a deterministically defined write. 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):

\[
\begin{align*}
\hat{\gamma} \preceq_{\text{var}} \hat{\gamma}' \iff & \text{PARTIALORDERVAR}(\hat{\gamma}, \hat{\gamma}') \\
\hat{\gamma} \preceq_{\text{var}} \hat{\gamma}' & \iff \exists x \in \text{Var} \forall \hat{t} \in \hat{\tau}_{\text{var}} \hat{\gamma}(\hat{t}) \preceq_{\text{var}} \hat{\gamma}'(\hat{t}) \end{align*}
\]

Based on this partial order relation, the lower bound and upper bound operators to be used within the analysis, \( \bigcap_{\text{var}} \) and \( \bigcup_{\text{var}} \), are given by Definitions 5.22 and 5.23, respectively. Note that MINVAR is defined in Algorithm 5.3 and JOINVAR is defined in Algorithm 5.4. Also note that the notation \(((\hat{\gamma} x) T) \leftarrow \ldots\) will be used as a shorthand for \( \hat{\gamma} \leftarrow \hat{\gamma}[x \mapsto (\hat{\gamma} x)[T \mapsto \ldots]] \) to
Algorithm 5.1 Partial order of abstract variable states

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

1: function EARLIEST_WRITE_THREAD(\( \tilde{W} \))
2:    if \( \tilde{W} = \emptyset \) then
3:        return \( \bot \)
4:    end if
5:    \( \tilde{t}_{\text{min}} \leftarrow \alpha_t(\{\infty\}) \)
6:    for all \( (\tilde{v}, \tilde{t}) \in \tilde{W} \) do
7:        if \( \min(\gamma_t(\tilde{t})) < \min(\gamma_t(\tilde{t}_{\text{min}})) \) then
8:            \( \tilde{t}_{\text{min}} \leftarrow \tilde{t} \)
9:        else if \( \min(\gamma_t(\tilde{t})) = \min(\gamma_t(\tilde{t}_{\text{min}})) \) then
10:            \( \tilde{t}_{\text{min}} \leftarrow \tilde{t} \sqcap_t \tilde{t}_{\text{min}} \)
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:            \( \tilde{v}_{\text{min}} \leftarrow \tilde{v} \sqcap_{\text{val}} \tilde{v}_{\text{min}} \)
20:        end if
21:    end for
22:    return \( (\tilde{v}_{\text{min}}, \tilde{t}_{\text{min}}) \)
23: end function
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):**

\[
\begin{align*}
\{ \hat{\top}_\text{var} \cap \hat{\top}_\text{var} \} \hat{x} &= \hat{x} \\
\{ \hat{\bot}_\text{var} \cap \hat{\bot}_\text{var} \} \hat{x} &= \hat{x} \\
\{ \hat{\top}_\text{var} \cap \hat{\bot}_\text{var} \} \hat{x} &= \hat{x} \\
\hat{x} \cap \hat{\top}_\text{var} \hat{x} &= \text{MEETVAR}(\hat{x}, \hat{x}')
\end{align*}
\]

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

\[
\begin{align*}
\{ \hat{\top}_\text{var} \cup \hat{\top}_\text{var} \} \hat{x} &= \hat{x} \\
\{ \hat{\bot}_\text{var} \cup \hat{\bot}_\text{var} \} \hat{x} &= \hat{x} \\
\{ \hat{\top}_\text{var} \cup \hat{\bot}_\text{var} \} \hat{x} &= \hat{x} \\
\hat{x} \cup \hat{\top}_\text{var} \hat{x}' &= \text{JOINVAR}(\hat{x}, \hat{x}')
\end{align*}
\]

**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 [37] is introduced to lower the complexity of the analysis, at least \(\cup'_\text{var}\) will be needed.

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

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

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

**Proof.** Since \(\text{WRITE}(T, \hat{x}, x, (\bar{v}, \bar{t}))\) simply adds the write \((\bar{v}, \bar{t})\) to the history of thread \(T\)'s writes on variable \(x\) in the state \(\hat{x}\), and \(\hat{x}\) is assumed to contain safe write history for \(T\) on \(x\), \(\text{WRITE}(T, \hat{x}, x, (\bar{v}, \bar{t}))\) trivially fulfills the safety condition in Definition 5.19 with regards to \(T\) and \(x\).

Using the sequence and timing information provided by Definition 5.20, \(\text{READ}(\hat{x}, x, T, \bar{t})\), as defined in Algorithm 5.6, only takes the writes that might be valid at \(\bar{t}\) (the point in time when \(T\) issues the \(\text{READ}\) on \(x\) given the variable state \(\hat{x}\)) into consideration for its returned value, \(\bar{v} \in \text{Val}\), which is safe.
Algorithm 5.3 Meeting two abstract variable states

1: function MEETVAR($\tilde{x}, \tilde{x}'$)
2: $\tilde{x}'' \leftarrow \top_{\text{val}}$
3: for all $x \in \text{Var}$ do
4: for all $T \in \text{Thrd}$ do
5: $W \leftarrow (\tilde{x} \times) T$
6: $W' \leftarrow (\tilde{x}' \times) T$
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 $\tilde{v} \cap_{\text{val}} \tilde{v}' \neq \bot_{\text{val}} \land \tilde{t} \cap_{\text{t}} \tilde{t}' \neq \bot_{\text{t}}$ then
16: $C \leftarrow C \cup \{(\tilde{v} \cap_{\text{val}} \tilde{v}', \tilde{t} \cap_{\text{t}} \tilde{t}')\}$
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: $(\tilde{x}'' \times) T \leftarrow \{\top_{\text{val}}, \top_{\text{t}}\}$
26: else
27: $(\tilde{x}'' \times) T \leftarrow C$
28: end if
29: end for
30: end for
31: return $\tilde{x}''$
32: end function
Algorithm 5.4 Joining two abstract variable states

1: function JOINVAR(\(\tilde{x}, \tilde{x}'\))
2: \(\tilde{x}'' \leftarrow \tilde{I}_{\text{var}}\)
3: for all \(x \in \text{Var}\) do
4: for all \(T \in \text{Thrd}\) do
5: \(\tilde{W} \leftarrow (\tilde{x})_T\)
6: \(\tilde{W}' \leftarrow (\tilde{x}')_T\)
7: \(C \leftarrow \emptyset\)
8: \(M \leftarrow (\tilde{I}_{\text{val}}, \tilde{I}_T)\)
9: while \(\tilde{W} \neq \emptyset \lor \tilde{W}' \neq \emptyset\) do
10: \(\tilde{w} \leftarrow \text{EARLIESTWRITETHREAD}(\tilde{W})\)
11: \(\tilde{w}' \leftarrow \text{EARLIESTWRITETHREAD}(\tilde{W}')\)
12: if \(\tilde{w} = \tilde{w}'\) then
13: \(C \leftarrow C \cup \{\tilde{w}\}\)
14: \(\tilde{W} \leftarrow \tilde{W} \setminus \{\tilde{w}\}\)
15: \(\tilde{W}' \leftarrow \tilde{W}' \setminus \{\tilde{w}'\}\)
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: \(M \leftarrow (\bigcup_w \tilde{W}) \cap_w (\bigcup_w \tilde{W}')\)
24: \(\tilde{W} \leftarrow \emptyset\)
25: \(\tilde{W}' \leftarrow \emptyset\)
26: end if
27: end while
28: \(((\tilde{x}'')_T) \leftarrow C\)
29: if \(M \neq (\tilde{I}_{\text{val}}, \tilde{I}_T)\) then
30: \(((\tilde{x}'')_T) \leftarrow ((\tilde{x}'')_T) \cup \{M\}\)
31: end if
32: end for
33: end for
34: return \(\tilde{x}''\)
35: end function
Algorithm 5.5 Write to variable

1: function WRITE(T, \(\overline{x}, x, \overline{w}\))
2:   for all \(x' \in \textbf{Var}\) do
3:     for all \(T' \in \textbf{Thrd}\) do
4:       \((\overline{x}' \leftarrow x') \mid T' \leftarrow \begin{cases} ((\overline{x}, x) \mid T) \cup \{\overline{w}\} & \text{if } x = x' \land T' = T \\ (\overline{x}' \leftarrow x') \mid T' & \text{otherwise} \end{cases}\)
5:     end for
6:   end for
7: return \(\overline{x}\)
8: end function

(Lemma 5.27). These writes, \(\overline{w} = (\overline{v}', \overline{t}')\), come from two categories, as specified in Definition 5.20: the first category covers the writes on \(x\) for threads \(T' \in \textbf{Thrd} \setminus \{T\}\) whose timestamps overlap in time with \(\overline{t}\), i.e. \(\overline{t} \cap t' \neq \emptyset\); 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 \(\overline{t}\) is discarded. So is any write for \(T' \in \textbf{Thrd} \setminus \{T\}\) such that its timestamp completely succeeds \(\overline{t}\). This is because such writes can simply not have occurred at the time of issuing \(\textbf{READ}\) (and will thus usually not be included in \(\overline{x}\) at all). Note that MOSTRECENTWRITEVALUE and MOSTRECENTWRITETIMETHREAD are defined based on Definition 5.18 in Algorithms 5.7 and 5.8, respectively, and that these functions give the 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 time of the most recent write in \(W\)).

Proof. This proof will be conducted based on the structure of Algorithm 5.8.

If \(\overline{W} = \emptyset\), then \(\overline{t}\) is returned. Otherwise, \(t_{\text{min}}\) is the greatest lower limit of the timestamp of any write in \(\overline{W}\) (\(\min(\{ \gamma_i(t) \mid \exists \overline{v} \in \textbf{Vál} : (\overline{v}, \overline{t}) \in \overline{W} \})\)) and \(t_{\text{max}}\) is the greatest upper limit of the timestamps of the writes in \(\overline{W}\) such that the lower limit of their timestamps are equal to \(t_{\text{min}}\) (\(\max(\{ \gamma_i(t) \mid \exists \overline{v} \in \textbf{Vál} : (\overline{v}, \overline{t}) \in \overline{W} \})\)). Thus, \(\alpha(\{t_{\text{min}}, t_{\text{max}}\})\) is the time of the most recent write in \(\overline{W}\), as given by Definition 5.18.

Lemma 5.26 (Soundness of MOSTRECENTWRITEVALUE):
MOSTRECENTWRITEVALUE(\(\overline{x}, x\), defined in Algorithm 5.7, gives the time of
Algorithm 5.6 Read from variable

1: function READ(\(\tilde{x}, x, T, \tilde{t}\))
2: \(\tilde{x}' \leftarrow \tilde{1}_{\text{var}}\)
3: for all \(T' \in \text{Thrd} \setminus \{T\}\) do
4: \((\tilde{x}' x) T') \leftarrow \{(\tilde{v}', t') \in ((\tilde{x} x) T') \mid \tilde{t} \not\in t'\}\)
5: end for
6: \((\tilde{x}' x) T) \leftarrow \{(\tilde{v}', t') \in ((\tilde{x} x) T) \mid \min(\gamma_t(t)) \geq \min(\gamma_t(t'))\}\)
7: \(\tilde{W} \leftarrow \emptyset\)
8: for all \(T' \in \text{Thrd} \setminus \{T\}\) do
9: \(\tilde{W}_T \leftarrow \{(\tilde{v}', t') \in ((\tilde{x}' x) T') \mid \tilde{t} \not\in t'\}\)
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: \(\tilde{p}_{\text{mrw}} \leftarrow \text{MOSTRECENTWRITE}(\tilde{x}', x)\)
14: if \(\tilde{p}_{\text{mrw}} \neq \tilde{1}_t\) then
15: for all \(T' \in \text{Thrd}\) do
16: \(\tilde{p}_{\text{mrw}}' \leftarrow \text{MOSTRECENTWRITE}(\tilde{x}' x, T')\)
17: \(\tilde{W} \leftarrow \tilde{W} \cup \{\tilde{v}', t' \in \text{Thrd} : (\tilde{v}', t') \in \tilde{W}\} \) if \(\tilde{W} \neq \emptyset\)
18: end for
19: end if
20: \(\tilde{v} \leftarrow \begin{cases} \bigcup_{t \in T} \{\tilde{v}' : \exists t' \in \text{Thrd} : (\tilde{v}', t') \in \tilde{W}\} & \text{if } \tilde{W} \neq \emptyset \vspace{2pt} \text{otr}w \\ [\infty, \infty] & \text{otherwise} \end{cases}\)
21: return \(\tilde{v}\)
22: end function

Algorithm 5.7 Time of most recent write

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

Algorithm 5.8 Time of most recent write in thread

1: function \(\text{MOSTRECENTWRITE}(\tilde{W})\)
2: if \(\tilde{W} = \emptyset\) then
3: return \(\tilde{1}_t\)
4: end if
5: \(t_{\text{min}} \leftarrow \max \{\min(\gamma_t(t)) : \exists \tilde{v} \in \text{V\tilde{a}l} : (\tilde{v}, \tilde{t}) \in \tilde{W}\}\)
6: \(t_{\text{max}} \leftarrow \max \{\bigcup \{\min(\gamma_t(t)) : \exists \tilde{v} \in \text{V\tilde{a}l} : (\tilde{v}, \tilde{t}) \in \tilde{W} \land \min(\gamma_t(t)) = t_{\text{min}}\}\}
7: return \(\alpha_t\{t_{\text{min}}, t_{\text{max}}\}\)
8: end function
the globally most recent write on \( x \) in \( \hat{\mathcal{X}} \).

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]
88 Chapter 5. Abstractly Interpreting PPL

5.8. The returned value, \( \bar{v} \), 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, READ\( (\bar{x}, x, T_1, i_1) \). The writes that fall into category 1 of Definition 5.20, presented on page 79, are \( \bar{w}^2_2 \) and \( \bar{w}^2_3 \). The writes that fall into category 2 of Definition 5.20 are \( \bar{w}^1_1 \) and \( \bar{w}^2_1 \) (note that \( \bar{t}^{\text{mrw}} \) is the timestamp of \( \bar{w}^1_1 \)).

Next consider the second read operation, READ\( (\bar{x}, x, T_2, i_2) \). The writes that fall into category 1 of Definition 5.20 are \( \bar{w}^1_2 \), \( \bar{w}^1_4 \) and \( \bar{w}^2_3 \). The writes that fall into category 2 of Definition 5.20 are \( \bar{w}^1_2 \) and \( \bar{w}^2_3 \) (note that \( \bar{t}^{\text{mrw}} \) is the timestamp of \( \bar{w}^2_3 \)).

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 READ\( (\bar{x}, x, T, i) \) discards writes from thread \( T' \in \text{Thrd} \) that are too old to be valid at time \( i \) 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 105), the discarded writes can safely be removed from \( (\bar{x}, x, T') \). TRIM, defined in Algorithm 5.9, safely (Lemma 5.28) removes the outdated writes from \( (\bar{x}, x, T') \) for all \( T' \in \text{Thrd} \). Thus, TRIM can be used to lower the space complexity of the analysis. Note that SPLITSET\( (\bar{W}, \bar{i}) \), as defined in Algorithm 5.10, is used to split a set of writes into two sets where the first set contains all writes, \( (\bar{v}, \bar{i}) \), such that \( \bar{t}' \cap \bar{t} = \bar{t}' \neq \bar{t} \), and the second set contains all other writes.

Lemma 5.28 (Soundness of TRIM):
If \( \bar{x} \) contains safe write history at time \( i \) (cf. Definition 5.19), then so does...
writes that fall into category 2 of Definition 5.20 are removed writes (note that these values are not shown in the figure).

5.8. The returned value, \( \vartriangledown \), removes the outdated writes from the read operation. It is however not possible to determine in which order the writes included in category 2 have timestamps such that they definitely precede \( \vartriangledown \), as defined in Algorithm 5.10, is used to split a set of writes.

Next consider the second read operation, \( \text{READ}(\vartriangledown) \), as defined in Algorithm 5.10, is used to split a set of writes.

Algorithm 5.9 Trim variable state

1: \textbf{function} \( \text{TRIM}(\bar{\bar{x}}, \bar{t}) \)
2: \( \bar{\bar{x}}' \leftarrow \bar{\bar{1}}_{\text{var}} \)
3: \( \bar{\bar{x}}'' \leftarrow \bar{\bar{1}}_{\text{var}} \)
4: \textbf{for all} \( x \in \text{Var} \) \textbf{do}
5: \( \langle [F]_{T \in \text{Thrd}} \rangle \leftarrow \langle [0]_{T \in \text{Thrd}} \rangle \)
6: \( \langle [O]_{T \in \text{Thrd}} \rangle \leftarrow \langle [0]_{T \in \text{Thrd}} \rangle \)
7: \( \langle [N]_{T \in \text{Thrd}} \rangle \leftarrow \langle [0]_{T \in \text{Thrd}} \rangle \)
8: \textbf{for all} \( T \in \text{Thrd} \) \textbf{do}
9: \( F_T \leftarrow \{ (\bar{v}, \bar{t}') \in (\bar{\bar{x}} x) T | \bar{t} \leq \bar{t}' \} \)
10: \( (O_T, N_T) \leftarrow \text{SPLITSET}((\bar{\bar{x}} x) T, \bar{t}) \)
11: \( ((\bar{\bar{x}}' x) T) \leftarrow N_T \setminus F_T \)
12: \textbf{end for}
13: \( \vartriangledown \text{mrw} \leftarrow \text{MOSTRECENTWRITE}(\bar{\bar{x}}', x) \)
14: \textbf{for all} \( T \in \text{Thrd} \) \textbf{do}
15: \( \bar{W}_T \leftarrow \emptyset \)
16: \( \vartriangledown \text{mrw}_T \leftarrow \text{MOSTRECENTWRITE}(\bar{\bar{x}}' x) T \)
17: \textbf{if} \( \vartriangledown \text{mrw}_T \cap \vartriangledown \text{mrw} = \bar{1}_T \land F_T = \emptyset \land O_T = \emptyset \) \textbf{then}
18: \( \bar{W}_T \leftarrow \{ (\bar{1}_{\text{val}}, \bar{1}_T) \} \)
19: \textbf{else}
20: \( \bar{W}_T \leftarrow \{ (\bar{v}, \bar{t}') \in (\bar{\bar{x}} x) T | \bar{t}' \cap \vartriangledown \text{mrw} \neq \bar{1}_T \land \vartriangledown \text{mrw} \cap \bar{t}' \vartriangledown \text{mrw} \neq \bar{1}_T \} \)
21: \textbf{end if}
22: \( ((\bar{\bar{x}}'' x) T) \leftarrow F_T \cup O_T \cup \bar{W}_T \)
23: \textbf{end for}
24: \textbf{end for}
25: \textbf{return} \( \bar{\bar{x}}'' \)
26: \textbf{end function}

Algorithm 5.10 Split set of writes

1: \textbf{function} \( \text{SPLITSET}(\bar{W}, \bar{t}) \)
2: \( O \leftarrow \{ (\bar{v}, \bar{t}') \in \bar{W} | \bar{t} \not\preceq \bar{t}' \cap \bar{t}' \not\preceq \bar{t} \} \)
3: \( N \leftarrow \{ (\bar{v}, \bar{t}') \in \bar{W} | \bar{t} \preceq \bar{t}' \lor \bar{t}' \preceq \bar{t} \} \)
4: \textbf{return} \( (O, N) \)
5: \textbf{end function}
TRIM(\(\tilde{x}, \tilde{t}\)).

\[\]

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

1. all writes, \((\tilde{v}, \tilde{t}')\), of \((\tilde{x} x) T\) such that \(\tilde{t}' \not\leq t, \tilde{t} \land \tilde{t} \not\leq \tilde{t}'\)

2. any write, \((\tilde{v}, \tilde{t}')\), of \((\tilde{x} x) T\) such that \(\tilde{t}' \not\leq \tilde{t}\), if \(\tilde{t}' \land \tilde{tmrw} \not\leq \tilde{t}\), where \(\tilde{tmrw}\) is the time of the globally most recent write of the writes preceding \(\tilde{t}\),

or,

3. \((\tilde{val}, \tilde{t})\), 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 \((\tilde{x} x) T = \{(\tilde{val}, \tilde{t})\}\)).

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

\[
\tilde{t}' \not\leq t, \tilde{t} \land \tilde{t} \not\leq \tilde{t}' \iff \max(\gamma_{t}(\tilde{t}')) \leq \min(\gamma_{t}(\tilde{t})) \land \max(\gamma_{t}(\tilde{t})) \leq \min(\gamma_{t}(\tilde{t}'))
\]

\[
\iff \max(\gamma_{t}(\tilde{t}')) \geq \min(\gamma_{t}(\tilde{t})) \land \max(\gamma_{t}(\tilde{t})) \geq \min(\gamma_{t}(\tilde{t}'))
\]

\[
\iff \min(\{\max(\gamma_{t}(\tilde{t})), \min(\gamma_{t}(\tilde{t}'))\}) \geq \max(\{\min(\gamma_{t}(\tilde{t})), \min(\gamma_{t}(\tilde{t}'))\})
\]

\[
\text{Def. 3.34} \iff \tilde{t} \land \tilde{t} \not\leq \tilde{t}
\]

Now, assume that \(\tilde{x}\) contains 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 \(\tilde{t} \leq \tilde{t}';\) i.e. writes that occur after \(\tilde{t}\). Note that this captures all writes, \((\tilde{v}, \tilde{t}')\), such that \(\tilde{t}' = \tilde{t}\) as long as \(\tilde{t} \neq \tilde{t}\).

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

- For each thread, \(T \in \text{Thrd}\), the set \(N_{T}\) contains the writes, \((\tilde{v}, \tilde{t}')\), by \(T\) on \(x\) such that \(\tilde{t}' \leq \tilde{t} \land \tilde{t} \not\leq \tilde{t}'\). Note that this captures all writes, \((\tilde{v}, \tilde{t}')\), such that \(\tilde{t}' = \tilde{t}\) or \(\tilde{t}' = \tilde{t}\).
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
\(\mathcal{P}(\text{Lck} \rightarrow (\text{Lck}_{\text{sst}} \times \text{Thrd}_{\perp} \times \text{Time} \times \text{Thrd}_{\perp} \times \text{Time})) \cup \{\mathcal{L}_{\text{lock}}, \mathcal{I}_{\text{lock}}\} \ni \mathbb{I}_{\text{lock}}\),
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):**

\[
\gamma_{\text{lock}}(\mathbb{I}_{\text{lock}}) = \text{Lck} \rightarrow (\text{Lck}_{\text{sst}} \times \text{Thrd}_{\perp} \times \text{Time} \times \text{Thrd}_{\perp} \times \text{Time})
\]
\[
\gamma_{\text{lock}}(\mathbb{I}_{\text{lock}}) = \emptyset
\]
\[
\gamma_{\text{lock}}(\mathbb{I}_{\text{lock}}) = \gamma_{\text{lock}}(\lambda \text{Lck} \in \text{Lck}. (u_{\text{lck}, T_{\text{lck}}, \mathbb{I}_{\text{lck}}, T'_{\text{lck}}, \mathbb{I}'_{\text{lck}}}))
\]
\[
= \{ \lambda \text{Lck} \in \text{Lck}. (u_{\text{lck}, T_{\text{lck}}, \mathbb{I}_{\text{lck}}, T'_{\text{lck}}, \mathbb{I}'_{\text{lck}}}) | u_{\text{lck}} \in \gamma_{l}(\mathbb{I}_{\text{lck}}) \land \mathbb{I}'_{\text{lck}} \in \gamma_{l}(\mathbb{I}'_{\text{lck}}) \}
\]

**Definition 5.30 (Abstraction of a set of lock states):**

\[
\alpha_{\text{lock}}(\mathbb{I}) = \mathcal{G}_{\text{lock}}\{\mathbb{I} | \mathbb{I} \subseteq \gamma_{\text{lock}}(\mathbb{I})\}
\]
Table 5.9: Definition of \( \tilde{STT}, \tilde{OWN}, \tilde{DL}, \tilde{POWN} \) and \( \tilde{REL} \) – abstract versions of \( STT, OWN, DL, POWN \) and \( REL \).

\[ \begin{align*}
\tilde{STT} : (Lck_{stt} \times Thrd_\perp \times T\text{im}_e \times Thrd_\perp \times T\text{im}_e) &\to Lck_{stt} \\
\tilde{STT}(u, T, t, T', t') &= u \\
\tilde{OWN} : (Lck_{stt} \times Thrd_\perp \times T\text{im}_e \times Thrd_\perp \times T\text{im}_e) &\to Thrd_\perp \\
\tilde{OWN}(u, T, t, T', t') &= T \\
\tilde{DL} : (Lck_{stt} \times Thrd_\perp \times T\text{im}_e \times Thrd_\perp \times T\text{im}_e) &\to T\text{im}_e \\
\tilde{DL}(u, T, t, T', t') &= t \\
\tilde{POWN} : (Lck_{stt} \times Thrd_\perp \times T\text{im}_e \times Thrd_\perp \times T\text{im}_e) &\to Thrd_\perp \\
\tilde{POWN}(u, T, t, T', t') &= T' \\
\tilde{REL} : (Lck_{stt} \times Thrd_\perp \times T\text{im}_e \times Thrd_\perp \times T\text{im}_e) &\to T\text{im}_e \\
\tilde{REL}(u, T, t, T', t') &= T'
\end{align*} \]

\( \perp \) is the bottom element, \( \top_{lock} \), if \( \forall lck \in Lck : \perp lck = (u, T, t, T', t') \) for some lock state, \( u \in \{ \text{unlocked}, \text{locked} \} \), owner, \( T \in Thrd \), and previous owner, \( T' \in Thrd \). The top element, \( \top_{lock} \), identifies the mappings, \( \top \), such that \( \forall lck \in Lck : \top lck = (u, T, t, T', t') \), for any lock state, \( u \in \{ \text{unlocked}, \text{locked} \} \), owner, \( T \in Thrd \), and previous owner, \( T' \in Thrd \).

The partial order, \( \sqsubseteq_{\text{lock}} \), greatest lower bound, \( \sqcap_{\text{lock}} \), and least upper bound, \( \sqcup_{\text{lock}} \), 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 \( \tilde{STT}, \tilde{OWN}, \tilde{DL}, \tilde{POWN} \) and \( \tilde{REL} \), as defined in Table 5.9, are the abstract versions of the masking functions \( STT, OWN, DL, POWN \) and \( REL \) (defined in Table 4.5 on page 50), respectively.
5.6 Abstract Lock States

Definition 5.31 (Partial order of abstract lock states):

\[ \begin{aligned}
\overline{\text{lock}} \subseteq \overline{\text{lock}}' & \iff \forall \text{lck} \in \text{Lck} : (\text{STT}(\overline{\text{lck}}) = \text{STT}(\overline{\text{lck}}') \land \\
\text{OWN}(\overline{\text{lck}}) = \text{OWN}(\overline{\text{lck}}') \land \\
\text{DL}(\overline{\text{lck}}) \sqsubseteq \text{DL}(\overline{\text{lck}}') \land \\
\text{POWN}(\overline{\text{lck}}) = \text{POWN}(\overline{\text{lck}}') \land \\
\text{REL}(\overline{\text{lck}}) \sqsubseteq \text{REL}(\overline{\text{lck}}'))
\end{aligned} \]

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

\[ \begin{aligned}
\overline{\text{lock}} \sqcap \overline{\text{lock}}' &= \overline{\text{lock}} \sqcap \overline{\text{lock}}' = \overline{\text{lock}} \\
\overline{\text{lock}} \sqcup \overline{\text{lock}}' &= \overline{\text{lock}} \sqcup \overline{\text{lock}}' = \overline{\text{lock}} \\
\lambda \text{lck} \in \text{Lck}.(u_{1}^{\text{lck}}, T_{1}^{\text{lck}}, r_{1}^{\text{lck}}, T_{1}^{l \text{ck}}, r_{1}^{l \text{ck}}) \sqcap \text{lock} &= \\
\lambda \text{lck} \in \text{Lck}.(u_{2}^{\text{lck}}, T_{2}^{\text{lck}}, r_{2}^{\text{lck}}, T_{2}^{l \text{ck}}, r_{2}^{l \text{ck}}) &= \\
\text{if } \forall \text{lck} \in \text{Lck} : \\
(u_{1}^{\text{lck}} = u_{2}^{\text{lck}} \land \\
T_{1}^{\text{lck}} = T_{2}^{\text{lck}} \land \\
T_{1}^{l \text{ck}} = T_{2}^{l \text{ck}})
\end{aligned} \]

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

\[ \begin{aligned}
\overline{\text{lock}} \sqcup \overline{\text{lock}}' &= \overline{\text{lock}} \sqcup \overline{\text{lock}}' = \overline{\text{lock}} \\
\overline{\text{lock}} \sqcap \overline{\text{lock}}' &= \overline{\text{lock}} \sqcap \overline{\text{lock}}' = \overline{\text{lock}} \\
\lambda \text{lck} \in \text{Lck}.(u_{1}^{\text{lck}}, T_{1}^{\text{lck}}, r_{1}^{\text{lck}}, T_{1}^{l \text{ck}}, r_{1}^{l \text{ck}}) \sqcup \text{lock} &= \\
\lambda \text{lck} \in \text{Lck}.(u_{2}^{\text{lck}}, T_{2}^{\text{lck}}, r_{2}^{\text{lck}}, T_{2}^{l \text{ck}}, r_{2}^{l \text{ck}}) &= \\
\text{if } \forall \text{lck} \in \text{Lck} : \\
(u_{1}^{\text{lck}} = u_{2}^{\text{lck}} \land \\
T_{1}^{\text{lck}} = T_{2}^{\text{lck}} \land \\
T_{1}^{l \text{ck}} = T_{2}^{l \text{ck}})
\end{aligned} \]
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 $\mathcal{I} \subseteq \mathcal{I}'$. If $\mathcal{I} = \mathcal{I}_{\text{lock}}$ or $\mathcal{I}' = T_{\text{lock}}$, then trivially, $\gamma_{\text{lock}}(\mathcal{I}) \subseteq \gamma_{\text{lock}}(\mathcal{I}')$. Otherwise, assume that $\mathcal{I} \in \gamma_{\text{lock}}(\mathcal{I}')$, $\mathcal{I}'_{\text{lock}} = (u_{\text{lock}}, T_{\text{lock}}, \bar{t}_{\text{lock}}, T'_{\text{lock}}, \bar{t}'_{\text{lock}})$ and $\mathcal{I}'_{\text{lock}} = (u'_{\text{lock}}, T''_{\text{lock}}, \bar{t}''_{\text{lock}}, T'''_{\text{lock}}, \bar{t}'''_{\text{lock}})$. Since $\mathcal{I} \subseteq \mathcal{I}'_{\text{lock}}$, it must be that $\forall \mathcal{L} \in \mathcal{L}_{\text{lock}} : (u_{\text{lock}} = u'_\text{lock} \land T_{\text{lock}} = T''_{\text{lock}} \land T'_{\text{lock}} = T'''_{\text{lock}} \land \bar{t}_{\text{lock}} \subseteq \bar{t}'_{\text{lock}} \land \bar{t}'_{\text{lock}} \subseteq \bar{t}''_{\text{lock}})$. But then, since $\mathcal{I} \in \gamma_{\text{lock}}(\mathcal{I}')$ and $\gamma$ is monotone (Theorem 3.39), it must be that $\mathcal{I} \in \gamma_{\text{lock}}(\mathcal{I}')$. 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 $\gamma_{\text{lock}}(T_{\text{lock}}) = \mathcal{L}_{\text{lock}} \rightarrow (\mathcal{L}_{\text{lock}} \times \mathcal{O}_{\text{thrd}} \times \mathcal{O}_{\text{time}} \times \mathcal{O}_{\text{thrd}} \times \mathcal{O}_{\text{time}}) = T_{\text{lock}}$.

Now, assume that $\mathcal{I}, \mathcal{I}' \in \mathcal{L}_{\text{lock}} \rightarrow (\mathcal{L}_{\text{lock}} \times \mathcal{O}_{\text{thrd}} \times \mathcal{O}_{\text{time}} \times \mathcal{O}_{\text{thrd}} \times \mathcal{O}_{\text{time}})$ are such that $\mathcal{I} \subseteq \mathcal{I}' \land \mathcal{I}' \subseteq \mathcal{I}$. From Definition 5.31, it follows that neither of $\mathcal{I}$ and $\mathcal{I}'$ can be $\bot_{\text{lock}}$ or $T_{\text{lock}}$. Thus, it is safe to assume that these states can be expressed as $\mathcal{I}_{\text{lock}} = (u_{\text{lock}}, T_{\text{lock}}, \bar{t}_{\text{lock}}, T'_{\text{lock}}, \bar{t}'_{\text{lock}})$ and $\mathcal{I}'_{\text{lock}} = (u'_{\text{lock}}, T''_{\text{lock}}, \bar{t}''_{\text{lock}}, T'''_{\text{lock}}, \bar{t}'''_{\text{lock}})$.

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

$$\gamma_{\text{lock}}(\mathcal{I} \cap \mathcal{I}'_{\text{lock}}) = \gamma_{\text{lock}}(\mathcal{I}) \cap \gamma_{\text{lock}}(\mathcal{I}')$$

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

Next, assume that $\forall \mathcal{L} \in \mathcal{L}_{\text{lock}} : (u_{\text{lock}} = u'_\text{lock} \land T_{\text{lock}} = T''_{\text{lock}} \land T'_{\text{lock}} = T'''_{\text{lock}})$ and note that $\langle \alpha_{\text{int}}, \gamma_{\text{int}} \rangle$ is a Galois connection (Theorem 3.39). Then,
\[ (\tilde{\lambda} \cap Lck \tilde{\lambda}') Lck = (u_{lck}, T_{lck}, \tilde{t}_{lck} \cap \tilde{u}_{lck}', T'_{lck}, \tilde{T}'_{lck} \cap \tilde{u}'_{lck}) \]

\[
\gamma_{lock}(\tilde{\lambda} \cap Lck \tilde{\lambda}') \overset{\text{Def.} 5.29}{=} \{ \lambda lck \in Lck. (u_{lck}, T_{lck}, t_{lck}, T'_{lck}, \tilde{u}_{lck}') \mid t_{lck} \in \gamma_{lck}(\tilde{t}_{lck} \cap \tilde{u}_{lck}') \land \tilde{u}_{lck}' \in \gamma_{lck}(\tilde{T}'_{lck} \cap \tilde{u}'_{lck}) \} \\
\overset{\text{Lem. 3.14}}{=} \{ \lambda lck \in Lck. (u_{lck}, T_{lck}, t_{lck}, T'_{lck}, \tilde{u}_{lck}') \mid t_{lck} \in \gamma_{lck}(\tilde{t}_{lck}) \land \gamma_{lck}(\tilde{T}'_{lck}) \cap \gamma_{lck}(\tilde{u}'_{lck}) \} \\
\overset{\text{calc.}}{=} \{ \lambda lck \in Lck. (u_{lck}, T_{lck}, t_{lck}, T'_{lck}, \tilde{u}_{lck}') \mid t_{lck} \in \gamma_{lck}(\tilde{t}_{lck}) \land \tilde{u}_{lck}' \in \gamma_{lck}(\tilde{T}'_{lck}) \} \cap \\
\{ \lambda lck \in Lck. (u_{lck}, T_{lck}, t_{lck}, T'_{lck}, \tilde{u}_{lck}') \mid t_{lck} \in \gamma_{lck}(\tilde{T}'_{lck}) \land \tilde{u}_{lck}' \in \gamma_{lck}(\tilde{u}'_{lck}) \} \\
\overset{\text{Def. 5.29}}{=} \gamma_{lock}(\tilde{\lambda}) \cap \gamma_{lock}(\tilde{\lambda}')
\]

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

### 5.7 Abstract Configurations

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

\[
C\text{onf} := ((\prod_{T \in \text{Thrd}_\subseteq} (\{T\} \times \text{Lbl}_T \times (\text{Reg}_T \rightarrow \text{Val}) \times \text{Time})) \times \\
(\text{Var} \rightarrow \text{Thrd} \rightarrow \mathcal{P}(\text{Val} \times \text{Time})) \times \\
(\text{Lck} \rightarrow (\text{Lck}_\text{sst} \times \text{Thrd}_\bot \times \text{Time} \times \text{Thrd}_\bot \times \text{Time})) \cup \\
\{ I_{\text{conf}}, \uparrow \}
\]

where \( \text{Thrd}_\subseteq \subseteq \text{Thrd} \) (the reason for this will become apparent when the analysis is presented in Chapter 6). The abstract configuration, \( \tilde{\epsilon} \in C\text{onf} \), will be denoted in the same manner as concrete configurations:

\[
\tilde{\epsilon} := \langle [T, pc_{\epsilon}, \tilde{\epsilon}_T, \tilde{r}_{\epsilon}] | T \in \text{Thrd}_\subseteq, \tilde{\epsilon}_T, \tilde{\epsilon} \rangle
\]

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

\[
\begin{align*}
\gamma_{\text{conf}} (\top_{\text{conf}}) &= \text{Conf} \\
\gamma_{\text{conf}} (\bot_{\text{conf}}) &= \emptyset \\
\gamma_{\text{conf}} (\{ [T, p_{T}, \tilde{t}_{T}, \tilde{a}_{T}] \mid T \in \text{Thrd}_{\tilde{c}, \tilde{I}} \}) &= \\
&= \{ [T, p_{T}, \tilde{t}_{T}, \tilde{a}_{T}] \mid T \in \text{Thrd}_{\tilde{c}, \tilde{I}} ] \}
\end{align*}
\]

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):

\[
\begin{align*}
\tilde{c} &\subseteq_{\text{conf}} \tilde{c}' \\
\bot_{\text{conf}} &\subseteq_{\text{conf}} \tilde{c}' \\
\langle [T, p_{T}, \tilde{t}_{T}, \tilde{a}_{T}] \rangle_{T \in \text{Thrd}_{\tilde{c}, \tilde{I}}} &\subseteq_{\text{conf}} \\
&\leftrightarrow \\
\langle [T, p_{T}, \tilde{t}_{T}, \tilde{a}_{T}] \rangle_{T \in \text{Thrd}_{\tilde{c}', \tilde{I}'}}, \tilde{x}' \subseteq_{\text{var}} \tilde{x}' \\
\tilde{x}' \subseteq_{\text{var}} &\bot'_{\text{lock}} \langle \tilde{I}' \rangle' \wedge \forall T \in \text{Thrd}_{\tilde{c}'}: (p_{T} = p_{T}' \wedge \tilde{t}_{T} \subseteq_{\text{reg}} \tilde{t}_{T}' \wedge \tilde{a}_{T} \subseteq_{\text{reg}} \tilde{a}_{T}')
\end{align*}
\]

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 \( \gamma_{\text{conf}} \colon \text{Conf} \to \wp (\text{Conf}) \) 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 \( \tilde{c} = \bot_{\text{conf}} \) or \( \tilde{c}' = \top_{\text{conf}} \), the lemma holds trivially. Otherwise, \( \tilde{c} \) and \( \tilde{c}' \) can be expressed as \( \tilde{c} = \langle [T, p_{T}, \tilde{t}_{T}, \tilde{a}_{T}] \rangle_{T \in \text{Thrd}_{\tilde{c}, \tilde{I}}} \) and \( \tilde{c}' = \langle [T, p_{T}', \tilde{t}_{T}', \tilde{a}_{T}'] \rangle_{T \in \text{Thrd}_{\tilde{c}', \tilde{I}'}} \). Assume that \( c \in \gamma_{\text{conf}} (\tilde{c}) \). Since \( \tilde{c} \subseteq_{\text{conf}} \tilde{c}' \), it must be that:

\[
\text{Thrd}_{\tilde{c}} = \text{Thrd}_{\tilde{c}'} \wedge \tilde{x}' \subseteq_{\text{var}} \tilde{x}' \wedge \forall T \in \text{Thrd}_{\tilde{c}'} : (p_{T} = p_{T}' \wedge \tilde{t}_{T} \subseteq_{\text{reg}} \tilde{t}_{T}' \wedge \tilde{a}_{T} \subseteq_{\text{reg}} \tilde{a}_{T}')
\]

The monotonicity of \( \gamma_{T}, \gamma_{\text{reg}}, \gamma_{\text{var}} \) and \( \gamma_{\text{lock}} \) (Theorems 3.39, 5.6 and 5.11, and Lemma 5.34, respectively) then implies that \( c \in \gamma_{\text{conf}} (\tilde{c}') \) as well. Thus, \( \gamma_{\text{conf}} (\tilde{c}) \subseteq \gamma_{\text{conf}} (\tilde{c}') \) and the lemma holds.
5.7 Abstract Configurations

The greatest lower bound operator for abstract configurations, \( \hat{\wedge}_{\text{conf}} \), 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 \( \hat{\wedge}_{\text{var}} \) cannot. A safe operator, \( \hat{\wedge}_{\text{conf}}' \), is obtained by replacing \( \hat{\wedge}_{\text{var}} \) by \( \hat{\wedge}_{\text{var}}' \) in the definition of \( \hat{\wedge}_{\text{conf}}' \).

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

\[
\begin{align*}
\hat{c} \hat{\wedge}_{\text{conf}} \hat{\top}_{\text{conf}} &= \hat{\top}_{\text{conf}} \hat{\wedge}_{\text{conf}} \hat{c} = \hat{c} \\
\hat{c} \hat{\wedge}_{\text{conf}} \hat{\bot}_{\text{conf}} &= \hat{\bot}_{\text{conf}} \hat{\wedge}_{\text{conf}} \hat{c} = \hat{c} \\
\langle [T, pc_T, \tilde{\pi}_T, \tilde{\iota}_T]_{T \in \text{Thrd}_c, \tilde{x}, \tilde{y}} \hat{\wedge}_{\text{conf}} \rangle &= \\
\langle [T, pc'_T, \tilde{\pi'}_T, \tilde{\iota'}_T]_{T \in \text{Thrd}_c', \tilde{x}', \tilde{y}'} \hat{\top}_{\text{conf}} \rangle \\
\langle [T, pc_T, \tilde{\pi}_T, \tilde{\iota}_T]_{T \in \text{Thrd}_c, \tilde{x}, \tilde{y}} \hat{\wedge}_{\text{conf}} \rangle &= \\
\langle [T, pc'_T, \tilde{\pi'}_T, \tilde{\iota'}_T]_{T \in \text{Thrd}_c', \tilde{x}', \tilde{y}'} \hat{\top}_{\text{conf}} \rangle \\
\end{align*}
\]

The least upper bound operator for abstract configurations, \( \hat{\sqcup}_{\text{conf}} \), 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 \( \hat{\sqcup}_{\text{var}} \) cannot. A safe operator, \( \hat{\sqcup}_{\text{conf}}' \), is obtained by replacing \( \hat{\sqcup}_{\text{var}} \) by \( \hat{\sqcup}_{\text{var}}' \) in the definition of \( \hat{\sqcup}_{\text{conf}}' \).

**Definition 5.40 (Least upper bound for two abstract configurations):**

\[
\begin{align*}
\hat{c} \hat{\sqcup}_{\text{conf}} \hat{\top}_{\text{conf}} &= \hat{\top}_{\text{conf}} \hat{\sqcup}_{\text{conf}} \hat{c} = \hat{\top}_{\text{conf}} \\
\hat{c} \hat{\sqcup}_{\text{conf}} \hat{\bot}_{\text{conf}} &= \hat{\bot}_{\text{conf}} \hat{\sqcup}_{\text{conf}} \hat{c} = \hat{\top}_{\text{conf}} \\
\langle [T, pc_T, \tilde{\pi}_T, \tilde{\iota}_T]_{T \in \text{Thrd}_c, \tilde{x}, \tilde{y}} \hat{\sqcup}_{\text{conf}} \rangle &= \\
\langle [T, pc'_T, \tilde{\pi'}_T, \tilde{\iota'}_T]_{T \in \text{Thrd}_c', \tilde{x}', \tilde{y}'} \rangle \\
\langle [T, pc_T, \tilde{\pi}_T, \tilde{\iota}_T]_{T \in \text{Thrd}_c, \tilde{x}, \tilde{y}} \hat{\sqcup}_{\text{conf}} \rangle &= \\
\langle [T, pc'_T, \tilde{\pi'}_T, \tilde{\iota'}_T]_{T \in \text{Thrd}_c', \tilde{x}', \tilde{y}'} \rangle \\
\end{align*}
\]

The abstraction function, \( \alpha_{\text{conf}} : \mathcal{P}(\text{Conf}) \to \text{Conf} \), 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):

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

\[ \square \]

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.

\[ \square \]

**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 \( \gamma_{\text{conf}}(\bigcap_{\text{conf}}) = \text{Conf} = \bigcup_{\text{conf}} \).

Now, assume that \( \vec{e}, \vec{e}' \in \text{Conf} \) are such that \( \vec{e} \bigcap_{\text{conf}} \vec{e}' = \bigcap_{\text{conf}} \). From Definition 5.37, it follows that neither of \( \vec{e} \) and \( \vec{e}' \) can be \( \bigcap_{\text{conf}} \) or \( \bigcap_{\text{conf}} \). Thus, it is safe to assume that these configurations can be expressed as \( \vec{e} = ([T, pc_T, \vec{t}^T, \vec{q}^T]_{T \in \text{Thrd}_\vec{e}}, \vec{x}, \vec{y}) \) and \( \vec{e}' = ([T, pc'_T, \vec{t}'_T, \vec{q}'_T]_{T \in \text{Thrd}_{\vec{e}'}} , \vec{x}' , \vec{y}') \).

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

\[ \gamma_{\text{conf}}(\bigcap_{\text{conf}} \vec{e}) = \gamma_{\text{conf}}(\vec{e}) \cap \gamma_{\text{conf}}(\vec{e}') \]

First, assume that \( \text{Thrd}_\vec{e} \neq \text{Thrd}_{\vec{e}'} \lor \exists T \in \text{Thrd}_\vec{e} : pc_T \neq pc'_T \). Then, \( \bigcap_{\text{conf}} \vec{e}' = \bigcap_{\text{conf}} \), and thus the L.H.S. becomes \( \gamma_{\text{conf}}(\bigcap_{\text{conf}} \vec{e}') = \gamma_{\text{conf}}(\bigcap_{\text{conf}}) = \emptyset \). The R.H.S. becomes \( \gamma_{\text{conf}}(\vec{e}) \cap \gamma_{\text{conf}}(\vec{e}') = \emptyset \), because it must be that \( \forall c \in \gamma_{\text{conf}}(\vec{e}) : \forall c' \in \gamma_{\text{conf}}(\vec{e}'): c \neq c' \), since \( \text{Thrd}_\vec{e} \neq \text{Thrd}_{\vec{e}'} \lor \exists T \in \text{Thrd}_\vec{e} : pc_T \neq pc'_T \). Thus, L.H.S. = R.H.S.

Next, assume that \( \text{Thrd}_\vec{e} = \text{Thrd}_{\vec{e}'} \land \forall T \in \text{Thrd}_\vec{e} : pc_T = pc'_T \) 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, \( \bigcap_{\text{conf}} \vec{e}' = ([T, pc_T, \vec{t}_T \cap \vec{q}_T]_{T \in \text{Thrd}_\vec{e}}, \vec{x} \cap \vec{y}, \vec{y} \cap \vec{y}') \)

and
\[
\gamma_{\text{conf}}(\tilde{c} \cap_{\text{conf}} \tilde{c}') \overset{\text{Def. 5.36}}{=} \{ ([T, pc_T, \oplus_T, t^{\oplus}_T]_{T \in \text{Thr}_d}, \bar{x}, \bar{1}) | \\
\oplus_T \in \gamma_{\text{reg}}(\tilde{\oplus}_T \cap_{\text{reg}} \tilde{\oplus}_T) \land t^{\oplus}_T \in \gamma_T(\tilde{t}_T \cap_{\text{reg}} \tilde{t}_T) \land \\
\bar{x} \in \gamma_{\text{var}}(\tilde{x} \cap_{\text{var}} \tilde{x}) \land \bar{1} \in \gamma_{\text{lock}}(\tilde{1} \cap_{\text{lock}} \tilde{1}) \} \\
\text{Lem. 3.14} \overset{=}{{}\{ ([T, pc_T, \oplus_T, t^{\oplus}_T]_{T \in \text{Thr}_d}, \bar{x}, \bar{1}) | \\
\oplus_T \in \gamma_{\text{reg}}(\tilde{\oplus}_T) \land t^{\oplus}_T \in \gamma_T(\tilde{t}_T) \land \\
\bar{x} \in \gamma_{\text{var}}(\tilde{x}) \land \bar{1} \in \gamma_{\text{lock}}(\tilde{1}) \} \cap \\
\{ ([T, pc_T, \oplus_T, t^{\oplus}_T]_{T \in \text{Thr}_d}, \bar{x}, \bar{1}) | \\
\oplus_T \in \gamma_{\text{reg}}(\tilde{\oplus}_T) \land t^{\oplus}_T \in \gamma_T(\tilde{t}_T) \land \\
\bar{x} \in \gamma_{\text{var}}(\tilde{x}) \land \bar{1} \in \gamma_{\text{lock}}(\tilde{1}) \} \\
\overset{\text{Def. 5.36}}{=} \gamma_{\text{conf}}(\tilde{c}) \cap \gamma_{\text{conf}}(\tilde{c}') \}
\]

Thus, it has been shown that \( \gamma_{\text{conf}}(\tilde{c} \cap_{\text{conf}} \tilde{c}') = \gamma_{\text{conf}}(\tilde{c}) \cap \gamma_{\text{conf}}(\tilde{c}') \). 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 \( \langle \alpha, \gamma_{\text{conf}} \rangle \) 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, \( \text{axConf}^{\text{in}}_T \ni \text{axConf}^{\text{out}}_T \ni \text{axConf}^{\text{out}}_T \), which will be used for the abstract axiom transition rules as presented in Table 5.10 in Section 5.8 on page 102. These domains are defined as:

\[
\text{axConf}^{\text{in}}_T := \{ \{T\} \times \text{Lbl}_T \times (\text{Reg}_T \rightarrow \text{Väl}) \times (\text{Var} \rightarrow \text{Thr}_d \rightarrow \mathcal{P}(\text{Väl} \times \text{Tiñe})) \times (\text{Lck} \rightarrow (\text{Lck}_{\text{stf}} \times \text{Thr}_d \times \text{Tiñe} \times \text{Thr}_d \times \text{Tiñe})) \times \text{Tiñe} \cup \{ \text{axConf}^{\text{in}}_T, \text{axConf}^{\text{in}}_T \} \}
\]

\[
\text{axConf}^{\text{out}}_T := \{ T, pc, \oplus, \tilde{x}, \bar{x}, \bar{1}, \bar{1} \}
\]
and:

\[ \text{axConf}_{\text{out}} := (\text{Lbl}_{\text{T}} \times (\text{Reg}_{\text{T}} \rightarrow \text{Val}) \times \] 
\[ (\text{Var} \rightarrow \text{Thrd} \rightarrow \mathcal{P}(\text{Val} \times \text{Time}) \times \] 
\[ (\text{Lck} \rightarrow (\text{Lck}_{\text{st}} \times \text{Thrd}_{\perp} \times \text{Time} \times \text{Thrd}_{\perp} \times \text{Time})) \cup \] 
\[ \{\text{ax}_{\perp}^{\text{out}} \mid \text{ax}^{\text{out}}_{\perp}\}\} \]

\[ \text{axConf}_{\text{in}} := \langle \text{T}, \text{pc}, \tilde{\tau}, \tilde{x}, \tilde{t}, \tilde{l} \rangle \]

\[ \text{axConfin} := \langle \text{T}, \text{pc}, \tilde{\tau}, \tilde{x}, \tilde{t}, \tilde{l} \rangle \]

It is easy to see that \( \langle \text{ax}_{\text{T}}^{\text{in}}, \text{ax}_{\text{T}}^{\text{in}}, \gamma_{\text{T}}^{\text{in}} \rangle \) and \( \langle \text{ax}_{\text{T}}^{\text{out}}, \text{ax}_{\text{T}}^{\text{out}}, \gamma_{\text{T}}^{\text{out}} \rangle \), where \( \text{ax}_{\text{T}}^{\text{in}} : \mathcal{P}(\text{axConf}_{\text{T}}^{\text{in}}) \rightarrow \text{axConf}_{\text{T}}^{\text{in}}, \) \( \text{ax}_{\text{T}}^{\text{out}} : \mathcal{P}(\text{axConf}_{\text{T}}^{\text{out}}) \rightarrow \text{axConf}_{\text{T}}^{\text{out}} \) and \( \text{ax}_{\text{T}}^{\text{out}} : \text{axConf}_{\text{T}}^{\text{out}} \rightarrow \mathcal{P}(\text{axConf}_{\text{T}}^{\text{out}}) \) are given by Definitions 5.43, 5.44, 5.45 and 5.46, respectively, are Galois connections (cf. Theorems 5.47 and 5.48).

**Definition 5.43 (Abstraction of a set of axiom input configurations):**

\[ ax_{\text{T}}^{\text{in}}(ax_{\text{T}}^{\text{in}}) = ax_{\text{T}}^{\text{in}}(ax_{\text{T}}^{\text{in}}) \]

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

\[ ax_{\text{T}}^{\text{in}}((\text{T}, \text{pc}, \tilde{\tau}, \tilde{x}, \tilde{t}, \tilde{l})) = \{ (\text{T}, \text{pc}, \tilde{\tau}, \tilde{x}, \tilde{t}, \tilde{l}) \mid \tilde{\tau} \in \gamma_{\text{reg}}(\tilde{\tau}) \land \tilde{x} \in \gamma_{\text{var}}(\tilde{x}) \land \] 
\[ \tilde{l} \in \gamma_{\text{lock}}(\tilde{l}) \land t \in \gamma_{\text{t}}(\tilde{l}) \} \]

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

\[ ax_{\text{T}}^{\text{out}}(ax_{\text{T}}^{\text{out}}) = ax_{\text{T}}^{\text{out}}(ax_{\text{T}}^{\text{out}}) \]

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

\[ ax_{\text{T}}^{\text{out}}((\text{pc}, \tilde{\tau}, \tilde{x}, \tilde{l})) = \{ (\text{pc}, \tilde{\tau}, \tilde{x}, \tilde{l}) \mid \tilde{\tau} \in \gamma_{\text{reg}}(\tilde{\tau}) \land \tilde{x} \in \gamma_{\text{var}}(\tilde{x}) \land \] 
\[ \tilde{l} \in \gamma_{\text{lock}}(\tilde{l}) \} \]

**Theorem 5.47 (Galois connection – Axiom input configurations):**

\( \langle ax_{\text{T}}^{\text{in}}, ax_{\text{T}}^{\text{in}} \rangle \), where \( ax_{\text{T}}^{\text{in}} \) and \( ax_{\text{T}}^{\text{out}} \) are given by Definitions 5.43 and 5.44, respectively, is a Galois connection.
PROOF. Similar to the proof of Theorem 5.42. ■

Theorem 5.48 (Galois connection – Axiom output configurations):
\(<ax_{\text{out}}^T, ax_{\text{out}}^\gamma>\), where \(ax_{\text{out}}^T\) and \(ax_{\text{out}}^\gamma\) 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 5.10 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 \(\exists T\) contains safe write history (cf. Definition 5.19), the transition relation \(\xrightarrow{\text{ax}}\) is a safe approximation of \(\xrightarrow{\text{ax}}\) iff

\[
\forall ax_{\text{in}}^T \in ax\text{Conf}^\text{in}_T : \forall ax_{\text{in}}^T \in ax\text{Conf}^\text{in}_T (ax_{\text{in}}^T) : \forall ax_{\text{out}}^T \in ax\text{Conf}^\text{out}_T :
\]

\[
(ax_{\text{in}}^T \xrightarrow{\text{ax}} ax_{\text{out}}^T) \Rightarrow \exists ax_{\text{out}}^T \in ax\text{Conf}^\text{out}_T :
\]

\[
(ax_{\text{in}}^T \xrightarrow{\text{ax}} ax_{\text{out}}^T \land ax_{\text{out}}^T \in ax\text{Conf}^\text{out}_T (ax_{\text{out}}^T))
\]

where \(ax_{\text{in}}^T\) is generated (cf. Table 4.3) 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.

Lemma 5.50 (Soundness of \(\xrightarrow{\text{ax}}\)):
\(\xrightarrow{\text{ax}}\) is a safe approximation of \(\xrightarrow{\text{ax}}\), with respect to Definition 5.49.

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

Assume that \(ax\text{Conf}^\text{in}_T \in ax\text{Conf}^\text{in}_T \in ax\text{Conf}^\text{in}_T \in ax\text{Conf}^\text{in}_T\), such that \(ax\text{Conf}^\text{in}_T \in ax\text{Conf}^\text{in}_T (ax\text{Conf}^\text{in}_T)\), that \(\exists T\) is valid with respect to \(t\) and that \(\exists T\) contains safe write history. Now consider each defined axiom statement.

1. Assume that \(\text{stm}(T, pc) = [\text{halt}]^{pc}\). From the concrete semantics, it must be that \(ax_{\text{in}}^T \xrightarrow{\text{ax}} ax_{\text{out}}^T\), where \(ax_{\text{out}}^T = \langle pc, r, \emptyset, \emptyset, t\rangle\). Choose \(ax_{\text{out}}^T\) so that \(ax_{\text{in}}^T \xrightarrow{\text{ax}} ax_{\text{out}}^T\), i.e. \(ax_{\text{out}}^T = \langle pc, r, \emptyset, \emptyset, t\rangle\). Thus, \(ax_{\text{out}}^T \in ax\text{Conf}^\text{out}_T (ax_{\text{out}}^T)\).
Table 5.10: \( \langle T, pc, \bar{x}, \bar{\bar{x}}, \bar{I} \rangle \xrightarrow{\alpha} \langle pc', \bar{x}', \bar{\bar{x}}, \bar{I}' \rangle \), semantics of abstract axiom transitions.

<table>
<thead>
<tr>
<th>STM((T, pc))</th>
<th>(\langle pc', \bar{x}', \bar{\bar{x}}, \bar{I}' \rangle)</th>
<th>If</th>
</tr>
</thead>
<tbody>
<tr>
<td>([\text{halt}]^{pc})</td>
<td>(\langle pc, \bar{x}, \bar{\bar{x}}, \bar{I} \rangle)</td>
<td>(\bar{\mathcal{R}}<em>g[!\mathring{!}b!\mathring{!}]\bar{x} \neq \bot</em>{\text{reg}})</td>
</tr>
<tr>
<td>([\text{skip}]^{pc})</td>
<td>(\langle pc, \bar{x}, \bar{\bar{x}}, \bar{I} \rangle)</td>
<td>(\bar{\mathcal{R}}<em>g[!\mathring{!}b!\mathring{!}]\bar{x} \neq \bot</em>{\text{reg}})</td>
</tr>
<tr>
<td>([r := a]^{pc})</td>
<td>(\langle pc + 1, \bar{x}, \bar{\bar{x}}, \bar{I} \rangle)</td>
<td>(\bar{\mathcal{R}}<em>g[!\mathring{!}b!\mathring{!}]\bar{x} \neq \bot</em>{\text{reg}})</td>
</tr>
<tr>
<td>([\text{if } b \text{ goto } l]^{pc})</td>
<td>(\langle pc + 1, \bar{x}, \bar{\bar{x}}, \bar{I} \rangle)</td>
<td>(\bar{\mathcal{R}}<em>g[!\mathring{!}b!\mathring{!}]\bar{x} \neq \bot</em>{\text{reg}})</td>
</tr>
<tr>
<td>([\text{if } b \text{ goto } l]^{pc})</td>
<td>(\langle pc + 1, \bar{x}, \bar{\bar{x}}, \bar{I} \rangle)</td>
<td>(\bar{\mathcal{R}}<em>g[!\mathring{!}b!\mathring{!}]\bar{x} \neq \bot</em>{\text{reg}})</td>
</tr>
<tr>
<td>([\text{store } r \text{ to } x]^{pc})</td>
<td>(\langle pc + 1, \bar{x}, \bar{\bar{x}}, \bar{I} \rangle)</td>
<td>OWN((\bar{I} lck) = T \land)</td>
</tr>
<tr>
<td>([\text{load } r \text{ from } x]^{pc})</td>
<td>(\langle pc + 1, \bar{x}, \bar{\bar{x}}, \bar{I} \rangle)</td>
<td>(sTT(\bar{I} lck) = unlocked \Rightarrow)</td>
</tr>
<tr>
<td>([\text{lock } lck]^{pc})</td>
<td>(\langle pc + 1, \bar{x}, \bar{\bar{x}}, \bar{I} lck \rightarrow (\text{locked}, T, DL(\bar{I} lck), \text{POWN}(\bar{I} lck), REL(\bar{I} lck))))</td>
<td>((\bar{i} \not&lt;_{\bar{I}} REL(\bar{I} lck) \land)</td>
</tr>
<tr>
<td>([\text{unlock } lck]^{pc})</td>
<td>(\langle pc + 1, \bar{x}, \bar{\bar{x}}, \bar{I} lck \rightarrow (\text{locked}, \perp_{\text{shrd}}, DL(\bar{I} lck), T, \bar{I})\rangle)</td>
<td>OWN(((\bar{I} lck) = T \land)</td>
</tr>
<tr>
<td>([\text{unlock } lck]^{pc})</td>
<td>(\langle pc + 1, \bar{x}, \bar{\bar{x}}, \bar{I} \rangle)</td>
<td>sTT(((\bar{I} lck) = locked \land)</td>
</tr>
</tbody>
</table>


2. Assume that $\text{STM}(T, pc) = [\text{skip}]^{pc}$. From the concrete semantics, it must be that $\frac{ax_{in}}{ax_{out}}$, where $\frac{ax_{out}}{ax_{out}} = \langle pc + 1, r, x, \varepsilon \rangle$. Choose $\frac{ax_{out}}{ax_{out}}$ so that $\frac{ax_{in}}{ax_{out}}$, i.e. $\frac{ax_{out}}{ax_{out}} = \langle pc + 1, \varepsilon, \underbar{x}, \underbar{\varepsilon} \rangle$. Thus, $\frac{ax_{out}}{ax_{out}} \in \frac{ax_{out}}{ax_{out}}(\frac{ax_{out}}{ax_{out}}).

3. Assume that $\text{STM}(T, pc) = [r := a]^{pc}$. From the concrete semantics, it must be that $\frac{ax_{in}}{ax_{out}}$, where $\frac{ax_{out}}{ax_{out}} = \langle pc + 1, r[r \mapsto a], \underbar{x}, \underbar{\varepsilon} \rangle$. Choose $\frac{ax_{out}}{ax_{out}}$ so that $\frac{ax_{in}}{ax_{out}}$, i.e. $\frac{ax_{out}}{ax_{out}} = \langle pc + 1, \varepsilon, a \rangle$. Since $\mathcal{A}$ is safely induced from $\mathcal{A}$ (see Section 5.3), it must be that $\mathcal{A}[a]_{\gamma} \in \gamma_{out}(\mathcal{A}[a]_{\varepsilon})$, and hence, $\gamma_{reg}(\mathcal{T}[r \mapsto a])$. Thus, $\frac{ax_{out}}{ax_{out}} \in \frac{ax_{out}}{ax_{out}}(\frac{ax_{out}}{ax_{out}}).

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, $\mathcal{B}[b]_{\varepsilon}$. This means that $\frac{ax_{in}}{ax_{out}}$, where $\frac{ax_{out}}{ax_{out}} = \langle l, \varepsilon, x, \varepsilon \rangle$. Now, choose $\frac{ax_{out}}{ax_{out}}$ so that $\frac{ax_{in}}{ax_{out}}$, i.e. $\frac{ax_{out}}{ax_{out}} = \langle l, \mathcal{B}[b]_{\varepsilon}, \underbar{x}, \underbar{\varepsilon} \rangle$. Since $\mathcal{B}$ is safely induced from $\mathcal{B}$ (see Section 5.4), it must be that $\mathcal{B}[b]_{\varepsilon} \neq \mathcal{I}_{\text{reg}}$ and that $\frac{ax_{out}}{ax_{out}}(\frac{ax_{out}}{ax_{out}})$. Thus, the case that $\frac{ax_{out}}{ax_{out}} \in \frac{ax_{out}}{ax_{out}}(\frac{ax_{out}}{ax_{out}}).

(b) In the second case, $\neg\mathcal{B}[b]_{\varepsilon}$. This means that $\frac{ax_{in}}{ax_{out}}$, where $\frac{ax_{out}}{ax_{out}} = \langle pc + 1, \varepsilon, x, \varepsilon \rangle$. Now, choose $\frac{ax_{out}}{ax_{out}}$ so that $\frac{ax_{in}}{ax_{out}}$, i.e. $\frac{ax_{out}}{ax_{out}} = \langle pc + 1, \mathcal{B}[b]_{\varepsilon}, \underbar{x}, \underbar{\varepsilon} \rangle$. Since $\mathcal{B}$ is safely induced from $\mathcal{B}$ (see Section 5.4), it must be that $\mathcal{B}[b]_{\varepsilon} \neq \mathcal{I}_{\text{reg}}$ and that $\frac{ax_{out}}{ax_{out}}(\frac{ax_{out}}{ax_{out}})$. Thus, the case that $\frac{ax_{out}}{ax_{out}} \in \frac{ax_{out}}{ax_{out}}(\frac{ax_{out}}{ax_{out}}).

5. Assume that $\text{STM}(T, pc) = [\text{store } r \text{ to } x]^{pc}$. From the concrete semantics, it must be that $\frac{ax_{in}}{ax_{out}}$, where $\frac{ax_{out}}{ax_{out}} = \langle pc + 1, r, \underbar{x}, \underbar{\varepsilon} \rangle$. Choose $\frac{ax_{out}}{ax_{out}}$ so that $\frac{ax_{in}}{ax_{out}}$, i.e. $\frac{ax_{out}}{ax_{out}} = \langle pc + 1, \varepsilon, \text{WRITE}(T, \underbar{x}, r, \varepsilon) \rangle$. It is easy to see that (cf. Algorithm 5.5) $\frac{ax_{out}}{ax_{out}} \in \gamma_{out}(\text{WRITE}(T, \underbar{x}, x, \varepsilon))$, thus $\frac{ax_{out}}{ax_{out}} \in \frac{ax_{out}}{ax_{out}}(\frac{ax_{out}}{ax_{out}}).
6. Assume that $\text{STM}(T, pc) = [\text{load } r \text{ from } x]^pc$. From the concrete semantics, $ax^{in}_{T \rightarrow ax^{out}}$, where $ax^{out}_{T} = \langle pc + 1, r, t[\rightarrow v], x, \emptyset \rangle$ for some $v$ such that $\exists t' \in \text{Time} : (v, t') \in \bigcup_{T \in \text{Thrd}} ((x, T') \neq \emptyset$ and $v \in \gamma_{int}(\langle -\infty, \infty \rangle)$ otherwise. Choose $ax^{out}_{T}$ so that $ax^{in}_{T \rightarrow ax^{out}}$, i.e. $ax^{out}_{T} = \langle pc + 1, \emptyset[r \rightarrow \text{READ}(x, T, \emptyset)], \emptyset, \emptyset \rangle$. Since $x$ is safe at time $\emptyset$ and READ then returns a safe value (Lemma 5.27), it must be that $v \in \gamma_{val}(\text{READ}(x, T, \emptyset))$ and thus $ax^{out}_{T} \in ax^{out}_{T}(ax^{out}_{T})$.

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

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

(b) In the second case, $\text{OWN}(lck) \neq T$. From the concrete semantics, it must be that $ax^{in}_{T \rightarrow ax^{out}}$, where $ax^{out}_{T} = \langle pc, r, x, \emptyset \rangle$. Choose $ax^{out}_{T}$ so that $ax^{in}_{T \rightarrow ax^{out}}$ by the corresponding branch, $\text{OWN}(lck) \neq T \lor (STT(lck) = \text{unlocked} \land (T \not\in \text{REL}(lck)) \lor (\text{DL}(lck) \not\in T))$; i.e. $ax^{out}_{T} = \langle pc, \emptyset, \emptyset, \emptyset[lck \rightarrow (\text{unlocked}, T, \text{DL}(lck), T, \emptyset)] \rangle$. Thus, it must be the case that $ax^{out}_{T} \in ax^{out}_{T}(ax^{out}_{T})$.

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

(a) In the first case, $\text{OWN}(lck) = T$. From the concrete semantics, it must be that $ax^{in}_{T \rightarrow ax^{out}}$, where $ax^{out}_{T} = \langle pc + 1, r, x, \emptyset[lck \rightarrow (\text{unlocked}, \bot_{\text{thrd}}, \text{DL}(lck), T, T) \rangle$. Choose $ax^{out}_{T}$ so that $ax^{in}_{T \rightarrow ax^{out}}$ by the corresponding branch, $(\text{OWN}(lck) = T \land STT(lck) = \text{locked})$; i.e. $ax^{out}_{T} = \langle pc + 1, \emptyset, \emptyset, \emptyset[lck \rightarrow (\text{unlocked}, \bot_{\text{thrd}}, \text{DL}(lck), T, T) \rangle$. Note that in the concrete case, $\text{STT}(lck) = \text{locked} whenever \text{OWN}(lck) \neq \bot_{\text{thrd}}$ for a valid configuration (Definition 4.4). Thus, it must be the case that $ax^{out}_{T} \in ax^{out}_{T}(ax^{out}_{T})$.
(b) In the second case, OWN(∥ lck) ≠ T. From the concrete semantics, it must be that \( ax_{cT}^{in} \xrightarrow{ax} ax_{cT}^{out} \), where \( ax_{cT}^{out} = \langle pc + 1, t, x, \lambda \rangle \).

Choose \( ax_{cT}^{out} \) so that \( ax_{cT}^{in} \xrightarrow{ax} ax_{cT}^{out} \) by the corresponding branch, OWN(∥ lck) = UNLOCKED; i.e. \( ax_{cT}^{out} = \langle pc + 1, t, x, \lambda \rangle \). Thus, it must be the case that \( ax_{cT}^{out} \in ax_{cT}^{out} (ax_{cT}^{in}) \).]

The abstract transition rule for program configurations in Table 5.11 is an approximation of the concrete rule in Table 4.3. The abstract rule now defines a window in time, \( \tau \), (since \( Ti\neq = Intv \)) that determines which threads are included in \( \text{Thrd}_{exe} \). 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 DLLOCK and ACCTIME are defined in Algorithms 5.11 and 5.12 (▷ begins a comment; cf. Appendix A), respectively, and that ABSTIME is assumed to be a safe approximation of TIME, as specified in Assumption 5.51. The definition of 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 ABSTIME is a “non-negative” function that safely approximates TIME in the interval domain. More formally, it is assumed that

\[
\forall c \in \{ [T, pc_T, t_T, \tau_T] \in \text{Thrd}_{exe}, x, \lambda \} \in \text{Conf} : \\
\forall T \in \text{Thrd}_\tau : 0 \leq \min(\gamma_t(\text{ABSTIME}(c, T)))
\]

and

\[
\forall c \in \{ [T, pc_T, t_T, \tau_T] \in \text{Thrd}_{exe}, x, \lambda \} \in \text{Conf} : \\
\forall c \in \{ [T, pc_T, t_T, \tau_T] \in \text{Thrd}_{exe}, x, \lambda \} \in \text{Conf} : \\
(\text{Thrd}_\tau \subseteq \text{Thrd} \Rightarrow \forall T \in \text{Thrd}_\tau : ((pc_T = pc_T^\tau \land t_T^\tau \in \gamma_t(\tau_T^\tau)) \Rightarrow \\
\text{TIME}(c, T) \in \gamma_t(\text{ABSTIME}(c, T))))
\]
Table 5.11: $c \xrightarrow{\text{prg}} c'$, semantics of abstract program transitions.

$$\begin{align*}
\text{Thrd}_{\text{exe}} & \neq 0 \land \forall T \in \text{Thrd}_{\text{exe}} : \langle T, pc_{T}, x_{T}, \tilde{x}, \tilde{x}', \tilde{i}'_{T} \rangle \xrightarrow{\delta_{\text{exe}}} \langle pc'_{T}, \tilde{x}', \tilde{i}'_{T} \rangle \\
\tilde{c} @ \langle T, pc_{T}, x_{T}, \tilde{i}'_{T} \rangle_{T \in \text{Thrd}_{\text{exe}}} & \tilde{x}, \tilde{i}' \xrightarrow{\text{prg}} \tilde{c}' @ \langle \langle T, (T \in \text{Thrd}_{\text{exe}} ? pc'_{T} : pc_{T}), (T \in \text{Thrd}_{\text{exe}} ? \tilde{x}'_{T} : \tilde{x}_{T}, \tilde{i}'_{T} \rangle_{T \in \text{Thrd}_{\text{exe}}} \tilde{x}', \tilde{i}' \rangle \\
\text{where} & \\
\tilde{i}'_{T} & = \text{ABSTIME}(\tilde{c}, T) \\
\tilde{i}_{T} & = \alpha_{l}(\min(\{\min(y_{l}(\tilde{i}'_{T} + \tilde{i}_{T})), B\}), \text{ where } B \iff T \in \text{Thrd}_{\text{exe}} \land \text{STM}(T, pc_{T}) \neq [\text{halt}]^{pc_{T}} \land \forall l \in \text{Lck} : \min(\{\max(y_{l}(\tilde{i}'_{T} + \tilde{i}_{T})), B\})) \\
\text{Thrd}_{\text{exe}}^{\text{unlock}} & = \{T \in \text{Thrd}_{\text{exe}} | \tilde{i}_{all} \cap (\tilde{i}'_{T} + \tilde{i}_{T} \neq \tilde{i}_{T}) \land \text{STM}(T, pc_{T}) \neq [\text{halt}]^{pc_{T}}\} \\
\tilde{z}_{lck} & = \begin{cases} (\text{unlocked}, T'), \text{DLLOCK}(\tilde{c}, lck), & \text{for some } T' \in \{T \in \text{Thrd}_{\text{exe}} | \exists l \in \text{Lck} : \text{STM}(T, l) = [\text{lock } l]^{lck}\}, \\
\text{POWN}(\tilde{lck}), \text{REL}(\tilde{lck})) & \text{if } \exists T \in \text{Thrd}_{\text{exe}}^{\text{unlock}} : \text{STM}(T, pc_{T}) = [\text{lock } lck]^{pc_{T}} \land \text{OWN}(\tilde{lck}) = \bot_{\text{thrd}} \\
\tilde{a}_{lck} & \text{otrW} & \end{cases} \\
\text{Thrd}_{\text{hold}} & = \{T \in \text{Thrd}_{\text{exe}} | \exists l \in \text{Lck} : \text{STM}(T, pc_{T}) = [\text{lock } lck]^{pc_{T}} \land \text{OWN}(\tilde{lck}) = \bot_{\text{thrd}} \} \\
\tilde{i} & = \alpha_{l}(\min(\{\min(y_{l}(\tilde{i}'_{T} + \tilde{i}_{T})), T \in \text{Thrd}_{\text{exe}} \setminus \text{Thrd}_{\text{hold}}\}), \min(\{\max(y_{l}(\tilde{i}'_{T} + \tilde{i}_{T})), T \in \text{Thrd}_{\text{exe}} \setminus \text{Thrd}_{\text{hold}}\})) \\
\text{Thrd}_{\text{exe}} & = \{T \in \text{Thrd}_{\text{exe}} \setminus \text{Thrd}_{\text{hold}} | \tilde{i}_{all} \cap (\tilde{i}'_{T} + \tilde{i}_{T} \neq \tilde{i}_{T})\} \\
\tilde{z}_{lck} & = \begin{cases} \tilde{a}_{lck} & \text{if } \text{OWN}(\tilde{lck}) \in \text{Thrd}_{\text{exe}} \\
\text{POWN}(\tilde{lck}), \text{REL}(\tilde{lck}) & \text{otrW} \end{cases} \\
\tilde{x}' & = \begin{cases} \text{TRIM}(\tilde{x}, \tilde{i}) & \text{if } \text{Thrd}_{\text{exe}} = \text{Thrd} \\
\tilde{x}' & \text{otrW} \end{cases} \text{ where } (\tilde{x}', x) \text{T} = \begin{cases} (\tilde{x}'_{T}, x) & \text{T} \text{ if } T \in \text{Thrd}_{\text{exe}} \\
(\tilde{x}, x) & \text{otrW} \end{cases} \\
\tilde{i}'_{T} & = \text{ACCTIME}(\langle T', pc_{T}, x_{T}, \tilde{i}'_{T} \rangle_{T' \in \text{Thrd}_{\text{exe}}}, \tilde{x}'_{T}, \tilde{i}'_{T}, T \rangle)
\end{align*}$$
Algorithm 5.11 Determine deadline for lock owner assignment
1: function DLLOCK(\(\tilde{e}@([T', pc_T', \tilde{x}_T', \tilde{\alpha}_T]_{T \in \text{Thrd}_d}, \tilde{x}, \tilde{l}, lck)\))
2: \(\tilde{\iota}_{dl} \leftarrow \tilde{\iota}_l\)
3: for all \(T \in \text{Thrd}_d\) do
4: if \(\text{STM}(T, pc_T) = \{\text{lock} lck\}^{pc_T}\) then
5: \(\tilde{e}' \leftarrow \tilde{e}\)
6: \(\tilde{\iota}_{dl} \leftarrow \tilde{\iota}_{dl}'\)
7: \(\tilde{\iota}_{dl} \leftarrow \alpha_T(\{-\infty\})\)
8: repeat
9: \(\tilde{\iota}' \leftarrow \text{ABSTIME}(\tilde{e}', T)\)
10: \(\tilde{\iota}_{dl}' \leftarrow \tilde{\iota}_{dl} \cup \tilde{\iota}'\)
11: \(\tilde{\iota}_{dl} \leftarrow (\tilde{\iota}_{dl} \cup T) \cap \alpha_T(\{-\infty\})\)
12: \(\tilde{\iota}' \leftarrow (\{T', pc_T', \tilde{x}_T'\}, (T = T' ? \tilde{\iota}_{dl}' : \tilde{\iota}_{dl}'))_{T \in \text{Thrd}_d}, \tilde{x}, \tilde{l}\)\)
13: until \(0 \in \gamma_T(\tilde{\iota}') \lor \tilde{\iota}_{dl}' = \top\)
14: if \(\tilde{\iota}_{dl}' \neq \bot \land 0 \notin \gamma_T(\tilde{\iota}')\) then
15: \(\tilde{\iota}_{dl} \leftarrow (\tilde{\iota}_{dl} \cup T) \cap \alpha_T(\{\infty\})\)
16: \(\tilde{\iota}' \leftarrow (\{T', pc_T', \tilde{x}_T'\}, (T = T' ? \tilde{\iota}_{dl}' : \tilde{\iota}_{dl}'))_{T \in \text{Thrd}_d}, \tilde{x}, \tilde{l}\)\)
17: \(\tilde{\iota}_{dl} \leftarrow \tilde{\iota}_{dl} \triangleq T \text{ABSTIME}(\tilde{e}', T)\)
18: \(\tilde{\iota}_{dl} \leftarrow \tilde{\iota}_{dl} \cup \tilde{\iota}_{dl}'\)
19: \(\tilde{\iota}_{dl} \leftarrow \tilde{\iota}_{dl} \cup \tilde{\iota}_{dl}'\)
20: else
21: \(\tilde{\iota}_{dl} \leftarrow \tilde{\iota}_{dl} \cup \tilde{\iota}_{dl}'\)
22: end if
23: end if
24: end for
25: return \(\tilde{\iota}_{dl}\)
26: end function

Algorithm 5.12 Determine accumulated execution time
1: function ACCTIME(\(\tilde{e}@([T', pc_T', \tilde{x}_T', \tilde{\alpha}_T]_{T \in \text{Thrd}_d}, \tilde{x}, \tilde{l}, \text{Thrd}_{exe}, T)\))
2: \(\tilde{\iota}_{dl} \leftarrow \tilde{\iota}_{dl}'\)
3: if \(T \in \text{Thrd}_{exe}\) then
4: \(\tilde{\iota}_{dl} \leftarrow \text{ABSTIME}(\tilde{e}, T)\)
5: if \(\forall lck \in \text{Lck} : \text{STM}(T, pc_T) \neq \{\text{lock} lck\}^{pc_T}\) then
6: \(\tilde{\iota}_{dl}' \leftarrow \tilde{\iota}_{dl} \triangleq T\)
7: else

Algorithm 5.12 Cont. Determine accumulated execution time

8:     
9:     for all lck ∈ Lck do
10:        if STM(T, pcT) = {lock lck}pcT ∧ OWN(⃗lck) = T then
11:            if S⃗tT(⃗lck) = locked then
12:                ⃗t′′ tT ← ⃗t′ + tT
13:            else if ⃗DL(⃗lck) ⊈ (⃗t′ + ⃗t′′ T) then
14:                ⃗t′′ tT ← ⃗t′ + tT
15:            else if (⃗t′ + ⃗t′′ T) ⊈ REL(⃗lck) then
16:                ⃗c′ ← ⃗c
17:                while (⃗t′′ tT, ABSTIME(⃗c′, T)) ⊈ REL(⃗lck) do
18:                    ⃗t′′ tT ← ⃗t′ + ABSTIME(⃗c′, T)
19:                ⃗c′ ← (⃗t′′ T, pcT, ⃗eT, (T = T′ ? ⃗t′′ T : ⃗t′′ T)) ∈ Thrd,c, ⃗x, ⃗lck)
20:            end while
21:        else if POWN(⃗lck) = T ∨ REL(⃗lck) ⊈ (⃗t′ + ⃗t′′ T) then
22:            ⃗t′′ tT ← ⃗t′ + ⃗t′′ T
23:            else if 0 ∈ γI(ABSTIME(⃗c′, T)) then
24:                ⃗c′ ← (⃗t′ + ⃗t′′ T) ⊈ ⃗eT, REL(⃗lck)
25:            end if
26:        end if
27:    end for
28: else if 0 ∈ γI(ABSTIME(⃗c′, T)) then
29:    ⃗c′ ← (⃗t′′ T, pcT, ⃗eT, (T = T′ ? ⃗t′′ T : ⃗t′′ T)) ∈ Thrd,c, ⃗x, ⃗lck)
30:    end if
31: end if
Since \textbf{Time} is approximated using \textbf{Tim} = \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 $\rightarrow_{\text{prg}}$ (cf. Tables 4.3 and 5.11) 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 $\bar{c} \in \textbf{Conf}$, even if $c \in \gamma_{\text{conf}}(\bar{c})$. 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 cannot be used to find a safe over-approximation of the concrete collecting semantics (see for example [23, 35]).

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, $\xrightarrow{\text{prg}}$ does not contain safe write history). To see this, assume that for some abstract configuration, $\textbf{Thrd}_{\text{exe}} = \{T_1, T_2\}$, $\text{STM}(T_1, pc_{T_1}) = [\texttt{load } r \texttt{ from } x]^{pc_{T_1}}$, $\text{STM}(T_2, pc_{T_2}) = [\texttt{skip}]^{pc_{T_2}}$ and $\text{STM}(T_2, pc_{T_2} + 1) = [\texttt{store } r' \texttt{ to } x]^{pc_{T_2} + 1}$. When a transition occurs, the \texttt{load-} and \texttt{skip}-statements are considered. However, since the execution time of the \texttt{store}-statement (the abstract “point” in time when the thread’s $pc$ is updated) overlaps with the execution time of the \texttt{load}-statement, 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 infinite load threads; i.e. either no thread issues a load (whenever it terminates) safely approximate the timing bounds of any infinite thread never acquires the lock at all with its future statements). There-

\[ \begin{align*}
\text{Reg}_T & = \exists x \in \text{Var}_g : \text{STM}(T, pc_T) = \{\text{load } r \text{ from } x\}^{pc_T} = 0, \\
& \text{Var}_g \text{ is the set of all global variables (i.e. variables that might transfer data between threads); i.e. either no thread issues a load statement on a global variable, or there is such a thread and it is the sole thread that is executed, which means that } \not\exists x \text{ must contain safe write history since no more writes on } x \text{ can occur before the given load statement has been executed. It should be noted that outdated writes (i.e. writes that will never be considered by a load statement in any thread) are trimmed away from the variable store resulting from a transition, } \not\exists x', \text{ given that } \text{Thrd}_{exe} = \text{Thrd} \text{ (the reason for this condition will become apparent in Chapter 6 where a recursive worklist algorithm, encapsulating } \not\to^{pg}, \text{ is presented).}
\end{align*} \]

One thing to notice from how \( \not\to^{pg} \) is defined is that an abstract configuration cannot have the same restrictions for it being valid as a concrete configuration does (cf. Definition 4.4). When a thread (in \( \text{Thrd}_{exe} \)) wants to acquire some free lock, \( \not\to^{pg} \) 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). There-
fore, an abstract configuration, \( \tilde{c} @ (\langle [T, pc_T, \tilde{\tau}_T, \tilde{t}_T]_{T \in \text{Thrd}_{c}}, \tilde{x}, \tilde{l} \rangle) \in \text{Conf} \), must be considered temporarily valid even if \( \exists lck \in \text{Lck} : (\text{OWN} (\tilde{l} lck) \neq \perp_{\text{thr}} \land s\tilde{T}(\tilde{l} lck) = \text{unlocked}) \). As also discussed in 3 above, however, such an abstract configuration can be considered invalid if \( \exists lck \in \text{Lck} : (\text{OWN} (\tilde{l} lck) \neq \perp_{\text{thr}} \land s\tilde{T}(\tilde{l} lck) = \text{unlocked} \land (\text{DL} (\tilde{l} lck) \prec_t (\tilde{t}_T^{a}_{\text{OWN} (\tilde{l} lck)} + t, \text{ABSTIME} (\tilde{c}, \text{OWN} (\tilde{l} lck))) \lor \text{STM} (\text{OWN} (\tilde{l} lck), pc_{\text{OWN} (\tilde{l} lck)}) = [\text{halt}^{pc \text{OWN} (\tilde{l} lck)}]) \), given that \( \text{DL} (\tilde{l} lck) \) is a safe approximation of when \( lck \) 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 \( \tilde{\tau}_{\text{all}}, \text{Thrd}_{\text{all}}, \tilde{\tau}'' \), \( \text{Thrd}_{\text{hold}}, \tilde{\tau} \) and \( \text{Thrd}_{\text{exe}} \) in Table 5.11.) 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 \( \tilde{t}_T^{a} \) 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 4.3 and Lemma 4.5). The purpose of \( \text{DL.LOCK} \), defined in Algorithm 5.11 on page 107, 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 all other threads.

**Lemma 5.52 (Time accumulation):**
Given the two configurations \( c @ (\langle [T, pc_T, \tau_T, t_T^{a}]_{T \in \text{Thrd}_{c}}, x, l \rangle) \in \text{Conf} \) and \( \tilde{c} @ (\langle [T, pc_T, \tilde{\tau}_T, \tilde{t}_T^{a}]_{T \in \text{Thrd}_{\tilde{c}}}, \tilde{x}, \tilde{l} \rangle) \in \text{Conf} \), such that \( \text{Thrd}_{c} \subseteq \text{Thrd} \), let \( \text{Thrd}' = \{ T \in \text{Thrd}_{c} | t_T^{a} \in \gamma_t (\tilde{t}_T^{a}) \land pc_T = pc_T^{\tilde{c}} \} \). Then \( \forall T \in \text{Thrd}' : (t_T^{a} + \text{TIME} (c, T)) \in \gamma_t (\tilde{t}_T^{a} + t, \text{ABSTIME} (\tilde{c}, T)) \). □

**Proof:** Assume that the configurations \( c @ (\langle [T, pc_T, \tau_T, t_T^{a}]_{T \in \text{Thrd}_{c}}, x, l \rangle) \in \text{Conf} \) and \( \tilde{c} @ (\langle [T, pc_T, \tilde{\tau}_T, \tilde{t}_T^{a}]_{T \in \text{Thrd}_{\tilde{c}}}, \tilde{x}, \tilde{l} \rangle) \in \text{Conf} \) are such that \( \text{Thrd}_{c} \subseteq \text{Thrd} \), and let \( \text{Thrd}' = \{ T \in \text{Thrd}_{c} | t_T^{a} \in \gamma_t (\tilde{t}_T^{a}) \land pc_T = pc_T^{\tilde{c}} \} \). Then, according to Assumption 5.51, \( \forall T \in \text{Thrd}' : \text{TIME} (c, T) \in \gamma_t (\text{ABSTIME} (\tilde{c}, T)) \). Since \( \forall T \in \text{Thrd}' : t_T^{a} \in \gamma_t (\tilde{t}_T^{a}) \), it is easy to see that \( \forall T \in \text{Thrd}' : (t_T^{a} + \text{TIME} (c, T)) \in \gamma_t (\tilde{t}_T^{a} + t, \text{ABSTIME} (\tilde{c}, T)) \). ■

**Lemma 5.53 (Thread isolation):**
If the two configurations \( c^{0} @ (\langle [T, pc_T^{0}, \tau_T^{0}, t_T^{a0}]_{T \in \text{Thrd}_{c}^{0}}, x^{0}, l^{0} \rangle) \in \text{Conf} \) and \( \tilde{c}^{0} @ (\langle [T, pc_T^{0}, \tilde{\tau}_T^{0}, \tilde{t}_T^{a0}]_{T \in \text{Thrd}_{\tilde{c}^{0}}}, \tilde{x}^{0}, \tilde{l}^{0} \rangle) \in \text{Conf} \),
\( \langle [T, pc^0_T, \tilde{z}^0_T, \tilde{a}^0_T] \rangle \in Conf, \) and some thread, \( T \in Thr\), are such that \( Thr \subseteq Thr, t^0_T \in \gamma_T(t^0_T) \) and \( pc^0_T = pc^0_T \), and some configuration \( c^{n+1} @ \langle [T, pc^n_T, \tilde{z}^n_T, \tilde{a}^n_T] \rangle \in Thr, \) where

\[
\begin{align*}
&c^0 \xrightarrow{prg} \ldots \xrightarrow{prg} c^1 \xrightarrow{prg} \ldots \xrightarrow{prg} c^2 \xrightarrow{prg} \ldots \xrightarrow{prg} c^n \xrightarrow{prg} c^{n+1}
\end{align*}
\]

for some \( n \geq 0 \), then

\[
\begin{align*}
t^0_T & \in \gamma_T(t^0_T) +_t \text{ABSTIME}(c^0_T, T) +_t \text{ABSTIME}(c^1_T, T) +_t \text{ABSTIME}(c^2_T, T) +_t \ldots +_t \text{ABSTIME}(c^n_T, T)
\end{align*}
\]

given that \( \forall i \in \{1, 2, \ldots, n\} : \tilde{t}^i_T = \tilde{t}^{i-1}_T +_t \text{ABSTIME}(c^{i-1}_T, T) \), \( \forall i \in \{0, 1, 2, \ldots, n\} : T \in Thr_{c^{i}}, \) and \( \forall c \in \{c^0, c^1, c^2, \ldots, c^n\} \setminus \{c^0, c^1, c^2, \ldots, c^n\} : T \not\in Thr_{c^{i}}, \) where \( Thr_{c^{i}} \) and \( Thr_{c^{i}} \) are as defined in Table 4.3 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 \( c^i \) etc. but for any such configuration, \( T \not\in Thr_{c^{i}}. \)

**Proof.** Assume that the configurations \( c^0 @ \langle [T, pc^0_T, \tilde{z}^0_T, \tilde{a}^0_T] \rangle \in Conf \) and \( c^0 @ \langle [T, pc^0_T, \tilde{z}^0_T, \tilde{a}^0_T] \rangle \in Conf, \) and some thread, \( T \in Thr, \) are such that \( Thr \subseteq Thr, t^0_T \in \gamma_T(t^0_T) \) and \( pc^0_T = pc^0_T \). Also assume that \( c^0 \xrightarrow{prg} \ldots \xrightarrow{prg} c^1 \xrightarrow{prg} \ldots \xrightarrow{prg} c^2 \xrightarrow{prg} \ldots \xrightarrow{prg} c^n \xrightarrow{prg} c^{n+1} \) for some configuration \( c^{n+1} @ \langle [T, pc^n_T, \tilde{z}^n_T, \tilde{a}^n_T] \rangle \in Thr, \) and \( n \geq 0 \), for which \( \forall i \in \{0, 1, 2, \ldots, n\} : T \in Thr_{c^{i}}, \) and \( \forall c \in \{c^0, c^1, c^2, \ldots, c^n\} \setminus \{c^0, c^1, c^2, \ldots, c^n\} : T \not\in Thr_{c^{i}}, \) where \( Thr_{c^{i}} \) and \( Thr_{c^{i}} \) are as defined in Table 4.3 for all \( c^i \) and all other \( c \) on the trace from \( c^0 \) to \( c^{n+1} \).

From Table 4.3, it is easy to see that:

\[
\begin{align*}
t^0_T &= t^0_T \\
t^1_T &= t^1_T + \text{TIME}(c^0_T, T) \\
t^2_T &= t^1_T + \text{TIME}(c^1_T, T) = t^0_T + \text{TIME}(c^0_T) + \text{TIME}(c^1_T) \\
&\vdots \\
t^n_T &= t^n_T + \text{TIME}(c^n_T) = t^0_T + \sum_{i=0}^{n} \text{TIME}(c^i, T)
\end{align*}
\]
Let \( \{c^0, c^1, c^2, \ldots, c^n @ \langle [T, pc^0_T, \tilde{n}^0_T, \tilde{a}^0_T] \rangle_{T \in \text{Thrd}} \} \) be a set of some abstract configurations such that \( c^0 \) has the properties assumed above, \( \forall i \in \{1, 2, \ldots, n\} : pc^i_T = pc^e_T \) and \( \forall i \in \{1, 2, \ldots, n\} : \tilde{a}^i_T = \tilde{a}^{i-1}_T \). Then, according to Lemma 5.52:

\[
(t^0_T + \text{TIME}(c^0, T)) \in \gamma (\tilde{a}^0_T, \text{ABSTIME}(c^0, T))
\]

\[
(t^n_T + \text{TIME}(c^n, T)) \in \gamma (\tilde{a}^n_T, \text{ABSTIME}(c^n, T))
\]

Since \( t^{n+1}_T = t^n_T + \text{TIME}(c^n, T) \), this concludes the proof.

\[\text{Lemma 5.54 (Soundness of DLLOCK):} \]

If the valid concrete configurations (cf. Definition 4.4), abstract configurations and lock

\[
c^0 @ \langle [T, pc^0_T, \tilde{n}^0_T, \tilde{a}^0_T] \rangle_{T \in \text{Thrd}} \in \text{Conf},
\]

\[
c^m @ \langle [T, pc^m_T, \tilde{n}^m_T, \tilde{a}^m_T] \rangle_{T \in \text{Thrd}} \in \text{Conf},
\]

\[
c^n @ \langle [T, pc^n_T, \tilde{n}^n_T, \tilde{a}^n_T] \rangle_{T \in \text{Thrd}} \in \text{Conf},
\]

\[
c^i @ \langle [T, pc^i_T, \tilde{n}^i_T, \tilde{a}^i_T] \rangle_{T \in \text{Thrd}_i} \in \text{Conf},
\]

\[
c^j @ \langle [T, pc^j_T, \tilde{n}^j_T, \tilde{a}^j_T] \rangle_{T \in \text{Thrd}_j} \in \text{Conf}, \quad \text{and}
\]

\[
lck \in \text{Lck},
\]

are such that

\[
0 \leq m \leq n,
\]

\[
c^0 \rightarrow_{prg} \ldots \rightarrow_{prg} c^m \rightarrow_{prg} \ldots \rightarrow_{prg} c^n,
\]

\[
0 \leq j,
\]

\[
\text{Thrd}_i \subseteq \text{Thrd}_j \subseteq \text{Thrd},
\]

\[
\forall i \in \{m, \ldots, n\} : \text{OWN}(\tilde{n}^i_lck) = \perp_{\text{thrd}},
\]

\[
\text{REL}(\tilde{n}^i_lck) \in \gamma (\text{REL}(\tilde{n}^i_lck)),
\]

\[
\exists T \in \text{Thrd}_i : \text{STM}(T, pc^0_T) = [\text{lock lck}]^\gamma (\tilde{n}^0_T) \land \tilde{a}^0_T = \tilde{a}^0_T \land
\]

\[
\forall i \in \{0, \ldots, n\} : \text{OWN}(\tilde{n}^i_lck) \neq T,
\]

where \( \text{Thrd}^i_{\text{exe}} \) is as defined in Table 4.3 for \( c^i \), then DLLOCK satisfies:

\[
\min (\{t^n_T + \text{TIME}(c^n, T) \mid T \in \text{Thrd}\}) \in \gamma (\text{DLLOCK}(c^i, lck))
\]
PROOF. Assume that the valid concrete configurations (cf. Definition 4.4), abstract configurations and lock

\[ \begin{align*}
    c^0 & \in \text{Conf}, \\
    c^m & \in \text{Conf}, \\
    c^n & \in \text{Conf}, \\
    \bar{c}^0 & \in \text{Conf}, \\
    \bar{c}^j & \in \text{Conf},
\end{align*} \]

are such that

\[\begin{align*}
    & 0 \leq m \leq n, \\
    & c^0 \xrightarrow{\text{prg}} \ldots \xrightarrow{\text{prg}} c^m \xrightarrow{\text{prg}} \ldots \xrightarrow{\text{prg}} c^n, \\
    & 0 \leq j, \\
    & \text{Thrd}^j \subseteq \text{Thrd}_{\text{exc}} \subseteq \text{Thrd}, \\
    & \forall i \in \{m, \ldots, n\} : \text{OWN}(lck) = \bot_t, \\
    & \text{REL}(lck) \in \gamma(t) \in \text{REL}(lck), \\
    & \exists T \in \text{Thrd}^j : \text{STM}(T, c^j) \neq [lck]^{pc_T}.
\end{align*}\]

where \(\text{Thrd}^j\) is as defined in Table 4.3 for \(c^j\). First note that:

- Since \(\forall i \in \{m, \ldots, n\} : \text{OWN}(lck) = \bot_t\) and \(\text{REL}(lck) = \text{REL}(lck)\) (Tables 4.2 and 4.3).
- Since \(\forall i \in \{m, \ldots, n\} : \text{OWN}(lck) = \bot_t\) and \(\text{REL}(lck) = \text{REL}(lck)\) (Tables 4.2 and 4.3).
- Since time only moves forward (Lemma 4.2), it must be that \(\forall T \in \text{Thrd}^j : \text{REL}(lck) \leq \text{TIME}(c^n, T)\).
- Since \(\forall i \in \{m, \ldots, n\} : \text{OWN}(lck) = \bot_t\) and \(\text{REL}(lck) = \text{REL}(lck)\) (Tables 4.2 and 4.3).
• Since \(\exists T \in \text{Thrd}_{\ell j} : (\text{STM}(T, pc_{c}^{0}) = [\text{Lock lck}]pc_{c}^{0} \land t_{T}^{\ell j} = \bar{t}_{T}^{0} \land pc_{c}^{\bar{t}_{T}^{0}} = pc_{c}^{0} = pc_{c}^{0} \land T \in \text{Thrd}_{\text{exe}} \land \forall i \in \{0, \ldots, n\} : \text{OWN}(\bar{t}_{i}^{\ell j} \text{ lck}) \neq T\), \(\text{Thrd}_{\ell j} \subseteq \text{Thrd}\), \(\forall T \in \text{Thrd} : t_{T}^{a_{n}} \leq \text{REL}(\bar{t}_{n}^{\ell j} \text{ lck})\), \(\forall T \in \text{Thrd} : \text{REL}(\bar{t}_{n}^{\ell j} \text{ lck}) \leq t_{T}^{a_{n}} + \text{TIME}(c^{a_{n}}, T)\) and \(\forall T \in \text{Thrd} : (\text{STM}(T, pc_{c}^{0}) = [\text{Lock lck}]pc_{c}^{0} \Rightarrow t_{T}^{a_{n}} = t_{T}^{a_{n}})\) it must be that \(\exists T \in \text{Thrd}_{\ell j} : t_{T}^{a_{n}} \leq \text{REL}(\bar{t}_{n}^{\ell j} \text{ lck}) \leq t_{T}^{a_{n}} + \text{TIME}(c^{a_{n}}, T)\) and \(\text{STM}(T, pc_{c}^{0}) = [\text{Lock lck}]pc_{c}^{0} \land t_{T}^{a_{n}} \in \gamma_{t}(\bar{t}_{T}^{0}) \land \bar{t}_{i}^{a_{n}} = \bar{t}_{i}^{0} \land pc_{\bar{t}_{i}^{0}} = pc_{\bar{t}_{i}^{0}} = pc_{\bar{t}_{i}^{0}} = pc_{\bar{t}_{i}^{0}} \land T \in \text{Thrd}_{\text{exe}} \land \forall i \in \{0, \ldots, n\} : \text{OWN}(\bar{t}_{i}^{\ell j} \text{ lck}) \neq T\).

From here on, it will be assumed that \(T' \in \text{Thrd}_{\ell j}\) is one of the threads such that \(\text{STM}(T', pc_{c}^{0}) = [\text{Lock lck}]pc_{c}^{0} \land t_{T'}^{a_{n}} \leq \text{REL}(\bar{t}_{n}^{\ell j} \text{ lck}) \leq t_{T'}^{a_{n}} + \text{TIME}(c^{a_{n}}, T')\) and \(t_{T'}^{a_{n}} \in \gamma_{t}(\bar{t}_{T'}^{0}) \land \bar{t}_{i}^{a_{n}} = \bar{t}_{i}^{0} \land pc_{\bar{t}_{i}^{0}} = pc_{\bar{t}_{i}^{0}} = pc_{\bar{t}_{i}^{0}} = pc_{\bar{t}_{i}^{0}} \land T' \in \text{Thrd}_{\text{exe}} \land \forall i \in \{0, \ldots, n\} : \text{OWN}(\bar{t}_{i}^{\ell j} \text{ lck}) \neq T'\).

• Let \(\{m_{1}, \ldots, m_{2}\}\) be the set of indices, such that \(0 \leq m_{1} \leq m_{2} \leq m, \forall i \in \{m_{1}, \ldots, m_{2}\} : T' \in \text{Thrd}_{\ell j}^{i}\) and \(\forall i \in \{0, \ldots, m\} \setminus \{m_{1}, \ldots, m_{2}\} : T' \notin \text{Thrd}_{\ell j}^{i}\), where \(\text{Thrd}_{\ell j}^{i}\) is as defined in Table 4.3 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 \(\text{Thrd}_{\ell j} \subseteq \text{Thrd}\), \(t_{T}^{a_{n}} \in \gamma_{t}(\bar{t}_{T}^{0})\), \(\bar{t}_{T}^{a_{n}} = \bar{t}_{T}^{0}\), \(c^{0} \xrightarrow{pc}_{pr} \cdots \xrightarrow{pc}_{pr} c^{m} \), \(t_{T'}^{a_{n}} \leq \text{REL}(\bar{t}_{n}^{\ell j} \text{ lck}) \leq t_{T'}^{a_{n}} + \text{TIME}(c^{a_{n}}, T')\), \(\text{REL}(\bar{t}_{n}^{\ell j} \text{ lck}) \in \gamma_{t}(\text{REL}(\bar{t}_{n}^{\ell j} \text{ lck}))\) and \(0 \leq m\), it is easy to see that every configuration, \(\bar{c}^{i}\), created by the repeat-loop 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 \(|\{m_{1}, \ldots, m_{2}\}| + 1\) times (given that \(0 \notin \text{ABSTIM}(c^{i}, T')\)) and for each of the iterations, the derived \(c^{i}\) safely approximates the corresponding concrete configuration (since in the concrete case, \(t_{T'}^{a_{n}} \leq \text{REL}(\bar{t}_{n}^{\ell j} \text{ lck})\) for \(i \in \{m_{1}, \ldots, m_{2}\}\), the abstract execution time can safely be trimmed as done on line 11).

For the sake of readability, let

\[
\begin{align*}
\bar{t}_{T}^{a_{n}} &= \bar{t}_{T}^{a_{n}} \text{ at the repeat-loop exit, and} \\
\bar{c}^{i} &= \bar{c}^{i} \text{ at the repeat-loop exit.}
\end{align*}
\]

Assuming that \(\bar{t}_{d_{i}}\) is safe at the start of each iteration of the for all \(T \in \text{Thrd}_{\ell j}\)-loop, where \(T\) is such that \(\text{STM}(T, pc_{c}^{0}) = [\text{Lock lck}]pc_{c}^{0}\) (cf. \(T'\)), it
should be shown that \( \min(\{t'^{\mu} + \text{TIME}(c', T) \mid T \in \text{Thrd}_T \cap \text{STM}(T, pc_T) = \{\text{lock lck}\}) \in \gamma_T(\bar{t}_d) \) is always fulfilled at the end of each loop iteration.

It is easy to see that the initial value of \( \bar{t}_d \) (i.e. \( \bar{t}_0 \)) is trivially safe since \( \forall T \in \text{Thrd}_T : t'^{\mu} + \text{TIME}(c', T) \in \gamma_T(\bar{t}_0) \). It is also easy to see that for any thread, \( T \in \text{Thrd}_T \), such that \( \text{STM}(T, pc_T) \neq \{\text{lock lck}\} \), \( \bar{t}_d \) 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 \( \bar{t}_d \) is \( \alpha_i(\{\infty\}) \) for each considered thread and only the \( \sqcup \) operator, where \( \bar{t}_d \) itself is one of the arguments, is used to change the value of \( \bar{t}_d \), it must be that after the \textbf{repeat}-loop, \( \infty \in \gamma_T(\bar{t}_d) \).

- The \textbf{repeat}-loop terminates for \( T' \) given that \( \text{REL}(\bar{t}, \text{lock}) \) is not infinite since it is either terminated if \( 0 \in \gamma_T(\text{ABSTIME}(c', T')) \) or \( \text{REL}(\bar{t}, \text{lock}) \preceq_T \text{ABSTIME}(c', T') \) (cf. Assumption 5.51), and thus \( \langle \bar{t}_d \bar{t}, \text{ABSTIME}(c', T') \rangle \cap (\text{REL}(\bar{t}, \text{lock}) \sqcup \alpha_i(\{\infty\})) = \bot ; \) i.e. in the latter case, it is safely determined that enough iterations have been considered.

- Since \( \infty \in \gamma_T(\bar{t}_d) \), and \( t'^{\mu} \) (where \( i \in \{m_1, \ldots, m_2\} \) as defined above) is safely approximated, in each iteration of the \textbf{repeat}-loop, it must be that \( \bar{t}_d \sqcup \alpha_i(\bar{t}_d, \text{ABSTIME}(c', T')) \) (occurring in the next iteration of the loop) is a safe approximation of \( t'^{\mu} + \text{TIME}(c', T') \); i.e. after the loop exits, \( \bar{t}'_d \) is a safe approximation of when \( T' \) acquires \text{lock} unless the loop terminated prematurely due to \( 0 \in \text{ABSTIME}(c', T') \) since the loop then iterates at least \( \{m_1, \ldots, m_2\} \) times. However, note that if the loop exits and \( 0 \in \text{ABSTIME}(c', T') \) but \( \bar{t}'_d \neq \bot \), it is actually the case that \( \text{REL}(\bar{t}, \text{lock}) \preceq_T \bar{t}'_d \) and \( \bar{t}_d \) is thus safely approximating the point in time when \( T' \) acquires \text{lock} since the loop must have iterated at least \( \{m_1, \ldots, m_2\} \) 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 \( \text{TIME}(c', T') = 0 \), then \( t'^{\mu} + \text{TIME}(c', T') = t'^{\mu} \) (remember that \( t'^{\mu} = t'^{\mu} \)) and \( 0 \in \text{ABSTIME}(c', T') \) (Assumption 5.51). Since \( 0 \in \gamma_T(\text{ABSTIME}(c', T')) \), the \textbf{repeat}-loop might exit after too few iterations. If the loop exits after too few iterations, it must be that \( \text{REL}(\bar{t}, \text{lock}) \preceq_T \bar{t}'_d \), and thus \( \bar{t}'_d \neq \bot \). However, \( \text{REL}(\bar{t}, \text{lock}) \) provides safe information for when \( T' \) would acquire \text{lock} since
Two (or rather, three) cases must now be considered for $T$ that should be shown that $\min\{\ldots\}$.

Since the initial value of $\tilde{\tau}$ is safely approximated, in each iteration of the repeat-loop, then $\tilde{\tau}' = (\tilde{\tau}' + \tilde{\tau}\tau)$ and $\tilde{\tau} = (\tilde{\tau}' + \tilde{\tau}) R\tilde{\tau} \tilde{\tau}$ $\tilde{\tau}$ be that $\tilde{\tau}'$ is safely approximated, in each iteration of the repeat-loop.

If the loop exits after too few iterations, it must be that $\tilde{\tau}'$ provides safe information for when $T$ $\tilde{\tau}'$ is safely approximated, in each iteration of the repeat-loop.

Thus, it must be that:

$$\min\{\{t_{\tilde{\tau}}'' + \text{TIME}(c, T) \mid T \in \text{Thrd}_\tilde{\tau} \land \text{STM}(T, p c_\tilde{\tau}) = [\text{lock} lck[p c_\tilde{\tau}]\} \} \in \gamma(\text{DLLOCK}(\tilde{\tau}, lck))$$

The accumulated time, $\tilde{\tau}'$, for a thread, $T \in \text{Thrd}_\tilde{\tau}$, is determined using $\text{ACCTIME}$, defined in Algorithm 5.12 on page 107, which is partially a safe approximation of the concrete accumulated time of $T$ (cf. Lemma 5.55). This is because the way that time accumulates for threads executing $\text{lock} lck$ for some lock, $lck \in \text{Lck}$, that is currently assigned to some other thread differs in the concrete and abstract semantics, as previously discussed.

In the concrete semantics, the $\text{lock}$-statement is just considered to finish its execution, without successfully acquiring $lck$, after the (relative) time given by $\text{TIME}$, then a new instance of the same $\text{lock}$-statement is executed (cf. Tables 4.2 and 4.3); i.e. the thread is actively spinning on the lock. However, in the abstract semantics (cf. Tables 5.10 and 5.11 and Algorithms 5.11 and 5.12), a thread issuing $\text{lock} lck$ 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).
If the lock, $lck \in Lck$, is not currently assigned to some other thread when some thread issues $lock lck$, the behavior is the same in both the concrete and abstract semantics in case the $lock$-issuing thread successfully acquires $lck$ (Lemma 5.55); i.e. the thread’s execution time will be accumulated based on $TIME$ and $ABSTIME$, respectively.

**NOTE.** $ACCTIME$ is not directly safe for the case that $STM(T, pc_T) = [lock lck]^{pc_T} \land OWN(\{lck\}^\prime) \neq T$. 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.

$ACCTIME$ is also not directly safe for the case that $T$ has been waiting but is now assigned $lck$ and $T^a \uparrow_t, ABSTIME(\gamma, T) \leq_t REL(lck)$ since this would generate an extra abstract configuration for which $T^a <_t REL(lck)$ and $T^a \uparrow_t, ABSTIME(\gamma, T) \not<_t REL(lck)$; 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 5.58.

**Lemma 5.55 (Partial soundness of $ACCTIME$):**

If the valid concrete configuration $c @ (T', pc_{T'}, x_{T'}, t^a_{T'}) \in \mathsf{Conf}$ (cf. Definition 4.4), the abstract configuration $c^0 @ (T', pc_{T'}, x_{T'}, t^a_{T'}) \in \mathsf{Conf}$, and some thread, $T \in \mathsf{Thrds}_c$, are such that

$$\mathsf{Thrds}_c \subseteq \mathsf{Thrds} \land$$
$$pc_T = pc^c_T \land$$
$$t^a_T \in \gamma^c_t(I^a_T) \land$$
$$((T \in \mathsf{Thrds}^{exe}_c \land \forall lck \in Lck : (STM(T, pc_T) = [lock lck]^{pc_T} \Rightarrow$$
$$\text{OWN(} \{lck\}^\prime) = T \land \text{OWN(} \{lck\}^\prime) = T)) \iff T \in \mathsf{Thrds}^{exe}_c) \land$$
$$\forall lck \in Lck : (\text{OWN(} \{lck\}^\prime) = T \Rightarrow (\text{OWN(} \{lck\}^\prime) = \text{OWN(} \{lck\}^\prime) \land$$
$$DL(\{lck\}^\prime) \in \gamma^c_t(\text{DL}(\{lck\}^\prime)) \land$$
$$\text{POWN}(\{lck\}^\prime) = \text{POWN}(\{lck\}^\prime) \land$$
$$\text{REL}(\{lck\}^\prime) \in \gamma^c_t(\text{REL}(\{lck\}^\prime)) \land$$
$$\text{min}(\gamma^c_t(\text{DL}(\{lck\}^\prime))) = -\infty)),$$

where $\mathsf{Thrds}^{exe}_c$ and $\{lck\}^\prime$, and $\mathsf{Thrds}^{exe}_c$ and $\{lck\}^\prime$, are as defined in Tables 4.3 and
5.11, respectively, then
\[ t_1'' \in \gamma_i(\text{ACCTIME}([T', pc_T', \bar{e}_T', \bar{r}_T'], T' \in \text{Thrd}_c, \bar{x}, \bar{y}'', \text{Thrd}'_{exe}, T)) \]
where \( t_1'' \) is as defined in Table 4.3.

\[ \square \]

**PROOF.** Assume that the (valid; cf. Definition 4.4) configurations \( c @ \langle [T', pc_T', \bar{e}_T', \bar{r}_T'] \rangle \in \text{Conf} \) and \( \bar{e} @ \langle [T', pc_T', \bar{e}_T', \bar{r}_T'] \rangle \in \text{Thrd}_c, \bar{x}, \bar{y}'', \bar{z} \rangle \in \text{Conf} \) and the thread \( T \in \text{Thrd}_c \) are such that

\[ \text{Thrd}_c \subseteq \text{Thrd} \land \]
\[ pc_T = pc_T' \land \]
\[ t_1'' \in \gamma_i(\bar{r}_T') \land \]
\[ ((T \in \text{Thrd}_{exe} \land \forall lck \in Lck : \text{STM}(T, pc_T) = [\text{lock lck}]_{pc_T} \Rightarrow (\text{OWN}(\bar{y}'', lck) = T \land \text{OWN}(\bar{y}'', lck) = T)) \leftrightarrow T \in \text{Thrd}_{exe}^{lck}) \land \]
\[ \forall lck \in Lck : (\text{OWN}(\bar{y}'', lck) = T \Rightarrow (\text{OWN}(\bar{y}'', lck) = T \land \text{DL}(\bar{y}'', lck) \in \gamma_i(\text{DL}(\bar{y}'', lck)) \land \text{POWN}(\bar{y}'', lck) = \text{POWN}(\bar{y}'', lck) \land \text{REL}(\bar{y}'', lck) \in \gamma_i(\text{REL}(\bar{y}'', lck)) \land \text{min}(\gamma_i(\text{DL}(\bar{y}'', lck)))) = -\infty)) \]

where \( \text{Thrd}_{exe}^{lck} \) and \( \bar{y}'', \bar{y}'', \bar{z} \), and \( \text{Thrd}_{exe}^{lck} \) and \( \bar{y}'', \bar{y}'', \bar{z} \), are as defined in Tables 4.3 and 5.11, respectively.

For the sake of readability, let \( \bar{c} = \langle [T', pc_T', \bar{e}_T', \bar{r}_T'] \rangle \in \text{Thrd}_c, \bar{x}, \bar{y}'', \rangle \) when considering the following cases. Note that \( \text{Time} = \text{Intv} \).

1. If \( T \not\in \text{Thrd}_{exe} \) (and thus, \( T \not\in \text{Thrd}_{exe}^{lck} \)), then \( t_1'' = t_1'' \) (Table 4.3) and \( \text{ACCTIME}(\bar{c}, \text{Thrd}_{exe}^{lck}, T) = t_1'' \). Thus, \( t_1'' \in \gamma_i(\text{ACCTIME}(\bar{c}, \text{Thrd}_{exe}^{lck}, T)) \).

2. If \( T \in \text{Thrd}_{exe} \) 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 \( lck \in Lck, \) \( \text{STM}(T, pc_T) \in \{[\text{skip}]_{pc_T}, [\text{if } b \text{ goto } l]_{pc_T}, [\text{store } r \text{ to } x]_{pc_T}, [\text{load } r \text{ from } x]_{pc_T}, [\text{unlock lck}]_{pc_T} \} \), i.e. \( \forall lck' \in Lck : \text{STM}(T, pc_T) \neq [\text{lock lck}]_{pc_T} \) (and thus, \( T \in \text{Thrd}_{exe} \) since \( (T \in \text{Thrd}_{exe} \land \forall lck' \in Lck : (\text{STM}(T, pc_T) = [\text{lock lck}]_{pc_T} \Rightarrow (\text{OWN}(\bar{y}'', lck) = T \land \text{OWN}(\bar{y}'', lck) = T)) \leftrightarrow T \in \text{Thrd}_{exe}^{lck}) \), then \( \text{ACCTIME}(\bar{c}, \text{Thrd}_{exe}^{lck}, T) = t_1'' + \text{ABSTIME}(\bar{c}, T) \) and by Table 4.3, \( t_1'' = t_1'' + \text{TIME}(\bar{c}, T) \). Thus, by Lemma 5.52, \( t_1'' \in \gamma_i(\text{ACCTIME}(\bar{c}, \text{Thrd}_{exe}^{lck}, T)) \).

3. If \( T \in \text{Thrd}_{exe} \) and for some \( lck \in Lck, \) \( \text{STM}(T, pc_T) = [\text{lock lck}]_{pc_T} \) and \( \text{OWN}(\bar{y}'', lck) = T, \) and thus, \( T \in \text{Thrd}_{exe} \land \)
∀lck′ ∈ Lck : (STM(T,pc_T) = [lock lck′]pc_T ⇒ (OWN(\[\tilde{lck}''\] lck′) = T ∧ OWN(\[\tilde{lck}''\] lck′) = T)))) \iff T ∈ Thrde_{exe}, then several cases need to be considered. Note that min(γ_{\tilde{T}}(DL(\[\tilde{lck}''\] lck′))) = -∞, since ∀lck′ ∈ Lck : (OWN(\[\tilde{lck}''\] lck′) = T ⇒ min(γ_{\tilde{T}}(DL(\[\tilde{lck}''\] lck′))) = -∞) and OWN(\[\tilde{lck}''\] lck′) = T, and that T cannot acquire lck at any time, \tilde{t}, such that \tilde{t} < t, REL(\[\tilde{lck}''\] lck′), since lck has not been released at \tilde{t}, or DL(\[\tilde{lck}''\] lck′) < t, since by then some other thread would have taken lck (cf. Tables 5.10 and 5.11).

(a) If STT(\[\tilde{lck}''\] lck′) = locked (and STT(\[\tilde{lck}''\] lck′) = locked since c is valid and OWN(\[\tilde{lck}''\] lck′) \neq ⊥_thr), then \(t_{\tilde{T}}'' = t_{\tilde{T}}' + \text{TIME}(c,T)\) and \(\text{ACCTIME}(\tilde{c},\text{Thrd}_{exe},T) = t_{\tilde{T}}^\gamma + t, \text{ABSTIME}(\tilde{c},T)\). Thus, \(t_{\tilde{T}}' \in γ_1(\text{ACCTIME}(\tilde{c},\text{Thrd}_{exe},T))\) (Lemma 5.52).

(b) Assume that STT(\[\tilde{lck}''\] lck′) = unlocked ∧ DL(\[\tilde{lck}''\] lck′) ≤ t_{\tilde{T}}' (\(t_{\tilde{T}}^\gamma + t, \text{ABSTIME}(\tilde{c},T)\)). Then, in the concrete case, it must be that T cannot be the thread acquiring lck since DL(\[\tilde{lck}''\] lck′) ∈ γ_{\tilde{T}}(DL(\[\tilde{lck}''\] lck′)), \(t_{\tilde{T}}'' \in γ_1(t_{\tilde{T}}^\gamma), \text{TIME}(c,T) \in γ_1(\text{ABSTIME}(\tilde{c},T))\) and \(t_{\tilde{T}}'' + \text{TIME}(c,T) = DL(\[\tilde{lck}''\] lck′)\) whenever T acquires lck (Tables 4.2 and 4.3). But, then it cannot be that STT(\[\tilde{lck}''\] lck′) = unlocked ∧ DL(\[\tilde{lck}''\] lck′) ≤ t_{\tilde{T}}' (\(t_{\tilde{T}}^\gamma + t, \text{ABSTIME}(\tilde{c},T)\)) since in the concrete case, T does successfully acquire lck, which means that 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 5.10 and 5.11.)

(c) Note that the STT(\[\tilde{lck}''\] lck′) = unlocked ∧ DL(\[\tilde{lck}''\] lck′) ≤ t_{\tilde{T}}' (\(t_{\tilde{T}}^\gamma + t, \text{ABSTIME}(\tilde{c},T)\)) \(\land (\tilde{t}_{\tilde{T}}^\gamma + t, \text{ABSTIME}(\tilde{c},T)) \leq t, \text{REL}(\[\tilde{lck}''\] lck′)\) conditioned branch, which applies to cases where T has been frozen for sure while waiting to acquire lck but has now been assigned lck, cannot be taken either. To see this, note that since c is valid, it must be that REL(\[\tilde{lck}''\] lck′) ≤ t_{\tilde{T}}'' + \text{TIME}(c,T) (Definition 4.4). Then, since t_{\tilde{T}}'' ∈ γ_{\tilde{T}}(t_{\tilde{T}}^\gamma), \text{TIME}(c,T) \in γ_1(\text{ABSTIME}(\tilde{c}^0,T))\) (cf. Assumption 5.51), OWN(\[\tilde{lck}''\] lck′) = T and OWN(\[\tilde{lck}''\] lck′) = T ⇒ REL(\[\tilde{lck}''\] lck′) ∈ γ_1(REL(\[\tilde{lck}''\] lck′)), it must be that t_{\tilde{T}}'' + t, \text{ABSTIME}(\tilde{c}^0,T) \leq t, \text{REL}(\[\tilde{lck}''\] 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 STT(\[\tilde{lck}''\] lck′) = unlocked ∧ DL(\[\tilde{lck}''\] lck′) ≤ t_{\tilde{T}}' (\(t_{\tilde{T}}^\gamma + t, \text{ABSTIME}(\tilde{c},T)\)) \(\land (\tilde{t}_{\tilde{T}}^\gamma + t, \text{ABSTIME}(\tilde{c},T)) \leq t, \text{REL}(\[\tilde{lck}''\] lck′) \land (\text{OWN}(\[\tilde{lck}''\] lck′) = T \lor \text{REL}(\[\tilde{lck}''\] lck′)) \leq t, (\(\tilde{t}_{\tilde{T}}^\gamma + t, \text{ABSTIME}(\tilde{c},T)\))), then two cases must be considered.
5.8 Abstract Semantics

i. If \( \text{POWN}(\bar{\mu} lck) = T \), then the sequential execution of the statements of a thread (cf. Tables 4.2 and 4.3) gives that \( T \) must acquire \( lck \) at \( \bar{t}_T^{\alpha} \vdash_t ABSTIME(c, T) \), but not at a point in time, \( \bar{t} \), such that \( DL(\bar{\mu} lck) \preceq_{\bar{t}} \bar{t} \). Because by then, some other thread must have already acquired \( lck \) (since \( DL(\bar{\mu} lck) \in \gamma_t(DL(\bar{\mu} lck)) \)). Thus, it must be that \( t_T^{\alpha} \in \gamma_t((\bar{t}_T^{\alpha} \vdash_t ABSTIME(c, T)) \cap_t DL(\bar{\mu} lck)) \).

ii. If \( REL(\bar{\mu} lck) \preceq_{\bar{t}} (\bar{t}_T^{\alpha} \vdash_t ABSTIME(c, T)) \), then Lemma 5.52 gives that \( t_T^{\alpha} \in \gamma_t((\bar{t}_T^{\alpha} \vdash_t ABSTIME(c, T)) \cap_t DL(\bar{\mu} lck)) \) since \( t_T^{\alpha} \in DL(\bar{\mu} lck) \) (cf. Tables 4.2 and 4.3), \( DL(\bar{\mu} lck) \in \gamma_t(DL(\bar{\mu} lck)) \) and \( REL(\bar{\mu} lck) \in \gamma_t(REL(\bar{\mu} lck)) \).

(e) If \( STT(\bar{\mu} lck) = \text{unlocked} \land DL(\bar{\mu} lck) \nsubseteq_{\bar{t}} (\bar{t}_T^{\alpha} \vdash_t ABSTIME(c, T)) \cap_t REL(\bar{\mu} lck) = \top \land T \neq \text{POWN}(\bar{\mu} lck) \), then let \( t_T^{\alpha} = \bar{t}_2 \), \( ABSTIME(c, T) \), which is obviously a safe approximation of the first point in time at which \( T \) can acquire \( lck \). Also let \( c' \) be any configuration derived before (i.e. \( c' = \bar{c} \)) or inside the \textbf{repeat}-loop. Note that \( \bar{t}_2 = \bar{t}_T \) is used to exit the loop in case \( DL(\bar{\mu} lck) \preceq_{\bar{t}_T} (\bar{t}_T^{\alpha} \vdash_t ABSTIME(c', T)) \) or \( 0 \in \gamma_t(ABSTIME(c', T)) \), where the latter case means that a \( t_T^{\alpha} \) such that \( REL(\bar{\mu} lck) \preceq_{\bar{t}} \bar{t}_T^{\alpha} \) cannot be derived.

i. If \( DL(\bar{\mu} lck) \preceq_{\bar{t}} \bar{t}_T^{\alpha} \vdash_t ABSTIME(c', T) \), then it must be that at \( \bar{t}_T^{\alpha} \vdash_t ABSTIME(c', T) \), some other thread will have acquired \( lck \) (hence, \( \bar{t}_T^{\alpha} \) is the last point in time when \( T \) can acquire \( lck \)). Thus, it must be that \( t_T^{\alpha} \in \gamma_t((\bar{t}_T^{\alpha} \vdash_t ABSTIME(c', T)) \cap_t DL(\bar{\mu} lck) \cap_t (REL(\bar{\mu} lck) \cup \alpha_t(\{\infty\})) \) since \( DL(\bar{\mu} lck) \in \gamma_t(DL(\bar{\mu} lck)) \) and \( REL(\bar{\mu} lck) \in \gamma_t(REL(\bar{\mu} lck)) \).

ii. If \( 0 \in \gamma_t(ABSTIME(c', T)) \) and also \( DL(\bar{\mu} lck) \nsubseteq_{\bar{t}} \bar{t}_T^{\alpha} \vdash_t ABSTIME(c', T) \), then it must be that \( \bar{t}_T^{\alpha} \vdash_t ABSTIME(c', T) \), where \( \bar{t} = (\bar{t}_T^{\alpha} \cup \alpha_t(\{\infty\})) \cap_t REL(lck) \) and \( c'' = \langle [T', p, c_T', \bar{x}_{T'}, (T = T' ? \bar{t} : t_T^{\alpha})] \rangle_{T' \in \text{Thrd}_c} \bar{x}, \bar{\mu}'' \rangle \), is a safe approximation of the last point in time when \( T \) can (or rather, will) acquire \( lck \) (cf. Assumption 5.51) since \( REL(\bar{\mu} lck) \in \gamma_t(REL(\bar{\mu} lck)) \). Thus, it must be that \( t_T^{\alpha} \in \gamma_t((\bar{t}_T^{\alpha} \vdash_t ABSTIME(c', T)) \cap_t DL(\bar{\mu} lck) \cap_t (REL(\bar{\mu} lck) \cup \alpha_t(\{\infty\})) \) since \( REL(\bar{\mu} lck) \in \gamma_t(REL(\bar{\mu} lck)) \) and \( DL(\bar{\mu} lck) \in \gamma_t(DL(\bar{\mu} lck)) \).

iii. If \( 0 \notin \gamma_t(ABSTIME(c', T)) \) and also \( DL(\bar{\mu} lck) \nsubseteq_{\bar{t}} \bar{t}_T^{\alpha} \vdash_t ABSTIME(c', T) \), then it must be that, at some point,
\textit{REL}(\tilde{\Pi}'' lck) \preceq_T \tilde{\eta}_{\text{prg}}. \quad \text{Since } \text{REL}(\tilde{\Pi}'' lck) \in \gamma_t(\text{REL}(\tilde{\Pi}'' lck)) \quad \text{and } \text{DL}(\tilde{\Pi}'' lck) \in \gamma_t(\text{DL}(\tilde{\Pi}'' lck)), \text{ it is thus easy to see that } \tilde{\eta}_{\text{prg}} \in \gamma_t((\tilde{\Pi}'' lck) \cap_t \tilde{\eta}_{\text{prg}}) \cap_t \text{DL}(\tilde{\Pi}'' lck) \cap_t (\text{REL}(\tilde{\Pi}'' lck) \cap_t \alpha_t(\{\infty\})). \quad \square

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 \textit{Time} = \textit{Intv}. Since \textit{Time} = \textit{Intv}, \textit{Thrd}_{\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, \textit{lock}, even though \textit{lock} \in \textit{Lck}, in the concrete case, some other thread could issue its corresponding \textit{lock} \textit{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, \textit{lck} \in \textit{Lck}, given a reference thread, \textit{T} \in \textit{Thrd}, in a lock state, \tilde{\Pi}, resulting from a transition using \xrightarrow{\text{prg}} are by definition as follows (cf. Tables 5.10 and 5.11).

1. \textit{OWN}(\tilde{\Pi} lck) \not\in \{\bot_{\text{thrd}}, T\} – This means that \textit{T} will be frozen if it issues \textit{lock} \textit{lck} and occurs when \textit{OWN}(\textit{lock}) \neq \textit{T}.

2. \textit{OWN}(\tilde{\Pi} lck) = \bot_{\text{thrd}} – This occurs when \textit{OWN}(\textit{lock}) = \bot_{\text{thrd}}. A safe (over-approximate) owner assignment will occur if \textit{T} issues \textit{lock} \textit{lck}. The soundness is given by that it is trivially the case that for all concrete and abstract configurations consisting of the threads in \textit{Thrd}, \{\textit{T}' \in \textit{Thrd}_{\text{exe}} \mid \text{STM}(\textit{T}', \textit{pc}_{\textit{T}'}) = [\textit{lock} \textit{lck}]_{\textit{pc}_{\textit{T}'}}, \exists \textit{l} \in \textit{Lbl}_{\textit{T}'} \cdot \text{STM}(\textit{T}', \textit{l}) = [\textit{lock} \textit{lck}]_{\textit{l}}\}; \text{ cf. Table 4.3.}

3. \textit{OWN}(\tilde{\Pi} lck) = T \wedge \text{STM}(\tilde{\Pi} lck) = \text{'unlocked'} – This means that \textit{T} has not yet done \textit{lock} \textit{lck}, but some other thread has (with the result that \textit{T} was assigned \textit{lck}; cf. the discussion for state 2). If \textit{T} issues \textit{lock} \textit{lck} within the deadline, it will successfully acquire \textit{lck}. If it does not, there is no corresponding concrete situation described by the owner assignment, given that \textit{DL}(\tilde{\Pi} lck) \in \textit{DL}(\tilde{\Pi} lck), and thus, the configuration will be discontinued; cf. Algorithms 6.1 and 6.10, which are discussed in Chapter 6. This occurs when \textit{OWN}(\tilde{\Pi} lck) = \bot_{\text{thrd}}.
4. OWN(\(\tilde{lck}\)) = T \land STT(\(\tilde{lck}\)) = locked – This occurs when OWN(\(\tilde{lck}\)) = T.

The possible transitions between these abstract states (as defined by \(\xrightarrow{\text{ax}}\) and \(\xrightarrow{\text{pg}}\)) are depicted in Figure 5.12. State 3 (a result from the over-approximate owner assignment performed by \(\xrightarrow{\text{pg}}\)) 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 \(\xrightarrow{\text{pg}}\) 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

\[c^0 @ \langle [T, pc^0_T, \tau^0_T, t^0_T] \rangle_{Thrд}, \Xi^0, \Pi^0 \rangle \in \text{Conf},\]
\[c^l @ \langle [T, pc^l_T, \tau^l_T, t^l_T] \rangle_{Thrд}, \Xi^l, \Pi^l \rangle \in \text{Conf},\]
\[c^n @ \langle [T, pc^n_T, \tau^n_T, t^n_T] \rangle_{Thrд}, \Xi^n, \Pi^n \rangle \in \text{Conf},\]
\[c^0 @ \langle [T, pc^0_T, \tilde{z}^0_T, \tilde{t}^0_T] \rangle_{Thrд}, \tilde{\Xi}^0, \tilde{\Pi}^0 \rangle \in \text{Conf},\]
\[\tilde{c}^l @ \langle [T, pc^l_T, \tilde{z}^l_T, \tilde{t}^l_T] \rangle_{Thrд}, \tilde{\Xi}^l, \tilde{\Pi}^l \rangle \in \text{Conf},\]
\[\tilde{c}^k @ \langle [T, pc^k_T, \tilde{z}^k_T, \tilde{t}^k_T] \rangle_{Thrд}, \tilde{\Xi}^k, \tilde{\Pi}^k \rangle \in \text{Conf},\]
\[lck' \in \text{Lck},\]
\[T' \in \text{Thrд}_{lck} \text{ and}\]
\[T'' \in \text{Thrд}_{lck},\]
are such that

\[
\begin{align*}
0 &\leq i < n, \\
c^0 \xrightarrow{\text{prg}} \ldots \xrightarrow{\text{prg}} c^i \xrightarrow{\text{prg}} \ldots \xrightarrow{\text{prg}} c^n, \\
0 &\leq j < k, \\
\tilde{c}^0 \xrightarrow{\text{prg}} \ldots \xrightarrow{\text{prg}} \tilde{c}^j \xrightarrow{\text{prg}} \ldots \xrightarrow{\text{prg}} \tilde{c}^k,
\end{align*}
\]

\(\text{Thrd}^{\tilde{c}_k} \subseteq \text{Thrd}^{\tilde{c}_j} \subseteq \text{Thrd}^{\tilde{c}_0} \subseteq \text{Thrd},\)

\[
\text{STM}(T'', p\text{c}_T^{n''}) = [\text{lck}^i]^\text{prg}_{T''},
\]

\(T'' \in \text{Thrd}^{\tilde{c}_k}_{\text{exe}},\)

\[
\text{STM}(T', p\text{c}_T^{k'}) = [\text{lck}^j]^\text{prg}_{T'},
\]

\(T' \in \text{Thrd}^{\tilde{c}_k}_{\text{exe}},\)

\[
\forall h \in \{0, \ldots, k-1\} : (T' \in \text{Thrd}^{\tilde{c}_h}_{\text{exe}} \Rightarrow \text{STM}(T', p\text{c}_T^{h'}) = [\text{lck}^i]^\text{prg}_{T'}),
\]

\[
\text{rel}(\tilde{c}^i) \in \gamma _i(\text{rel}(\tilde{c}^i)),
\]

\[
(p\text{c}_T^i = p\text{c}_T^{k'} \land \\
t^{d)^i_T \in \gamma _i(\tilde{c}^{d)^i_T} ) \land \\
T' \in \text{Thrd}^{\tilde{c}_j}_{\text{exe}} \land \\
\text{own}(	ilde{c}^i)^I_lck' = \bot_{\text{thr}} \land \\
\text{own}([\tilde{c}^{i+1}]^I_lck') = T') \text{ and}
\]

\[
(p\text{c}_T^{n''} = p\text{c}_T^{k''} \land \\
t^{d)_{n''} \in \gamma _i(\tilde{c}^{d)_{n''}} ) \land \\
T'' \in \text{Thrd}^{\tilde{c}''}_{\text{exe}} \land \\
\text{OWN}([\tilde{c}^j]^{I_lck'}) = \bot_{\text{thr}} \land \\
\text{OWN}([\tilde{c}^{j+1}]^{I_lck'}) = T'),
\]

where the trace for \(T'\) in \(c^0 \xrightarrow{\text{prg}} \ldots \xrightarrow{\text{prg}} c^i\) is the same as in \(c^0 \xrightarrow{\text{prg}} \ldots \xrightarrow{\text{prg}} c^i\), the trace for \(T''\) in \(c^0 \xrightarrow{\text{prg}} \ldots \xrightarrow{\text{prg}} c^i\) is the same as in \(c^0 \xrightarrow{\text{prg}} \ldots \xrightarrow{\text{prg}} c^n\), \(\text{Thrd}^{\tilde{c}_k}_{\text{exe}}\) is as defined in Table 4.3, and \(\text{Thrd}^{\tilde{c}''}_{\text{exe}}\) and \(\text{Thrd}^{\tilde{c}_j}_{\text{exe}}\) are as defined in Table 5.11, then \(\tilde{c}^i\) satisfies:

\[
\begin{align*}
\text{OWN}([\tilde{c}^j]^{I_lck'}) = \text{OWN}([\tilde{c}^{j+1}]^{I_lck'}) = \text{OWN}([\tilde{c}^{j+1}]^{I_lck'}) = T' &\land \\
\text{STT}([\tilde{c}^j]^{I_lck'}) = \text{STT}([\tilde{c}^{j+1}]^{I_lck'}) = \text{STT}([\tilde{c}^{j+1}]^{I_lck'}) = \text{unlocked} &\land \\
\text{dl}([\tilde{c}^j]^{I_lck'}) = \text{dl}([\tilde{c}^{j+1}]^{I_lck'}) &\land \\
\text{min}(\gamma _i(\text{dl}([\tilde{c}^j]^{I_lck'}))) = -\infty &\land \\
t^{d)^{n''}_T + \text{time}(c', T') \in \gamma _i(\text{dl}([\tilde{c}^j]^{I_lck'})) &\land
\end{align*}
\]
PROOF. Assume that the valid concrete configurations, abstract configurations, lock and threads

\[
c^0 \rightarrow \ldots \rightarrow c^i \rightarrow \ldots \rightarrow c^n,
\]

\[
0 \leq i < n,
\]

\[
c^0 \rightarrow \ldots \rightarrow c' \rightarrow \ldots \rightarrow c^i.
\]

\[
0 \leq j < k,
\]

\[
c^0 \rightarrow \ldots \rightarrow c' \rightarrow \ldots \rightarrow c^k.
\]

\[\text{Thrd} \subseteq \text{Thrd} \subseteq \text{Thrd} \subseteq \text{Thrd} \subseteq \text{Thrd},\]

\[\text{STM}(T', pc^n_T) = [\text{lock} lck']^{pc^n_T},\]

\[T' \in \text{Thrd}_{exe},\]

\[\text{STM}(T', pc^k_T) = [\text{lock} lck']^{pc^k_T},\]

\[T' \in \text{Thrd}_{exe},\]

\[\forall h \in \{0, \ldots, k - 1\} : (T' \in \text{Thrd} \Rightarrow \text{STM}(T', pc^h_T) \neq [\text{lock} lck']^{pc^h_T}),\]

\[\text{REL}(\text{lock}) \in T' \in \mathbb{N} \text{ \(lck')\},}\]

\[\text{pc}^i_T = pc^i_T \land\]

\[\text{pc}^j_T = pc^j_T \land\]

\[\text{OWN}([i] lck') = \bot \land\]

\[\text{OWN}([i+1] lck') = T' \land\]

\[\text{pc}^i_T = pc^i_T \land\]

\[\text{pc}^j_T = pc^j_T \land\]

\[\text{OWN}([i] lck') = \bot \land\]

\[\text{OWN}([i+1] lck') = T',\]
where the trace for $T'$ in $c_0 \xrightarrow{prg} \ldots \xrightarrow{prg} c_k$ is the same as in $c_0 \xrightarrow{prg} \ldots \xrightarrow{prg} c_i$, the trace for $T''$ in $c_0 \xrightarrow{prg} \ldots \xrightarrow{prg} c_i$ is the same as in $c_0 \xrightarrow{prg} \ldots \xrightarrow{prg} c_n$. Thrd$^{ce}_ex$ is as defined in Table 4.3, and Thrd$^{ce}$ and Thrd$^{ce}_{ex}$ are as defined in Table 5.11.

First note that since the trace for $T'$ in $c_0 \xrightarrow{prg} \ldots \xrightarrow{prg} c_i$, the trace for $T''$ in $c_0 \xrightarrow{prg} \ldots \xrightarrow{prg} c_i$ is the same as in $c_0 \xrightarrow{prg} \ldots \xrightarrow{prg} c_n$, STM($T', pc^{ca}_{T''}$) = [lkck]$prg^{ca}_{T''}$, $T'' \in$ Thrd$^{ce}$

STM($T', pc^{ce}_{T'}$) = [lkck]$prg^{ce}_{T'}$, $T' \in$ Thrd$^{ce}$, $pc^{ce}_{T'} = pc^{ce}_{T''}$, $t^{ce}_{T'} \in \gamma_i(r^{ce}_{T''})$, $T' \in$ Thrd$^{ce}_ex$, OWN($\tilde{lck} k'$) = ⊥, OWN($\tilde{lck} k' + 1$) = $T'$, $pc^{ce}_{T''} = pc^{ce}_{T''}$, $t^{ca}_{T''} \in \gamma_i(r^{ca}_{T''})$, OWN($\tilde{lck} k$) = ⊥ and OWN($\tilde{lck} k + 1$) = $T'$, it must be that $T''$ acquires lck$ k'$ in the transition between $c'$ and $c' + 1$ and $T''$ wants to acquire lck$ k'$ in a transition from $c'$, while the abstract trace represents a situation (that can occur due to that Time = Intv) where $T''$ reaches the lkck$ k'$-statement (i.e. it reaches $pc^{ca}_{T''}$) before $T'$ (i.e. before $T''$ reaches $pc^{ca}_{T''}$), but lck$ k'$ is assigned to $T'$ as shown below.

Since $\forall h \in \{0, \ldots, k - 1\}$: ($T' \in$ Thrd$^{ce}_ex \Rightarrow$ STM($T', pc^{ca}_{T''}$) = [lkck]$prg^{ca}_{T''}$), OWN($\tilde{lck} k'$) = ⊥, OWN($\tilde{lck} k' + 1$) = ⊥ and OWN($\tilde{lck} k' + 1$) = $T'$, it is easy to see that OWN($\tilde{lck} k'$) = OWN($\tilde{lck} k' + 1$) = OWN($\tilde{lck} k' + 1$) = $T'$, STT($\tilde{lck} k'$) = STT($\tilde{lck} k' + 1$) = STT($\tilde{lck} k' + 1$) = unlocked and DL($\tilde{lck} k'$) = DL($\tilde{lck} k' + 1$) = DL($\tilde{lck} k' + 1$) (cf. Table 5.11).

Since DLLOCK is used to determine DL($\tilde{lck} k' + 1$) and DL($\tilde{lck} k' + 1$) = DL($\tilde{lck} k' + 1$), it is easy to see that min($\gamma_i$(DL($\tilde{lck} k'$))) = −∞ since DLLOCK is used only if $\exists T \in$ Thrd$^{ce}$ : STM($T, pc^{ce}_{T}$) = [lkck]$prg^{ce}_{T}$ (cf. Table 5.11) which is the case since $pc^{ca}_{T''} = pc^{ca}_{T''}, OWN($\tilde{lck} k'$) = ⊥ and OWN($\tilde{lck} k' + 1$) = $T'$ (cf. Algorithm 5.11).

Since $T' \in$ Thrd$^{ce}_ex$, it must be that $t^{ce}_{T'} +$ TIME($c', T'$) = min($\{t^{ce}_{T'} +$ TIME($c', T' \mid T \in$ Thrd$^{ce}_ex})$, and since $T'' \in$ Thrd$^{ce}_ex$, it must be that $t^{ca}_{T''} +$ TIME($c'', T'' \mid T \in$ Thrd$^{ce}_ex}). But since $c_0 \xrightarrow{prg} \ldots \xrightarrow{prg} c_n$, it must be that $t^{ce}_{T'} +$ TIME($c', T'$) ≤ $t^{ca}_{T''} +$ TIME($c'', T'' \mid T \in$ Thrd$^{ce}_ex$) (Lemma 4.2). Note that by choosing $c_0, c_n, c_0, c_0$ and $c_i$ (defined by Lemma 5.54) to be $c_0$, $c_n$, $c_0$, $c_0$ and $c_i$ (defined by this proof), respectively, and assuming that OWN($\tilde{lck} k'$) = ⊥ and REL($\tilde{lck} k'$) = $\gamma_i$(REL($\tilde{lck} k'$)) (which is actually not necessarily the case since $T'$ acquires lck$ k'$ in the transition between $c'$ and $c' + 1$; however, note that this assumption is okay since if $T'$ would not
acquire \( lck' \), then \( \text{OWN}(\llbracket lck' \rrbracket) = \bot_{\text{thrd}} \) and \( \text{REL}(\llbracket lck' \rrbracket) \in \gamma_t(\text{REL}(\llbracket lck' \rrbracket)) \) would hold since \( \text{OWN}(\llbracket lck' \rrbracket) = \bot_{\text{thrd}} \) and \( \text{REL}(\llbracket lck' \rrbracket) \in \gamma_t(\text{REL}(\llbracket lck' \rrbracket)) \), it is easy to see that \( t_{\text{thrd}}^a + \text{TIME}(e^n, T') \in \gamma_t(\text{DLLOCK}(\tilde{c}^j, lck')) \) since \( \text{Thrd}_{e^j} \subseteq \text{Thrd}_{e^j} \subseteq \text{Thrd}_{e^j} \subseteq \text{Thrd} \), \( \text{STM}(T'', p_{T''}^{a_e}) = [\text{lock} lck']p_{T''}^{a_e} \), \( T'' \in \text{Thrd}_{e''} \) and \( t_{T''}^a \in \gamma_t(\llbracket i \rrbracket) \) (Lemma 5.54). But then, since \( \min(\gamma_t(\text{DL}(\llbracket lck' \rrbracket))) = -\infty \), \( \text{DL}(\llbracket lck' \rrbracket) = \text{DLLOCK}(\tilde{c}^j, lck') \) and \( t_{T''}^a + \text{TIME}(e^j, T') \leq t_{T''}^a + \text{TIME}(e^n, T'') \), it must be that \( t_{T''}^a + \text{TIME}(e^j, T') \in \gamma_t(\text{DL}(\llbracket lck' \rrbracket)) \) which concludes the proof. 

Three lemmas will be presented in order to prove that the abstract transitions described by \( \rightarrow_{\text{pg}} \) safely approximate the concrete transitions described by \( \rightarrow_{\text{pg}} \). 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 \text{load}-statement on a global variable or that the thread issuing the \text{load}-statement is the sole thread in \( \text{Thrd}_{e''} \) in any step of the transition sequence. The first lemma (Lemma 5.57) states that the \text{halt}-, \text{skip}-, :=-, \text{if}-, \text{load}-, \text{store}- and \text{unlock}-statements, and also the \text{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 161).

**Lemma 5.57 (Soundness of \( \rightarrow_{\text{pg}} \), no frozen thread):**

*If the valid concrete configurations (cf. Definition 4.4), abstract configurations and thread*

\[
\begin{align*}
c^0 @ \langle [T, pc_T^0, x_T^0, t_T^0]_{T \in \text{Thrd}}, \infty^0, \bot^0 \rangle & \in \text{Conf}, \\
c^n @ \langle [T, pc_T^n, x_T^n, t_T^n]_{T \in \text{Thrd}}, \infty^n, \bot^n \rangle & \in \text{Conf}, \\
c^0 @ \langle [T, pc_T^0, x_T^0, t_T^0]_{T \in \text{Thrd}}, \infty^0, \bot^0 \rangle & \in \text{Conf}, \\
c^k @ \langle [T, pc_T^k, x_T^k, t_T^k]_{T \in \text{Thrd}}, \infty^k, \bot^k \rangle & \in \text{Conf}, \\
T' & \in \text{Thrd}_{e^k},
\end{align*}
\]
are such that

\[
\begin{align*}
0 &\leq n, \\
\varepsilon^0 \xrightarrow{\text{prg}} \ldots \xrightarrow{\text{prg}} \varepsilon^n, \\
0 &\leq k, \\
\varepsilon^0 \xrightarrow{\text{prg}} \ldots \xrightarrow{\text{prg}} \varepsilon^k,
\end{align*}
\]

\[\text{Thr}_{\leq k} \subseteq \text{Thr}_{\leq 0} \subseteq \text{Thr},\]

\[p_{c_T}^0 = p_{c_T}^{\varepsilon_0^0},\]

\[t_{T'}^{a_T} \in \gamma_T(t_{T'}^{\varepsilon_T^0}),\]

\[\exists x' \in \gamma_{\text{var}}(\mathbb{N}^0) : \forall x \in \text{Var} : \forall T \in \text{Thr} : ((x^0 x) T) \subseteq ((x') x) T,\]

\[\forall \text{lck} \in \text{Lck} : ((\text{OWN}(\mathbb{I}^0 \text{lck}) \neq \bot_{\text{thrd}}) \implies (\text{STT}(\mathbb{I}^0 \text{lck}) = \text{STT}(\mathbb{I}^0 \text{lck}) \land \text{OWN}(\mathbb{I}^0 \text{lck}) = \text{OWN}(\mathbb{I}^0 \text{lck}) \land \text{DL}(\mathbb{I}^0 \text{lck}) \in \gamma_T(\text{DL}(\mathbb{I}^0 \text{lck})) \land \text{POWN}(\mathbb{I}^0 \text{lck}) = \text{POWN}(\mathbb{I}^0 \text{lck}) \land \text{REL}(\mathbb{I}^0 \text{lck}) \in \gamma_T(\text{REL}(\mathbb{I}^0 \text{lck})) \land \min(\gamma_T(\text{DL}(\mathbb{I}^0 \text{lck}))) = -\infty) \land (\text{OWN}(\mathbb{I}^0 \text{lck}) = \bot_{\text{thrd}}) \implies ((\text{OWN}(\mathbb{I}^0 \text{lck}) = \text{OWN}(\mathbb{I}^0 \text{lck}) \lor (\text{OWN}(\mathbb{I}^0 \text{lck}) = T' \land \text{STT}(\mathbb{I}^0 \text{lck}) = \text{unlocked} \land t_{T'}^{a_T} + \text{TIME}(\varepsilon^a, T') \in \gamma_T(\text{DL}(\mathbb{I}^0 \text{lck})) \land \min(\gamma_T(\text{DL}(\mathbb{I}^0 \text{lck}))) = -\infty) \land \text{POWN}(\mathbb{I}^0 \text{lck}) = \text{POWN}(\mathbb{I}^0 \text{lck}) \land \text{REL}(\mathbb{I}^0 \text{lck}) \in \gamma_T(\text{REL}(\mathbb{I}^0 \text{lck})) \land \text{STM}(T', p_{c_T}^0) = [\text{lock lck}]^{p_{c_T}^0} \Rightarrow ((\mathbb{I}^0 \text{lck} = \mathbb{I}^0 \text{lck} \land \mathbb{I}^k \text{lck} = \mathbb{I}^0 \text{lck})))\right],
\]

\[\forall i \in \{0, \ldots, n-1\} : T' \notin \text{Thr}_{\leq i}^e,\]

\[\text{STM}(T', p_{c_T}^n) \neq [\text{halt}]^{p_{c_T}^n} \Rightarrow T' \in \text{Thr}_{\leq n}^e,\]

\[\forall i \in \{0, \ldots, k-1\} : T' \notin \text{Thr}_{\leq i}^e,\]

\[\text{STM}(T', p_{c_T}^k) \neq [\text{halt}]^{p_{c_T}^k} \Rightarrow T' \in \text{Thr}_{\leq k}^e,\]

\[\forall i \in \{0, \ldots, k\} : (|\text{Thr}_{\leq i}^e| \neq 1 \lor \{T \in \text{Thr}_{\leq i}^e | \exists r \in \text{Reg} : \exists x \in \text{Var}_g : \text{STM}(T, p_{c_T}^i) = [\text{load r from x}]^{p_{c_T}^i} = \emptyset\}),\]

where for all \(i \in \{0, \ldots, n\}\), \(\text{Thr}_{\leq i}^e\) is as defined in Table 4.3, for all \(i \in \{0, \ldots, k\}\), \(\text{Thr}_{\leq i}^e\) is as defined in Table 5.11, 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), then $\xrightarrow{prg}$ satisfies:

$$
\forall c @ \langle [T, pc_T, z_T, t_T]_{T \in \text{Thrd}}, \emptyset, [] \rangle \in \text{Conf} : 
(\overset{c^n}{\xrightarrow{prg}} c \Rightarrow \exists \tilde{c} @ \langle [T, pc^\tilde{c}_T, \tilde{z}_T, \tilde{t}_T]_{T \in \text{Thrd}_{\tilde{c}}}, \tilde{x}, \tilde{l} \rangle \in \text{Conf} : 
(\overset{\tilde{c}^k}{\xrightarrow{prg}} \tilde{c} \land 
\begin{align*}
&pc_{T'} = pc_{T'}, \land 
&z_{T'} \in \gamma_{\text{reg}}(\tilde{z}_{T'}) \land 
&t_{T'}^\gamma \in \gamma_{\text{ta}}(\tilde{t}_{T'}) \land 
&\exists \tilde{x}' \in \gamma_{\text{var}}(\tilde{x}) : (\forall x \in \text{Var} : ((\langle x \rangle T') \subseteq ((\langle x' \rangle T')) \land 
&\forall lck \in \text{Lck} : ((\langle \text{OWN}(\emptyset lck) = T' \lor \text{OWN}(\emptyset lck) = T' \Rightarrow 
(\text{STT}(\emptyset lck) = \text{STT}(\emptyset lck) \land 
\text{OWN}(\emptyset lck) = \text{OWN}(\emptyset lck) \land 
\text{DL}(\emptyset lck) \in \gamma_{\text{DL}}(\emptyset lck) \land 
\text{POWN}(\emptyset lck) = \text{POWN}(\emptyset lck) \land 
\text{REL}(\emptyset lck) \in \gamma_{\text{REL}}(\emptyset lck) \land 
\min(\gamma_{\text{DL}}(\emptyset lck))) = -\infty)))))
\end{align*}
) \Rightarrow
$$

\text{PROOF. Assume that the valid concrete configurations (cf. Definition 4.4), abstract configurations and thread}

$$
c^0 @ \langle [T, pc_T^0, z_T^0, t_T^0]_{T \in \text{Thrd}}, \emptyset, [] \rangle \in \text{Conf}, 
c^n @ \langle [T, pc_T^n, z_T^n, t_T^n]_{T \in \text{Thrd}}, \emptyset, [] \rangle \in \text{Conf}, 
c^0 @ \langle [T, pc_T^0, z_T^0, t_T^0]_{T \in \text{Thrd}_{\tilde{c}}}, \tilde{x}, \tilde{l} \rangle \in \text{Conf}, 
c^k @ \langle [T, pc_T^k, \tilde{z}_T^k, \tilde{t}_T^k]_{T \in \text{Thrd}_{\tilde{c}}}, \tilde{x}, \tilde{l} \rangle \in \text{Conf}, \text{ and}
T' \in \text{Thrd}_{\tilde{c}},$$
are such that
\[ 0 \leq n, \]
\[ e^0 \xrightarrow{prg} \ldots \xrightarrow{prg} e^n, \]
\[ 0 \leq k, \]
\[ e^0 \xrightarrow{prg} \ldots \xrightarrow{prg} e^k, \]
\[ \text{Thrd}_{e^k} \subseteq \text{Thrd}_{e^0} \subseteq \text{Thrd}, \]
\[ pc^0_{T'} = pc^0_{T}, \]
\[ \tau^0_{T'} \in \gamma_{reg}(T_{T'}^0), \]
\[ t_{T'}^a \in \gamma_{thrd}(T_{T'}^0), \]
\[ \exists x' \in \gamma_{var}(x^0) : \forall x \in \text{Var} : \forall T \in \text{Thrd} : ((x^0) x) T \subseteq ((x') x) T, \]
\[ \forall lck \in \text{Lck} : ((\text{OWN}(lck)^0) \neq \perp_{thrd} \Rightarrow \text{STT}(lck)^0 = \text{STT}(lck)^0 \land \text{OWN}(lck)^0 = \text{OWN}(lck)^0 \land \text{DL}(lck)^0 = \gamma_{thrd}(\text{DL}(lck)^0) \land \text{POWN}(lck)^0 = \gamma_{thrd}(\text{POWN}(lck)^0) \land \text{REL}(lck)^0 \in \gamma_{thrd}(\text{REL}(lck)^0) \land \text{min}(\gamma_{thrd}(\text{DL}(lck)^0)) = -\infty) \land \]
\[ (\text{OWN}(lck)^0) = \perp_{thrd} \Rightarrow (\text{OWN}(lck)^0) = \text{OWN}(lck)^0 \land \text{STT}(lck)^0 = \text{STT}(lck)^0 \land \text{OWN}(lck)^0 = T' \land \]
\[ t_{T'}^a + \text{TIME}(e^n, T') \in \gamma_{thrd}(\text{DL}(lck)^0) \land \text{min}(\gamma_{thrd}(\text{DL}(lck)^0)) = -\infty) \land \]
\[ \text{POWN}(lck)^0 = \gamma_{thrd}(\text{POWN}(lck)^0) \land \text{REL}(lck)^0 \in \gamma_{thrd}(\text{REL}(lck)^0) \land \]
\[ \text{STM}(T', pc^0_{T'}) = [\text{lock} \ lck]^0 \Rightarrow (lck)^0 = lck \land lck = lck \land (lck = lck))], \]
\[ \forall i \in \{0, \ldots, n - 1\} : T' \notin \text{Thrd}_{ex^i}, \]
\[ \text{STM}(T', pc^0_{T'}) \neq [\text{halt}]pc^0_{T'} \Rightarrow T' \in \text{Thrd}_{ex^n}, \]
\[ \forall i \in \{0, \ldots, k - 1\} : T' \notin \text{Thrd}_{ex^i}, \]
\[ \text{STM}(T', pc^k_{T'}) \neq [\text{halt}]pc^k_{T'} \Rightarrow T' \in \text{Thrd}_{ex^k}, \]
and
\[ \forall i \in \{0, \ldots, k\} : (|\text{Thrd}_{ex}^i| \neq 1 \lor \]
\[ \{T \in \text{Thrd}_{ex}^i | \exists r \in \text{Reg}_T : \exists x \in \text{Var}_g : \]
\[ \text{STM}(T, pc^i_{T}) = [\text{load} \ r \ from \ x]pc^i_{T} = \emptyset), \]
where for all \( i \in \{0, \ldots, n\} \), \( \text{Thrd}_{ex}^i \) is as defined in Table 4.3, for all \( i \in \{0, \ldots, k\} \), \( \text{Thrd}_{ex}^i \) is as defined in Table 5.11, 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 might be a data dependency between the threads).

First note that:

- Since \( \forall i \in \{0, \ldots, n-1\} : T' \not\subseteq \text{Thrd}_{\text{exe}}^i \), it must be that \( pc_{T'}^i = pc_0^0 \), \( n_{T'}^i = n_0^0 \), \( t_{T'}^i = t_0^0 \) and \( \forall \text{ lck} \in \text{Lck} : (\text{own}(\text{!}^0 \text{ lck}) = T' \Rightarrow \text{!}^n \text{ lck} = \text{!}^0 \text{ lck}) \) (cf. Table 4.3).

- Since \( \forall i \in \{0, \ldots, k-1\} : T' \not\subseteq \text{Thrd}_{\text{exe}}^i \), it must be that \( pc_{T'}^i = pc_0^0 \), \( n_{T'}^i = n_0^0 \), \( t_{T'}^i = t_0^0 \), \( \forall \text{ lck} \in \text{Lck} : (\text{own}(\text{!}^0 \text{ lck}) = T' \Rightarrow (\text{!}^k \text{ lck} = \text{!}^0 \text{ lck} \land \min(\gamma_i(\text{DL}(\text{!}^k \text{ lck}))) = -\infty)) \).

- Since \( pc_{T'} = pc_0^0 \), \( n_{T'} = n_0^0 \), \( t_{T'} = t_0^0 \), \( \forall \text{ lck} \in \text{Lck} : (\text{own}(\text{!}^0 \text{ lck}) = T' \Rightarrow (\text{!}^n \text{ lck} = \text{!}^0 \text{ lck}), pc_{T'}^k = pc_0^0 \), \( n_{T'}^k = n_0^0 \), \( t_{T'}^k = t_0^0 \), \( \forall \text{ lck} \in \text{Lck} : (\text{own}(\text{!}^0 \text{ lck}) = T' \Rightarrow (\text{!}^k \text{ lck} = \text{!}^0 \text{ lck} \land \min(\gamma_i(\text{DL}(\text{!}^k \text{ lck}))) = -\infty)) \), it must be that:

\[
\begin{align*}
pc_{T'}^i &= pc_0^k \\
n_{T'}^i &= n_{T'}^k \\
t_{T'}^i &= t_{T'}^k \\
\forall \text{ lck} \in \text{Lck} : (\text{own}(\text{!}^n \text{ lck}) = T' \Rightarrow (\text{STT}(\text{!}^n \text{ lck}) = \text{STT}(\text{!}^k \text{ lck}) \land \\
\text{own}(\text{!}^n \text{ lck}) = \text{own}(\text{!}^k \text{ lck}) \land \\
\text{DL}(\text{!}^n \text{ lck}) = \gamma_i(\text{DL}(\text{!}^k \text{ lck})) \land \\
\text{POWN}(\text{!}^n \text{ lck}) = \gamma_i(\text{REL}(\text{!}^k \text{ lck})) \land \\
\min(\gamma_i(\text{DL}(\text{!}^k \text{ lck}))) = -\infty))
\end{align*}
\]

- Since \( \gamma_0^0 \rightarrow_p \cdots \rightarrow_p \gamma^n \) and \( \forall i \in \{0, \ldots, n-1\} : T' \not\subseteq \text{Thrd}_{\text{exe}}^i \), it must be that for all \( x \in \text{Var} \), \( ((\gamma^n x) T') = ((\gamma^0 x) T') \) if no thread writes to \( x \) in the sequence \( \gamma_0^0 \rightarrow_p \cdots \rightarrow_p \gamma^n \), or \( ((\gamma^n x) T') = \emptyset \) if some other thread has written to \( x \) in the given sequence (cf. Table 4.3). Thus, \( \forall x \in \text{Var} : ((\gamma^n x) T') \subseteq ((\gamma^0 x) T') \).
• Since $\exists x' \in \gamma_{\text{var}}(\tilde{x}_0) : \forall x \in \text{Var} : ((\tilde{x}_0^0 x) T') \subseteq ((\tilde{x}' x) T'), c^0_{\text{prog}} \rightarrow \ldots \rightarrow c^n, \tilde{c}_0^0_{\text{prog}} \rightarrow \ldots \rightarrow \tilde{c}^k, \forall i \in \{0, \ldots, n - 1\} : T' \not\in \text{Thrd}_{\text{exe}}^{\tilde{c}_i}, \forall i \in \{0, \ldots, k - 1\} : T' \not\in \text{Thrd}_{\text{exe}}^{\tilde{c}_i}$ and TRIM is safe (Lemma 5.28), it must be that $\exists x' \in \gamma_{\text{var}}(\tilde{x}_k) : \forall x \in \text{Var} : ((\tilde{x}_k x) T') \subseteq ((\tilde{x}' x) T')$.

• Since $\forall i \in \{0, \ldots, k\} : (\|\text{Thrd}_{\text{exe}}^{\tilde{c}_i} \| \neq 1 \lor \{ T \in \text{Thrd}_{\text{exe}}^{\tilde{c}_i} | \exists r \in \text{Reg}_T : \exists x \in \text{Var}_g : \text{STM}(T, pc_{\tilde{c}_i}) = [\text{load } r \text{ from } x] pc_{\tilde{c}_i}^T = 0\})$, it must be that $\forall i \in \{0, \ldots, k\} : (\{ T \in \text{Thrd}_{\text{exe}}^{\tilde{c}_i} | \exists r \in \text{Reg}_T : \exists x \in \text{Var}_g : \text{STM}(T, pc_{\tilde{c}_i}) = [\text{load } r \text{ from } x] pc_{\tilde{c}_i}^T = 0\} \neq 0 \Rightarrow \|\text{Thrd}_{\text{exe}}^{\tilde{c}_i} \| = 1)$. This means that if some thread in $\text{Thrd}_{\text{exe}}^{\tilde{c}_i}$, where $i \in \{0, \ldots, k\}$, performs a load-statement, there is only one single thread in $\text{Thrd}_{\text{exe}}^{\tilde{c}_i}$, thus that thread performs the load-statement. It is then easy to see from the definition of $\text{Thrd}_{\text{exe}}^{\tilde{c}_i}$, that there cannot occur any other write than those represented by $\tilde{x}_i$ such that it could affect the load-statement of the thread in $\text{Thrd}_{\text{exe}}^{\tilde{c}_i}$ (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 safe write history (cf. Definition 5.19).

• Since, trivially, $\forall lck \in \text{Lck} : \{ T \in \text{Thrd}_{\text{exe}}^{\tilde{c}_i} \cap \text{Thrd}_{\text{exe}}^{\tilde{c}_k} | \text{STM}(T, pc_{\tilde{c}_i}^T) = [\text{load } lck] pc_{\tilde{c}_i}^T \} \subseteq \{ T \in \text{Thrd}_{\text{exe}}^{\tilde{c}_k} | \exists l \in \text{Lbl}_T : \text{STM}(T, l) = [\text{lock } lck] l \} \}$, 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 $lck \in \text{Lck}$, $\text{STM}(T', pc_{\tilde{c}_k}^T) = [\text{lock } lck] pc_{\tilde{c}_k}^T$, it must be that $\text{OWN}(\tilde{l}^k_0 lck) = T'$, since $\forall i \in \{0, \ldots, k - 1\} : T' \not\in \text{Thrd}_{\text{exe}}^{\tilde{c}_i}$ and $T' \in \text{Thrd}_{\text{exe}}^{\tilde{c}_k}$.

• Since $T' \in \text{Thrd}_{\text{exe}}^{\tilde{c}_k}$, $\text{Thrd}_{\text{exe}}^{\tilde{c}_k} \subseteq \text{Thrd}_{\text{exe}}^{\tilde{c}_k}$. $pc_{\tilde{c}_k}^T = pc_{\tilde{c}_k}^{T'}$, $t_{\tilde{c}_k}^{T'} \in \gamma_{\tilde{c}_k}(T_{\tilde{c}_k}^{T'}), \forall lck \in \text{Lck} : (\text{OWN}(\tilde{l}^k_0 lck) = T' \Rightarrow (\text{STT}(\tilde{l}^k lck) = \tilde{c}_{\text{TT}}(\tilde{l}^k lck) \land \text{OWN}(\tilde{l}^k_0 lck) = \text{OWN}(\tilde{l}^k lck) \land \text{DL}(\tilde{l}^k lck) \in \gamma_{\tilde{c}_k}(\text{DL}(\tilde{l}^k lck)) \land \text{OWN}(\tilde{l}^k_0 lck) = \text{OWN}(\tilde{l}^k lck) \land \text{REL}(\tilde{l}^k lck)))$, $T' \in \text{Thrd}_{\text{exe}}^{\tilde{c}_k}$, $T' \in \text{Thrd}_{\text{exe}}^{\tilde{c}_k} \forall lck \in \text{Lck} : (\text{STM}(T', pc_{\tilde{c}_k}^{T'}) = [\text{lock } lck] pc_{\tilde{c}_k}^{T'} \Rightarrow \text{OWN}(\tilde{l}^k_0 lck) = T')$, it must be that $t_{\tilde{c}_k}^{T'} \in \gamma_{\tilde{c}_k}(\text{ACC_TIME}(\{ [T, pc_{\tilde{c}_k}^{T'}, \tilde{c}_k, t_{\tilde{c}_k}^{T'}] T \in \text{Thrd}_{\text{exe}}^{\tilde{c}_k} | \tilde{x}_k, \tilde{l}^k_0 lck \}, \text{Thrd}_{\text{exe}}^{\tilde{c}_k}(T')) \}$ whenever $\forall lck \in \text{Lck} : (\text{STM}(T', pc_{\tilde{c}_k}^{T'}) = [\text{lock } lck] pc_{\tilde{c}_k}^{T'} \Rightarrow (\text{OWN}(\tilde{l}^k_0 lck) = T' \land \text{OWN}(\tilde{l}^k_0 lck) = T'))$ (Lemma 5.55), where $t_{\tilde{c}_k}^{T'}$.
is derived from $c'^n \xrightarrow{p_{pg}} \langle [T, pc_T, \tau_T, t_T] | T \in \text{Thrd}, x, \emptyset \rangle$ and $\llbracket^m$ and $\llbracket^k$ are defined as in Tables 4.3 and 5.11, respectively.

- Since $\forall lck \in \text{Lck} : (\text{OWN}([0] lck) = \bot_{-\text{thrd}} \Rightarrow (\text{STM}(T', pc_{T'}^0) = [\text{lock} lck] pc_{T'}^0 \Rightarrow ([n] lck = [0] lck \land [k] lck = [0] lck))), \forall i \in \{0, \ldots, n-1\} : T' \notin \text{Thrd}_{exe}, \text{STM}(T', pc_{T'}^n) \neq [\text{halt}] pc_{T'}^k \Rightarrow T' \in \text{Thrd}_{exn}$ and $\text{OWN}([k] lck) = T'$, it must be that $T'$ immediately acquires $lck$ (i.e. without any other thread acquiring and possibly releasing $lck$ in the sequence $c'^0 \xrightarrow{p_{pg}} \ldots \xrightarrow{p_{pg}} c'^n$) if $\text{STM}(T', pc_{T'}^n) = [\text{lock} lck] pc_{T'}^k$, both in the concrete and abstract cases (based on $c'$ and $c^k$).

- If, for some lock, $lck' \in \text{Lck}$, $\text{STM}(T', pc_{T'}^0) = [\text{lock} lck'] pc_{T'}^0$ and $\text{OWN}([0] lck') = \bot_{-\text{thrd}}$, it must be that $\min\{\{t_{T'}^n + \text{TIME}(c, T) | T \in \text{Thrd}\} \} \in \gamma_i(\text{DLLOCK}(\tilde{c}, lck'))$, since $T' \in \text{Thrd}_{ek}$, $\text{Thrd}_{ek} \subseteq \text{Thrd}$, $m$, $n$ and $j$ (cf. Lemma 5.54) can be chosen to be $0$, $n$ and $k$ (given by this proof), respectively, $t_{T'}^0 \in \gamma_i(\tilde{t}_{T'}^0)$, $t_{T'}^n = t_{T'}^0$, $\tilde{t}_{T'}^n = \tilde{t}_{T'}^0$, $pc_{T'}^n = pc_{T'}^0 = pc_{T'} = pc_{T'}^0$, $T' \in \text{Thrd}_{exe}$ and (since $\forall lck \in \text{Lck} : (\text{OWN}([0] lck) = \bot_{-\text{thrd}} \Rightarrow (\text{STM}(T', pc_{T'}^0) = [\text{lock} lck] pc_{T'}^0 \Rightarrow ([n] lck = [0] lck \land [k] lck = [0] lck))))) and $\text{REL}([0] lck') \in \gamma_i(\text{REL}([0] lck')) \) \in \gamma_i(\text{REL}([k] lck')) \) (Lemma 5.54). Thus, $t_{T'}^n + \text{TIME}(c,T') \in \gamma_i(\text{DLLOCK}(\tilde{c}, lck'))$, since $T' \in \text{Thrd}_{exn}$ which means that $t_{T'}^n + \text{TIME}(c,T') = \min\{\{t_{T'}^n + \text{TIME}(c', T) | T \in \text{Thrd}\} \} \) \in \gamma_i(\text{DLLOCK}(\tilde{c}, lck'))$, since $T' \in \text{Thrd}_{exn}$ which means that $t_{T'}^n + \text{TIME}(c,T') = \min\{\{t_{T'}^n + \text{TIME}(c', T) | T \in \text{Thrd}\} \} \) (cf. Table 4.3).

The proof will now be conducted by considering the different statements that $T'$ could issue in $c^0$ (i.e. in $c'$).

1. If $\text{STM}(T', pc_{T'}^0) = [\text{halt}] pc_{T'}^0$, then it must be that $T' \notin \text{Thrd}_{exn}$. Thus, it must be that $c'^n \xrightarrow{p_{pg}} c$, where $c @ \langle [T, pc_T, \tau_T, t_T] | T \in \text{Thrd}, x, \emptyset \rangle$ is such that $pc_T = pc_T^n$, $\tau_T = \tau_T^n$, $t_T^a = t_T^{a^0}$, $\forall lck \in \text{Lck} : (\text{OWN}([0] lck) = T' \Rightarrow [lck = [0] lck])$ and $\forall x \in \text{Var} : ((\lambda x) T') \subseteq ((\lambda x) T')$, provided that $\exists T \in \text{Thrd} : \text{STM}(T, pc_T^0) \neq [\text{halt}] pc_{T'}^0$ (otherwise $\xrightarrow{p_{pg}}$ is not applicable; cf. Table 4.3).

Note that $T' \notin \text{Thrd}_{exn}$ and choose $\tilde{c} @ \langle [T, pc_T^0, \tau_T, t_T] | T \in \text{Thrd}, x, \emptyset \rangle$ such that $\tilde{c}^k \xrightarrow{p_{pg}} \tilde{c}$, i.e. $pc_T^k = pc_T^k$, $\tau_T = \tau_T^k$, $t_T^k = t_T^k$, $\forall lck \in \text{Lck} :
(\text{OWN}(\mathbb{I}^0 \text{lck}) = T' \Rightarrow (\mathbb{I} \text{lck} = \mathbb{I}^k \text{lck} \land \min(\gamma_t(\tilde{\text{DL}}(\tilde{\text{lck})))) = -\infty)).

Note that \(\tilde{x}\) must still be such that for all \(x \in \text{Var}\), \(((\tilde{x}, x) T')\) is a safe approximation of the writes performed on \(x\) by \(T'\) since TRIM is safe (Lemma 5.28). Thus, it must be that:

\[
\begin{align*}
\text{pc}_{T'} &= \text{pc}_{\tilde{x}}^0 \\
\tau_{T'} &\in \gamma_{\text{reg}}(\tilde{x}_{T'}) \\
t^a_{T'} &\in \gamma_l(t_{\tilde{x}}^{a, b}_{T'}) \\
\exists \tilde{x}' \in \gamma_{\text{var}}(\tilde{x}) : (\forall x \in \text{Var} : (((x, x) T') \subseteq ((x', x) T')) \\
\forall \text{lck} \in \text{Lck} : ((\text{OWN}(\mathbb{I}^0 \text{lck}) = T' \lor \text{OWN}(\mathbb{I} \text{lck}) = T') \Rightarrow \\
(\text{STT}(\mathbb{I} \text{lck}) = \text{STT}(\tilde{\text{lck}}) \\
\text{OWN}(\mathbb{I} \text{lck}) = \text{OWN}(\tilde{\text{lck}}) \\
\tilde{\text{DL}}(\tilde{\text{lck}}) \in \gamma_t(\tilde{\text{DL}}(\tilde{\text{lck}})) \\
\text{POWN}(\tilde{\text{lck}}) = \text{POWN}(\tilde{\text{lck}}) \\
\text{REL}(\tilde{\text{lck}}) \in \gamma_t(\tilde{\text{REL}}(\tilde{\text{lck}})) \\
\min(\gamma_t(\tilde{\text{DL}}(\tilde{\text{lck}}))) = -\infty))
\end{align*}
\]

2. If, for some \(a \in \text{Aexp}, b \in \text{Bexp}, l \in \text{Llbl}_{T'}, r \in \text{Reg}_{T'}, x \in \text{Var}\) and \(\text{lck} \in \text{Lck}, \text{STM}(T', \text{pc}_{T'}^0) \in \{[\text{skip}]^{\text{pc}_{T'}^0}, [r := a]^{\text{pc}_{T'}^0}, [\text{if } b \text{ goto } l]^{\text{pc}_{T'}^0}, [\text{store } r \text{ to } x]^{\text{pc}_{T'}^0}, [\text{unlock lck}]^{\text{pc}_{T'}^0}\},\) then let the configuration \(c@([T, \text{pc}_{T}, \tau_{T}, t^a_{T}])_{T \in \text{Thrd}_{\tilde{x}}, \mathbb{I}, \tilde{lck}}\) be such that \(c^n \xrightarrow{prg} c\) and choose \(\tilde{c}@([T, \text{pc}_{T}, \tilde{\tau}_{T}, t^a_{T}])_{T \in \text{Thrd}_{\tilde{x}}, \mathbb{I}, \tilde{lck}}\) such that \(\tilde{c}^k \xrightarrow{prg} \tilde{c}\). Thus, since \(\forall i \in \{0, \ldots, n - 1\} : T' \not\in \text{Thrd}^{c^i}_{\text{exe}}, T' \in \text{Thrd}^{c^i}_{\text{exe}}, \forall i \in \{0, \ldots, k - 1\} : T' \not\in \text{Thrd}^{c^i}_{\text{exe}}, T' \in \text{Thrd}^{c^i}_{\text{exe}}, \xrightarrow{ax}\) is a safe approximation of \(\xrightarrow{ax}\) (Lemma 5.50), TRIM is safe (Lemma 5.28), \(\text{Thrd}_{c^k} \subseteq \text{Thrd}\) and \(\text{ACC} \text{TIME}\) is safe (Lemma 5.55), it must be that:

\[
\begin{align*}
\text{pc}_{T'} &= \text{pc}_{\tilde{x}}^0 \\
\tau_{T'} &\in \gamma_{\text{reg}}(\tilde{x}_{T'}) \\
t^a_{T'} &\in \gamma_l(t_{\tilde{x}}^{a, b}_{T'}) \\
\exists \tilde{x}' \in \gamma_{\text{var}}(\tilde{x}) : (\forall x \in \text{Var} : (((x, x) T') \subseteq ((x', x) T')) \\
\forall \text{lck} \in \text{Lck} : ((\text{OWN}(\mathbb{I}^0 \text{lck}) = T' \lor \text{OWN}(\mathbb{I} \text{lck}) = T') \Rightarrow \\
(\text{STT}(\mathbb{I} \text{lck}) = \text{STT}(\tilde{\text{lck}}) \\
\text{OWN}(\mathbb{I} \text{lck}) = \text{OWN}(\tilde{\text{lck}}) \\
\tilde{\text{DL}}(\tilde{\text{lck}}) \in \gamma_t(\tilde{\text{DL}}(\tilde{\text{lck}})) \\
\text{POWN}(\tilde{\text{lck}}) = \text{POWN}(\tilde{\text{lck}}) \\
\text{REL}(\tilde{\text{lck}}) \in \gamma_t(\tilde{\text{REL}}(\tilde{\text{lck}})) \\
\min(\gamma_t(\tilde{\text{DL}}(\tilde{\text{lck}}))) = -\infty))
\end{align*}
\]
Note that in the case $\text{STM}(T', pc^0_{T'}) = [\text{if } b \text{ goto } l]^{pc^0_{T'}}$, $\hat{c}$ can be chosen so that the corresponding branch to that taken in $c$ is taken since $t^0_T \in \gamma_{reg}(\hat{x}^0_T)$ (cf. Table 5.10 and Definition 5.8).

3. If, for some $r \in \text{Reg}^T$ and $x \in \text{Var}$, $\text{STM}(T', pc^0_{T'}) = [\text{load } r \text{ from } x]^{pc^0_{T'}}$, then let $c@\langle[T, pc_T, \hat{z}_T, t^0_T]_{T \in \text{Thrd}^e_X, \hat{z}, \hat{\ell}} \rangle$ be such that $c^\mu_{prg} \rightarrow c$ and choose $\hat{c}@\langle[T, pc^\hat{e}_T, \hat{z}_T, \hat{t}^0_T]_{T \in \text{Thrd}^e_X, \hat{z}, \hat{\ell}} \rangle$ such that $\hat{c}^k_{prg} \rightarrow \hat{c}$. Since $\forall i \in \{0, \ldots, n - 1\}$: $T' \not\in \text{Thrd}^e_{exe}, T' \in \text{Thrd}^d_{exe}, \forall i \in \{0, \ldots, k - 1\}$: $T' \not\in \text{Thrd}^d_{exe}, T' \in \text{Thrd}^d_{exe}, \rightarrow_{as}$ is a safe approximation of $\rightarrow$ (Lemma 5.50), $\hat{x}^k$ contains safe write history, TRIM is safe (Lemma 5.28), $\text{Thrd}_{ek} \subseteq \text{Thrd}$ and $\text{ACCTIME}$ is safe (Lemma 5.55), it must be that:

$$
\begin{align*}
pc_T &= pc^\hat{e}_T \\
\forall y_T \in \gamma_{reg}(\hat{x}^T_Y) \land t^a_T &\in \gamma_i(\hat{x}^T_Y) \\
\exists x'_T \in \gamma_{var}(\hat{x}) : (\forall x \in \text{Var} : ((x, x) T') \subseteq ((x', x) T')) \land \\
\forall lck \in \text{Lck} : ((\text{OWN}(\ell) lck) = T' \lor \text{OWN}(\ell) lck) = T') \Rightarrow \\
& (\text{STT}(\ell lck) = \text{STT}(\ell lck) \land \\
\text{OWN}(\ell lck) = \text{OWN}(\ell lck) \land \\
\text{DL}(\ell lck) \in \gamma_i(\text{DL}(\ell lck)) \land \\
\text{POWN}(\ell lck) = \text{POWN}(\ell lck) \land \\
\text{REL}(\ell lck) \in \gamma_i(\text{REL}(\ell lck)) \land \\
\text{min}(\gamma_i(\text{DL}(\ell lck))) = -\infty)
\end{align*}
$$

4. If, for some $lck' \in \text{Lck}$, $\text{STM}(T', pc^0_{T'}) = [\text{lock } lck']^{pc^0_{T'}}$, 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, $\text{ACCTIME}$ is safe since it must be that $\text{OWN}(\ell_{\text{num}} lck') = T'$ and $\text{OWN}(\ell_{\text{num}} lck') = T'$ (Lemma 5.55).

Since $\text{OWN}(\ell_{\text{num}} lck') = T' \Rightarrow \text{OWN}(\ell_{\text{num}} lck') = T'$ and $\text{OWN}(\ell_{\text{num}} lck') = \perp_{\text{thrd}} \Rightarrow (\text{OWN}(\ell_{\text{num}} lck') = \perp_{\text{thrd}} \lor \text{OWN}(\ell_{\text{num}} lck') = T')$, there are three cases to consider.

(a) Assume that $\text{OWN}(\ell_{\text{num}} lck') = T'$ (and thus, $\text{OWN}(\ell_{\text{num}} lck') = T'$) and let $c@\langle[T, pc_T, \hat{z}_T, t^0_T]_{T \in \text{Thrd}^e_X, \hat{z}, \hat{\ell}} \rangle$ be such that $c^\mu_{prg} \rightarrow c$. Then choose $\hat{c}@\langle[T, pc^\hat{e}_T, \hat{z}_T, \hat{t}^0_T]_{T \in \text{Thrd}^e_X, \hat{z}, \hat{\ell}} \rangle$ such that $\hat{c}^k_{prg} \rightarrow \hat{c}$. It is
trivially the case that \(\text{OWN}(\downarrow\text{lck}') = \overline{\text{OWN}}(\uparrow\text{lck}') = T'\) and thus

\[
p_{c_T'} = p_{c_T'}' \land \\
\gamma_{T'} \in \gamma_{\text{reg}}(\tilde{c}_{T'}) \land \\
r_{T'} \in \gamma_{(T')} \\
\exists x' \in \gamma_{\text{var}}(\tilde{x}) : (\forall x \in \text{Var} : ((x \ x) T') \subseteq ((x' \ x) T')) \land \\
\forall lck \in \text{Lck} : ((\text{OWN}(\uparrow^0\text{lck}) = T' \lor \text{OWN}(\downarrow\text{lck}) = T') \Rightarrow \\
(\text{STT}(\uparrow\text{lck}) = \text{STT}(\downarrow\text{lck}) \land \\
\text{OWN}(\downarrow\text{lck}) = \overline{\text{OWN}}(\uparrow\text{lck}) \land \\
\text{DL}(\downarrow\text{lck}) \in \gamma_{(\text{DL}(\uparrow\text{lck}))} \land \\
\text{POWN}(\uparrow\text{lck}) = \overline{\text{POWN}}(\downarrow\text{lck}) \land \\
\text{REL}(\downarrow\text{lck}) \in \gamma_{(\text{REL}(\uparrow\text{lck}))} \land \\
\text{min}(\gamma_{(\text{DL}(\uparrow\text{lck}))}) = (\infty))
\]

since \(\forall i \in \{0, \ldots, n - 1\} : T' \not\in \text{Thrd}_{\text{exe}}^c, T' \in \text{Thrd}_{\text{exe}}^a, \forall i \in \{0, \ldots, k - 1\} : T' \not\in \text{Thrd}_{\text{exe}}^d, T' \in \text{Thrd}_{\text{exe}}^e, \xrightarrow{ax} \text{TRIM is safe} \) (Lemma 5.50) and \(\text{TRIM is safe} \) (Lemma 5.28). (b) Assume that \(\text{OWN}(\uparrow^0\text{lck}') = \overline{\text{OWN}}(\uparrow^0\text{lck}') = \perp_{\text{thr}}\) and let \(c @ \langle [T, p_{c_T}, r_{T'}, r_{T'}]_{T \in \text{Thrd}_{\text{exe}}} \rangle \) be such that \(c_{\text{pr}} \xrightarrow{ax} c \land \text{OWN}(\downarrow\text{lck}') = T'\)

and choose \(\overline{c} @ \langle [T, p_{c_T}, \overline{r}_{T'}, r_{T'}]_{T \in \text{Thrd}_{\text{exe}}} \rangle\) such that \(\overline{c}_{\text{pr}} \xrightarrow{ax} \overline{c}\).

Then it must be that \(\overline{\text{OWN}}(\uparrow\text{lck}) = T'\) (since \(T' \in \text{Thrd}_{\text{exe}}^c\) and thus \(\overline{\text{OWN}}(\uparrow^{c_k}\text{lck}') = T'\) and

\[
p_{c_T'} = p_{c_T'}' \land \\
\gamma_{T'} \in \gamma_{\text{reg}}(\tilde{c}_{T'}) \land \\
r_{T'} \in \gamma_{(T')} \\
\exists x' \in \gamma_{\text{var}}(\tilde{x}) : (\forall x \in \text{Var} : ((x \ x) T') \subseteq ((x' \ x) T')) \land \\
\forall lck \in \text{Lck} : ((\text{OWN}(\uparrow^0\text{lck}) = T' \lor \text{OWN}(\downarrow\text{lck}) = T') \Rightarrow \\
(\text{STT}(\uparrow\text{lck}) = \text{STT}(\downarrow\text{lck}) \land \\
\text{OWN}(\downarrow\text{lck}) = \overline{\text{OWN}}(\uparrow\text{lck}) \land \\
\text{DL}(\downarrow\text{lck}) \in \gamma_{(\text{DL}(\uparrow\text{lck}))} \land \\
\text{POWN}(\uparrow\text{lck}) = \overline{\text{POWN}}(\downarrow\text{lck}) \land \\
\text{REL}(\downarrow\text{lck}) \in \gamma_{(\text{REL}(\uparrow\text{lck}))} \land \\
\text{min}(\gamma_{(\text{DL}(\uparrow\text{lck}))}) = (\infty))
\]

since \(\forall i \in \{0, \ldots, n - 1\} : T' \not\in \text{Thrd}_{\text{exe}}^c, T' \in \text{Thrd}_{\text{exe}}^a, \forall i \in \{0, \ldots, k - 1\} : T' \not\in \text{Thrd}_{\text{exe}}^d, T' \in \text{Thrd}_{\text{exe}}^e, \xrightarrow{ax} \text{is a safe approxi-}
mation of $\overrightarrow{a}$ (Lemma 5.50), TRIM is safe (Lemma 5.28), $\text{Thrd}_{\text{ck}} \subseteq \text{Thrd}$, $\forall c' \in \text{Conf} : \forall lck \in \text{Lck} : \min(\text{DLLock}(c', lck)) = -\infty$ (Algorithm 5.11), $\text{DL}(\llbracket lck' \rrbracket) = t^a_{\text{Thr}} = t^\theta_{\text{Thr}} + \text{TIME}(c^n, T')$ (Table 4.3) and $t^a_{\text{Thr}} \in \gamma_i(\text{DLLock}(c^k, lck'))$ (Lemma 5.54).

(c) Assume that $\text{OWN}(\llbracket 0 lck' \rrbracket) = \bot_{\text{Thr}}$ and $\text{OWN}(\llbracket 0 lck' \rrbracket) = T'$ and let $c @ [T, pc_T, \tau_T, t^a_T]_{T \in \text{Thr}} \in \text{Thr}_{\text{ck}}$, $\llbracket x, 0 \rrbracket$ be such that $c \rightarrow\rightarrow c'$ such that $\llbracket T, pc_T, \tau_T, t^a_T \rrbracket_{T \in \text{Thr}_{\text{ck}}}$, $\llbracket x, 0 \rrbracket$ such that $c^k \rightarrow\rightarrow \overrightarrow{c}$. Then it is easy to see that $\text{OWN}(\llbracket lck' \rrbracket) = \text{OWN}(\llbracket lck' \rrbracket) = T'$.

First note that since $\text{OWN}(\llbracket 0 lck' \rrbracket) = \text{OWN}(\llbracket 0 lck' \rrbracket) = \bot_{\text{Thr}} \land \text{OWN}(\llbracket 0 lck' \rrbracket) = \text{OWN}(\llbracket 0 lck' \rrbracket) = T'$, it must be that $\text{STT}(\llbracket 0 lck' \rrbracket) = \text{STT}(\llbracket 0 lck' \rrbracket) = \text{STT}(\llbracket 0 lck' \rrbracket) = \text{unlocked}, \text{DL}(\llbracket lck' \rrbracket) = \text{DL}(\llbracket 0 lck' \rrbracket) = \text{DL}(\llbracket 0 lck' \rrbracket), \text{and thus min}(\gamma_i(\text{DL}(\llbracket lck' \rrbracket))) = -\infty$ and $t^a_{\text{Thr}} + \text{TIME}(c^n, T') \in \gamma_i(\text{DL}(\llbracket lck' \rrbracket))$, means that $\text{DL}(\llbracket lck' \rrbracket) \in \gamma_i(\text{DL}(\llbracket lck' \rrbracket))$ since $\text{DL}(\llbracket lck' \rrbracket) = t^a_{\text{Thr}} + \text{TIME}(c^n, T')$.

Note that since $t^a_{\text{Thr}} \in \gamma_i(\text{Thrd}_{\text{ck}})$, it must be that $\text{DL}(\llbracket 0 lck' \rrbracket) \overrightarrow{c}, t^a_{\text{Thr}} \bot_{\text{Thr}} \text{ABSTIME}(c^k, T')$. It is thus easy to see that

$$
\begin{align*}
pc_T &= pc_T^\theta / \\
\tau_T^\theta &= \gamma_{\text{reg}}(\tilde{x}_T^\theta) / \\
t^a_T &= \gamma_i(\tilde{t}^a_T) / \\
\exists \tilde{x}^\theta : \gamma_{\text{var}}(\tilde{x}^\theta) : (\forall x \in \text{Var} : ((\llbracket x \rrbracket x) T') \subseteq ((\llbracket x' \rrbracket x' T'))) / \\
\forall lck \in \text{Lck} : (((\text{OWN}(\llbracket 0 lck' \rrbracket) = T' \lor \text{OWN}(\llbracket lck' \rrbracket) = T')) \Rightarrow \\
\text{STT}(\llbracket lck' \rrbracket) = \text{STT}(\llbracket lck' \rrbracket) \land \\
\text{OWN}(\llbracket lck' \rrbracket) = \text{OWN}(\llbracket lck' \rrbracket) \land \\
\text{DL}(\llbracket lck' \rrbracket) = \gamma_i(\text{DL}(\llbracket lck' \rrbracket)) \land \\
\text{POWN}(\llbracket lck' \rrbracket) = \text{POWN}(\llbracket lck' \rrbracket) \land \\
\text{REL}(\llbracket lck' \rrbracket) \in \gamma_i(\text{REL}(\llbracket lck' \rrbracket)) \land \\
\text{min}(\gamma_i(\text{DL}(\llbracket lck' \rrbracket))) = -\infty)
\end{align*}
$$

since $\forall i \in \{0, \ldots , n - 1\} : T' \not\in \text{Thrd}_{\text{exe}}$, $T' \in \text{Thrd}_{\text{exe}}$, $\forall i \in \{0, \ldots , k - 1\} : T' \not\in \text{Thrd}_{\text{exe}}$, $T' \in \text{Thrd}_{\text{exe}}$, $\overrightarrow{a}$ is a safe approximation of $\overrightarrow{a}$ (Lemma 5.50), TRIM is safe (Lemma 5.28) and $\text{Thrd}_{\text{ck}} \subseteq \text{Thrd}$.

This concludes the proof. \qed
Lemma 5.58 states that \( \xrightarrow{prg} \) safely approximates the case that a thread, \( T \in \text{Thrd}_e \) issuing \texttt{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 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 \texttt{load}-statement on a global variable or that the thread issuing the \texttt{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 161).

\textbf{Lemma 5.58 (Soundness of } \xrightarrow{prg} \text{, frozen thread):}

If the valid concrete configurations (cf. Definition 4.4), abstract configurations, \texttt{lock} and \texttt{thread}

\[ c^0 @ \langle [T, pc^0_T, x^0_T, t^0_T] | T \in \text{Thrd}, x^0, [0] \rangle \in \text{Conf}, \]
\[ c^n @ \langle [T, pc^n_T, x^n_T, t^n_T] | T \in \text{Thrd}, x^n, [n] \rangle \in \text{Conf}, \]
\[ c^m @ \langle [T, pc^m_T, x^m_T, t^m_T] | T \in \text{Thrd}, x^m, [m] \rangle \in \text{Conf}, \]
\[ c^n @ \langle [T, pc_n^T, x_n^T, t_n^T] | T \in \text{Thrd}, x_n, [n] \rangle \in \text{Conf}, \]
\[ c^0 @ \langle [T, pc^0_T, x^0_T, t^0_T] | T \in \text{Thrd}_g, x^0, [0] \rangle \in \text{Conf}, \]
\[ c^n @ \langle [T, pc^n_T, x^n_T, t^n_T] | T \in \text{Thrd}_g, x^n, [n] \rangle \in \text{Conf}, \]
\[ lck' \in \text{Lck}, \text{ and} \]
\[ T' \in \text{Thrd}_{\text{ck}}, \]
are such that

\[
\text{STM}(T', pc_{T'}^0) = \{\text{lock } lck \}^{| pc_{T'}^0 |},
\]

\[
0 \leq n_1 \leq n_2 \leq n_m \leq n,
\]

\[
e_0 \xrightarrow{prg} \ldots \xrightarrow{prg} e^{n_1} \xrightarrow{prg} \ldots \xrightarrow{prg} e^{n_2} \xrightarrow{prg} \ldots \xrightarrow{prg} e^{n_m} \xrightarrow{prg} \ldots \xrightarrow{prg} e^n,
\]

\[
0 \leq k,
\]

\[
e_0 \xrightarrow{prg} \ldots \xrightarrow{prg} e^k,
\]

\[
\text{Thrd}_{ck} \subseteq \text{Thrd}_{c0} \subseteq \text{Thrd},
\]

\[
pc_{T'}^0 = pc_{T'}^0,
\]

\[
\exists T' \in \text{Reg}(\tilde{x}_T^0),
\]

\[
\exists \tilde{x}' \in \gamma_{\text{var}}(\tilde{x}_0), \forall x \in \text{Var} : \forall T \in \text{Thrd} : ((\tilde{x}_0^0 x) T) \subseteq ((\tilde{x}' x) T),
\]

\[
\forall lck \in \text{Lck} : (\text{OWN}(\tilde{lck}) \neq \bot_{\text{thrd}} \Rightarrow (\text{STT}(\tilde{lck}) = \text{SIT}(\tilde{lck}) \land \\
\text{OWN}(\tilde{lck}) = \text{OWN}(\tilde{lck}) \land \\
\text{DL}(\tilde{lck}) \in \gamma_l(\text{DL}(\tilde{lck})) \land \\
\text{POWN}(\tilde{lck}) = \text{POWN}(\tilde{lck}) \land \\
\text{REL}(\tilde{lck}) \in \gamma_l(\text{REL}(\tilde{lck})) \land \\
\text{min}(\gamma_l(\text{DL}(\tilde{lck}))) = -\infty)) \land \\
(\text{OWN}(\tilde{lck}) = \bot_{\text{thrd}} \Rightarrow (\text{OWN}(\tilde{lck}) \neq \bot_{\text{thrd}} \Rightarrow (\text{STT}(\tilde{lck}) = \text{SIT}(\tilde{lck}) \land \\
\text{OWN}(\tilde{lck}) = T' \land \\
\text{SIT}(\tilde{lck}) = \text{unlocked} \land \\
\text{REL}(\tilde{lck}) \in \gamma_l(\text{REL}(\tilde{lck})))))
\]

\[
\forall i \in \{0, \ldots, n - 1\} \setminus \{n_1, n_2, \ldots, n_m\} : T' \notin \text{Thrd}_{ex}^{\tilde{x}_i},
\]

\[
\forall i \in \{n_1, n_2, \ldots, n_m\} : (T' \in \text{Thrd}_{ex}^{\tilde{x}_i} \land \text{OWN}(\tilde{lck}_i) \neq \bot_{\text{thrd}}),
\]

\[
T' \notin \text{Thrd}_{ex}^{\tilde{x}_i},
\]

\[
T' \in \text{Thrd}_{ex}^{\tilde{x}_i}, \text{ and}
\]

\[
\forall i \in \{0, \ldots, k \} : (\text{Thrd}_{ex}^{\tilde{x}_i} \neq 1 \lor \\
\{T \in \text{Thrd}_{ex}^{\tilde{x}_i} | \exists r \in \text{Reg}_T : \forall x \in \text{Var}_g : \\
\text{STM}(T, pc_{T'}^0) = [\text{load } r \text{ from } x]^{pc_{T'}^0} \} = \emptyset),
\]

where for all \(i \in \{0, \ldots, n\}\), \(\text{Thrd}_{ex}^{\tilde{x}_i}\) is as defined in Table 4.3, for all \(i \in \{0, \ldots, k\}\), \(\text{Thrd}_{ex}^{\tilde{x}_i}\) is as defined in Table 5.11, 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), then $\xrightarrow{pg}$ satisfies:

$$\forall c \in \langle [T, pc_T, \tilde{T}, t^0_T] \in \text{Thrd}, \tilde{x}, \xi \rangle \in \text{Conf} :$$

$$(c^0 \xrightarrow{pg} c \Rightarrow \exists \tilde{c} @ \langle [T, pc_T, \tilde{T}, t^0_T] \in \text{Thrd}, \tilde{x}, \xi \rangle \in \text{Conf} :$$

$$(\tilde{c}^k \xrightarrow{pg} \ldots \xrightarrow{pg} \tilde{c} \wedge$$

$$pc_T = pc_{\tilde{T}} \wedge$$

$$\tilde{T} \in \gamma(\tilde{T}) \wedge$$

$$t^0_T \in \gamma(t^0_T) \wedge$$

$$\exists \tilde{x}' \in \gamma_{\text{var}}(\tilde{x}) : (\forall x \in \text{Var} : ((x x) T') \subseteq ((x x) T')) \wedge$$

$$\forall lck \in \text{Lck} : (\text{OWN}(\tilde{lck}) = T' \Rightarrow (\text{STT}(\tilde{lck}) = \text{STT}(lck) \wedge$$

$$\text{OWN}(-lck) = \text{OWN}(lck) \wedge$$

$$\text{DL}(\tilde{lck}) \in \gamma(\text{DL}(lck)) \wedge$$

$$\text{OWN}(\tilde{lck}) = \text{OWN}(lck) \wedge$$

$$\text{REL}(\tilde{lck}) \in \gamma(\text{REL}(lck)) \wedge$$

$$\min(\gamma(\text{DL}(lck))) = -\infty))) \quad \square$$

PROOF. Assume that the valid concrete configurations (cf. Definition 4.4), abstract configurations, lock and thread

$$c^0 @ \langle [T, pc_T^0, z^0_T, t^0_T] \in \text{Thrd}, \tilde{x}^0, \xi^0 \rangle \in \text{Conf},$$

$$c^{n1} @ \langle [T, pc_T^{n1}, z^{n1}_T, t^{n1}_T] \in \text{Thrd}, \tilde{x}^{n1}, \xi^{n1} \rangle \in \text{Conf},$$

$$c^{n2} @ \langle [T, pc_T^{n2}, z^{n2}_T, t^{n2}_T] \in \text{Thrd}, \tilde{x}^{n2}, \xi^{n2} \rangle \in \text{Conf},$$

$$\vdots$$

$$c^{nm} @ \langle [T, pc_T^{nm}, z^{nm}_T, t^{nm}_T] \in \text{Thrd}, \tilde{x}^{nm}, \xi^{nm} \rangle \in \text{Conf},$$

$$c^n @ \langle [T, pc_T^n, z^n_T, t^n_T] \in \text{Thrd}, \tilde{x}^n, \xi^n \rangle \in \text{Conf},$$

$$\tilde{c}^0 @ \langle [T, pc_T^0, z^0_T, t^0_T] \in \text{Thrd}, \tilde{x}^0, \xi^0 \rangle \in \text{Conf},$$

$$\tilde{c}^k @ \langle [T, pc_T^k, z^k_T, t^k_T] \in \text{Thrd}, \tilde{x}^k, \xi^k \rangle \in \text{Conf},$$

$$lck' \in \text{Lck}, \text{ and }$$

$$T' \in \text{Thrd}_{\tilde{x}^k},$$

and the conditions hold.
are such that

\[
\text{STM}(T', pc_{T'}^0) = [\text{lock } lck']^{pc_{T'}^0},
\]

\[
0 \leq n_1 \leq n_2 \leq n_m \leq n,
\]

\[
c^0 \xrightarrow{prg} \ldots \xrightarrow{prg} c^{n_1} \xrightarrow{prg} \ldots \xrightarrow{prg} c^{n_2} \xrightarrow{prg} \ldots \xrightarrow{prg} c^{n_m} \xrightarrow{prg} \ldots \xrightarrow{prg} c^n,
\]

\[0 \leq k,
\]

\[
c^0 \xrightarrow{prg} \ldots \xrightarrow{prg} c^k,
\]

\[
\text{Thrd}_{\text{lk}} \subseteq \text{Thrd}_{\text{co}} \subseteq \text{Thrd},
\]

\[
\begin{align*}
p_c^0 & = p_c^0, \\
\tilde{z}_T^0 & \in \gamma_i(\tilde{r}_T^0), \\
t_v^0 & \in \gamma_i(\tilde{r}_T^0),
\end{align*}
\]

\[
\exists x' \in \gamma_i(\tilde{x}^0) : \forall x \in \text{Var} : \forall T \in \text{Thrd} : ((\tilde{x}^0 \ x) \ T) \subseteq ((\tilde{x}' \ x) \ T),
\]

\[
\forall \text{Lck} \in \text{Lck} : ((\text{OWN}(\tilde{l}^0 \ \text{Lck})) \neq \bot_{\text{thrd}} \Rightarrow (\text{STM}(\tilde{l}^0 \ \text{Lck}) = r \text{TT}(\tilde{l}^0 \ \text{Lck}) \land \\
\text{OWN}(\tilde{l}^0 \ \text{Lck}) = O\text{WN}(\tilde{l}^0 \ \text{Lck}) \land \\
\text{DL}(\tilde{l}^0 \ \text{Lck}) \in \gamma_i(\text{DL}(\tilde{l}^0 \ \text{Lck})) \land \\
\text{POWN}(\tilde{l}^0 \ \text{Lck}) = O\text{WN}(\tilde{l}^0 \ \text{Lck}) \land \\
\text{REL}(\tilde{l}^0 \ \text{Lck}) \in \gamma_i(\text{REL}(\tilde{l}^0 \ \text{Lck})) \land \\
\min(\gamma_i(\text{DL}(\tilde{l}^0 \ \text{Lck})))) = -(\infty)) \land \\
(\text{OWN}(\tilde{l}^0 \ \text{Lck}) = \bot_{\text{thrd}} \Rightarrow ((\text{OWN}(\tilde{l}^0 \ \text{Lck}) \neq \text{OWN}(\tilde{l}^0 \ \text{Lck}) \lor \\
(\text{OWN}(\tilde{l}^0 \ \text{Lck}) = T' \land \\
\text{STM}(\tilde{l}^0 \ \text{Lck}) = \text{unlocked} \land \\
t_v^0 + \text{TIME}(c^n, T') \in \gamma_i(\text{DL}(\tilde{l}^0 \ \text{Lck})) \land \\
\min(\gamma_i(\text{DL}(\tilde{l}^0 \ \text{Lck})))) = -(\infty)) \land \\
\text{POWN}(\tilde{l}^0 \ \text{Lck}) = \text{POWN}(\tilde{l}^0 \ \text{Lck}) \land \\
\text{REL}(\tilde{l}^0 \ \text{Lck}) \in \gamma_i(\text{REL}(\tilde{l}^0 \ \text{Lck}))))),
\]

\[
\forall i \in \{0, \ldots, n-1\} \ \{ n_1, n_2, \ldots, n_m \} : T' \not\in \text{Thrd}_{\text{exe}},
\]

\[
\forall i \in \{ n_1, n_2, \ldots, n_m \} : (T' \in \text{Thrd}_{\text{exe}} \land \text{OWN}(\tilde{l}^{n_i} \ \text{Lck}) \neq T'),
\]

\[
T' \in \text{Thrd}_{\text{exe}},
\]

\[
\forall i \in \{0, \ldots, k-1\} : T' \not\in \text{Thrd}_{\text{exe}},
\]

\[
T' \in \text{Thrd}_{\text{lk}}, \ \text{and}
\]

\[
\forall i \in \{0, \ldots, k\} : (|\text{Thrd}_{\text{exe}}| \neq 1 \lor \\
\{ T \in \text{Thrd}_{\text{exe}} | \exists r \in \text{Reg}_T : \exists x \in \text{Var} : \\
\text{STM}(T, pc_T^0) = [\text{load } r \ \text{from } x]^{pc_T^0} = \emptyset),
\]

where for all \( i \in \{0, \ldots, n\}, \ \text{Thrd}_{\text{exe}}^i \) is as defined in Table 4.3, for all \( i \in \{0, \ldots, k\}, \ \text{Thrd}_{\text{exe}}^i \) is as defined in Table 5.11, 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).

First note that:

- Since $\forall i \in \{0, \ldots, n - 1\} \setminus \{n_1, n_2, \ldots, n_m\} : T' \not\in \text{Thr}_i^\text{exe}$, $\forall i \in \{n_1, n_2, \ldots, n_m\} : (T' \in \text{Thr}_{i}^\text{exe} \land \text{OWN}(lck^i) \neq T')$ and \(\text{STM}(T', \text{pc}_{T'}^0) = [\text{lock} lck^i]^{\text{pc}_{T'}^0}\), it must be that $pc_{T'}^0 = pc_{T'}^0$, $z_{T'}^0 = z_{T'}^0$, $t_{T'}^0 = t_{T'}^0 + \text{TIME}(e_{n1}, T') + \text{TIME}(e_{n2}, T') + \ldots + \text{TIME}(e_{nm}, T')$ and $\forall lck \in \text{Lck} : (\text{OWN}(lck^i) = T' \Rightarrow \{0\} lck = \{0\} lck)$ (cf. Table 4.3).

- Since $\forall i \in \{0, \ldots, k - 1\} : T' \not\in \text{Thr}_i^\text{exe}$, it must be that $pc_{T'}^0 = pc_{T'}^0$, $z_{T'}^0 = z_{T'}^0$, $t_{T'}^0 = t_{T'}^0$, and $\forall lck \in \text{Lck} : (\text{OWN}(lck^0) = T' \Rightarrow \{0\} lck = \{0\} lck \land \min(\gamma_l(\text{DL}(lck))) = -\infty))$.

- Since $pc_{T'}^0 = pc_{T'}^0$, $z_{T'}^0 = z_{T'}^0$, $\forall lck \in \text{Lck} : (\text{OWN}(lck^0) = T' \Rightarrow \{0\} lck = \{0\} lck)$, $pc_{T'}^0 = pc_{T'}^0$, $z_{T'}^0 = z_{T'}^0$, $\forall lck \in \text{Lck} : (\text{OWN}(lck^0) = T' \Rightarrow \{0\} lck = \{0\} lck \land \min(\gamma_l(\text{DL}(lck))) = -\infty))$, $pc_{T'}^0 = pc_{T'}^0$, $z_{T'}^0 = z_{T'}^0$, $\forall lck \in \text{Lck} : (\text{OWN}(lck^0) = T' \Rightarrow \{0\} lck = \{0\} lck)$, it must be that:

  \[
  \begin{align*}
  &pc_{T'}^0 = pc_{T'}^0; \\
  &z_{T'}^0 = z_{T'}^0; \\
  &\forall lck \in \text{Lck} : (\text{OWN}(lck^0) = T' \Rightarrow (\text{STT}(lck^0) = \text{STT}(lck))) \\
  &\text{OWN}(lck^0) = \text{OWN}(lck) \land \\
  &\text{DL}(lck^0) \subseteq \gamma_l(\text{DL}(lck)) \land \\
  &\text{POWN}(lck^0) = \text{POWN}(lck) \land \\
  &\text{REL}(lck^0) \subseteq \gamma_l(\text{REL}(lck)) \land \\
  &\min(\gamma_l(\text{DL}(lck))) = -\infty))
  \end{align*}
\]

- Since $\exists x' \in \gamma_{\text{var}}(x^0) : \forall x \in \text{Var} : ((x^0 x) T') \subseteq ((x' x) T')$, $\forall i \in \{0, \ldots, n - 1\} \setminus \{n_1, n_2, \ldots, n_m\} : T' \not\in \text{Thr}_i^\text{exe}$, $\forall i \in \{n_1, n_2, \ldots, n_m\} : \text{OWN}(lck^i) \neq T'$, $\forall i \in \{0, \ldots, k - 1\} : T' \not\in \text{Thr}_i^\text{exe}$, $\text{STM}(T', pc_{T'}^0) = [\text{lock} lck^i]^{pc_{T'}^0}$ and TRIM is safe (Lemma 5.28), it must be that $\exists x' \in \gamma_{\text{var}}(x^0) : \forall x \in \text{Var} : ((x^0 x) T') \subseteq ((x' x) T')$. 

Since \( \forall i \in \{0,\ldots,k\} \) : \(|\text{Thrd}^a_{\text{exe}}| \neq 1 \lor \{T \in \text{Thrd}^a_{\text{exe}} \mid \exists r \in \text{Reg}_T : \\
exists x \in \text{Var}_g : \text{STM}(T,pc^a_{\text{exe}}) = [\text{load } r \text{ from } x]^ {pc^a_T} \} = \emptyset\), it must be that \( \forall i \in \{0,\ldots,k\} \) : \(\{T \in \text{Thrd}^a_{\text{exe}} \mid \exists r \in \text{Reg}_T : \exists x \in \text{Var}_g : \text{STM}(T,pc^a_T) = [\text{load } r \text{ from } x]^ {pc^a_T} \} \neq \emptyset \Rightarrow |\text{Thrd}^a_{\text{exe}}| = 1\). This means that if some thread in \( \text{Thrd}^a_{\text{exe}} \), where \( i \in \{0,\ldots,k\} \), performs a load-statement, there is only one single thread in \( \text{Thrd}^a_{\text{exe}} \); thus that thread performs the load-statement. It is then easy to see, from the definition of \( \text{Thrd}^a_{\text{exe}} \), that there cannot occur any other write than those represented by \( \tilde{\omega}^k \) such that it could affect the load-statement of the thread in \( \text{Thrd}^a_{\text{exe}} \) (cf. Assumption 5.51) – thus, it must be that \( \tilde{\omega}^k \) (and also all \( \tilde{\omega}^i \), where \( i \in \{0,\ldots,k\} \)) contains safe write history (cf. Definition 5.19).

Since, trivially, \( \forall lck \in \text{Lck} : \{T \in \text{Thrd}^a_{\text{exe}} \cap \text{Thrd}^a_{\text{ck}} | \text{STM}(T,pc^a_T) = [\text{lock } lck]^ {pc^a_T} \} \subseteq \{T \in \text{Thrd}^a_{\text{ck}} \mid \exists l \in \text{Lbl}_T : \text{STM}(T,l) = [\text{lock } lck]^l\} \), 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 \( \overrightarrow{\text{prg}} \) over-approximates the lock-owner assignment possible for \( \overrightarrow{\text{prg}} \) and \( T' \in \text{Thrd}^a_{\text{ck}} \), it must be that \( \overline{\text{OWN}}(\overrightarrow{\text{0}} lck') = T' \) is possible even if \( \overline{\text{OWN}}(\overrightarrow{\text{0}} lck') = \bot_{\text{thrd}} \), given that some other thread (i.e. not \( T' \)) does \( \text{lock } lck' \) 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 \( \text{STM}(T',pc^a_{\text{prg}}) = [\text{lock } lck']^ {pc^a_{\text{prg}}} \), \( \overline{\text{OWN}}(\overrightarrow{\text{0}} lck') = \bot_{\text{thrd}} \) and \( \overline{\text{OWN}}(\overrightarrow{\text{0}} lck') = T' \), it must be that \( \overline{\text{STM}}(\overrightarrow{\text{0}} lck') = \text{unlocked} \), \( it^a + \text{TIME}(c^a,T') \in \gamma_t(\overrightarrow{\text{DL}}(\overrightarrow{\text{0}} lck')) \) and \( \text{min}(\gamma_t(\overrightarrow{\text{DL}}(\overrightarrow{\text{0}} lck'))) = -\infty \) (Table 5.11, Algorithm 5.11 and Lemmas 5.54 and 5.56).

Since \( \text{STM}(T',pc^a_{\text{ck}}) = [\text{lock } lck']^ {pc^a_{\text{ck}}} \) and \( T' \in \text{Thrd}^a_{\text{exe}} \), it must be that \( \overline{\text{OWN}}(\overrightarrow{\text{0}} lck') = T' \).

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, \( \forall i \in \{0,\ldots,n-1\} : T' \notin \text{Thrd}^c_{\text{exe}} \). (Note that
$c^0$ can be chosen to be the first configuration satisfying $\text{OWN}(\emptyset, lck') = \bot_{\text{thrd}} \land 
abla c \in \text{Conf}:
(c^0 \xrightarrow{\text{prg}} c \Rightarrow \exists x \in \text{Var} : ((\exists x \ T') \subseteq ((\exists x' \ T')) \land
\forall lck \in \text{Lck} : (\text{OWN}(\emptyset, lck) = T' \Rightarrow
\text{STT}(\emptyset, lck) = \text{STT}(\emptyset, lck) \land
\text{OWN}(\emptyset, lck) = \text{OWN}(\emptyset, lck) \land
\text{DL}(\emptyset, lck) = \gamma(\text{DL}(\emptyset, lck)) \land
\text{POWN}(\emptyset, lck) = \text{POWN}(\emptyset, lck) \land
\text{REL}(\emptyset, lck) = \gamma(\text{REL}(\emptyset, lck)) \land
\min(\gamma(\text{DL}(\emptyset, lck))) = -\infty)))$

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 un$\text{lock}$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 $t^0_{T'} \leq t^0_{T'} = t^0_{T'} + \text{TIME}(c^{n_1}, T') + \text{TIME}(c^{n_2}, T') + \ldots + \text{TIME}(c^{n_m}, T') =$
\[ t^\text{stm}_{\text{acc}} + \text{TIME}(e^\text{stm}, T') \] (cf. Assumption 4.1 and Table 4.3).

Since the lemma holds for all \( j \) owners that acquire \( lck' \) while \( T' \) is waiting, and all other cases involving other locks, and \( \overrightarrow{\text{ps}} \) safely over-approximates the transitions described by \( \overrightarrow{\text{ps}} \) 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 \( \tilde{c}^0 \) and ending at \( \tilde{c}^k \)) 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 \( T' \in \text{Thrd}^\text{exe}_k \land \text{OWN}(\llbracket lck' \rrbracket) \neq T' \), \( \forall i \in \{ n_m + 1, \ldots, n - 1 \} : T' \not\in \text{Thrd}^j_i \) and \( T' \in \text{Thrd}^a_i \land \text{OWN}(\llbracket lck' \rrbracket) = \perp_{\text{thrd}} \land \text{OWN}(\llbracket lck' \rrbracket) = T' \) (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 \( c^m \overrightarrow{\text{ps}} \cdots \overrightarrow{\text{ps}} c^n' \overrightarrow{\text{ps}} \cdots \overrightarrow{\text{ps}} c^n \) and \( n_m < n' \leq n \). Thus, it must be that \( t^\text{stm}_{\text{acc}} + \text{TIME}(c^n, T') = t^a_{\text{acc}} \leq \text{REL}(\llbracket lck' \rrbracket) \leq t^a_{\text{acc}} + \text{TIME}(c^n, T') \) (cf. Assumption 4.1 and Table 4.3), where \( \text{REL}(\llbracket lck' \rrbracket) = \text{REL}(\llbracket lck' \rrbracket) \) and \( \text{REL}(\llbracket lck' \rrbracket) \in \gamma_{\text{DLLOCK}}(\llbracket lck' \rrbracket) \) (given the abstract trace from \( \tilde{c}^0 \) to \( \tilde{c}^k \) 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 \( t^a_{\text{acc}} + \text{TIME}(c^n, T') \in \gamma_{\text{DLLOCK}}(\llbracket lck' \rrbracket) \) (Lemma 5.54).

To show that ACC TIME is safe for this case, first note that \( T' \in \text{Thrd}^a_k \), \( \text{STM}(T', p\text{c}^k_{\text{acc}}) = [\text{lock} lck']^p_{\text{acc}} \) and \( \text{STM}(\llbracket lck' \rrbracket) = \text{unlock} \). Also note that since \( t^a_{\text{acc}} + \text{TIME}(c^n, T') \in \gamma_{\text{DLLOCK}}(\llbracket lck' \rrbracket) = \text{DL}(\llbracket lck' \rrbracket) = t^a_{\text{acc}} + \text{TIME}(c^n, T') \) (cf. Tables 4.2 and 4.3 since \( T' \) acquires \( lck' \) in a transition from \( c^n \) and \( \min(\gamma_{\text{DL}}(\llbracket lck' \rrbracket), \text{ABSTIME}(c^0, T')) \leq t^a_{\text{acc}} + \text{TIME}(c^n, T') \leq \max(\gamma_{\text{DL}}(\llbracket lck' \rrbracket), \text{ABSTIME}(c^0, T')) \) (cf. Assumptions 4.1 and 5.51), it must be the case that \( \text{ABSTIME}(c^0, T') \in \gamma_{\text{DL}}(\llbracket lck' \rrbracket) \) (cf. Assumptions 4.1 and 5.51), which means that there are three branches of Algorithm 5.12 that must be considered here. Note that this also means that \( \text{ABSTIME}(c^0, T') \in \gamma_{\text{DL}}(\llbracket lck' \rrbracket) \). For the sake of readability, let \( \tilde{c}^{k'n'} = \langle [T', p\text{c}^k_{\text{acc}}], \tilde{c}^k_{\text{acc}} \rangle \). Also let \( \tilde{c}^{k'n'}_{\text{acc}} \) be defined as \( \tilde{c}^{k'n'}_{\text{acc}} \) in Algorithm 5.12.

1. Since \( T' \) has been frozen while waiting to acquire \( lck' \), it can be the case that \( \tilde{c}^{k'n}_{\text{acc}} \), \( \text{ABSTIME}(\tilde{c}^{k'n}, T') \), \( \tilde{c}^{k'n}_{\text{acc}} \), \( \text{REL}(\llbracket lck' \rrbracket) \), where \( \tilde{c}^{k'n}_{\text{acc}} = \tilde{c}^{k'n}_{\text{acc}} \). (Note...
that this does not necessarily have to be the case, though.) Let \( \bar{c}' \) be any configuration derived before (i.e. \( \bar{c}' = \bar{c}^{kn'} \)) or inside the \texttt{while}-loop.

First note that it cannot be that ASB TIME(\( \bar{c}', T' \)) = \( \alpha \cap \{0\} \) and \( \bar{n}_{T'}^{at} \uparrow_t \text{ASB TIME}(\( \bar{c}', T' \)) \( \geq_t \text{REL}(\bar{k}^{kn'} \text{lck}') \) (cf. Assumptions 4.3 and 5.51). This means that the \texttt{while}-loop will eventually terminate. It does so when \( \bar{n}_{T'}^{at} \) is the last point in time that safely represents the situation that \( T' \) has not yet acquired \text{lck}'\;\text{since, at } \bar{n}_{T'}^{at} \uparrow_t \text{ASB TIME}(\( \bar{c}', T' \)), \( T' \) might have acquired \text{lck}' (i.e. \( \bar{n}_{T'}^{at} \uparrow_t \text{REL}(\bar{k}^{kn'} \text{lck}') \) and \( \bar{n}_{T'}^{at} \uparrow_t \text{ASB TIME}(\( \bar{c}', T' \)) \( \geq_t \text{REL}(\bar{k}^{kn'} \text{lck}') \)). In later references within this proof, the \( \bar{n}_{T'}^{at} \) obtained at the exit of the \texttt{while}-loop will be referred to as \( \bar{n}_{T'}^{\text{est}} \).

Since \( \bar{n}_{T'}^{\text{est}} \uparrow_t \text{REL}(\bar{k}^{kn'} \text{lck}') \) and \( \bar{n}_{T'}^{\text{est}} \uparrow_t \text{ASB TIME}(\( \bar{c}', T' \)) \( \geq_t \text{REL}(\bar{k}^{kn'} \text{lck}') \), it is easy to see that this branch will lead to an auxiliary configuration, \( \bar{c}^{k} \), such that \( \bar{c}^{k} \rightarrow_{\text{reg}} \bar{c}' \), for which \( \bar{k}^{kn'} \text{lck}' = \bar{k}^{kn'} \text{lck}' \); i.e. \( T' \) has not yet acquired \text{lck}'. The only difference for \( T' \) between \( \bar{c}^{k} \) and \( \bar{c}^{k} \) is that, in the latter, it has an advanced accumulated execution time (cf. Table 5.10). Since \( \bar{n}_{T'}^{\text{est}} \uparrow_t \text{ASB TIME}(\( \bar{c}', T' \)) \( \geq_t \text{REL}(\bar{k}^{kn'} \text{lck}') \), it is also easy to see that this branch of Algorithm 5.12 will not be taken when \text{ACCTIME} \text{is called based on } \bar{c}^{k} \text{. Note that it must be that } |\{n_1, n_2, \ldots, n_m\}| \geq 1 \text{ is equal to the number of iterations of the } \texttt{while}-\text{loop (Assumption 5.51 and Lemma 5.53). Thus, it is also easy to see that } \text{DL}(\bar{k}^{kn'} \text{lck}') \( \geq_t \bar{n}_{T'}^{\text{est}} \uparrow_t \text{ASB TIME}(\( \bar{c}^{k}, T' \)) \text{ since } \text{TIME}(\( \bar{c}', T' \)) \in \gamma_i(\text{ASB TIME}(\( \bar{c}^{k}, T' \))) \text{, where } i \in \{n_1, n_2, \ldots, n_m\} \text{ is the corresponding concrete configuration for which the } \texttt{while}-\text{loop terminates (Assumption 5.51). This means that for } \bar{c}^{k} \text{, one of the two last branches (considered in the next two bullets) of Algorithm } 5.12 \text{ will apply.}

2. First note that it must be that POWN(\( \bar{k}^{kn'} \text{lck}' \)) \neq T' since \( T' \) has been waiting for at least one other thread to release \text{lck}' before it is allowed to acquire it (cf. Tables 5.10 and 5.11 and the induction assumption). If, on the other hand, REL(\( \bar{k}^{kn'} \text{lck}' \)) \( \geq_t \bar{n}_{T'}^{at} \uparrow_t \text{ASB TIME}(\( \bar{c}^{k'}, T' \)), then the proof is equivalent to the corresponding part of the proof for Lemma 5.55 since \( \bar{n}_{T'}^{at} + \text{TIME}(\( \bar{c}', T' \)) = \text{DL}(\bar{n}_{T'}^{at} \text{lck}') \), \( \text{DL}(\bar{n}_{T'}^{at} \text{lck}') \in \gamma_i(\text{DL}(\bar{k}^{kn'} \text{lck}')) \) and \( \text{REL}(\bar{n}_{T'}^{at} \text{lck}') \in \gamma_i(\text{REL}(\bar{k}^{kn'} \text{lck}')) \)). Note that this also applies if \( \bar{n}_{T'}^{at} = \bar{n}_{T'}^{\text{est}} \) (i.e. if \( \bar{k}^{kn'} = \bar{k}^{kn} \)) since it must be that \( \bar{n}_{T'}^{at} \in \gamma_i(\bar{n}_{T'}^{est}) \), which follows from Assumption 5.51 and Lemma 5.52 based on 1 above.

3. If (\( \bar{n}_{T'}^{at} \uparrow_t \text{ASB TIME}(\( \bar{c}', T' \)) \) \( \bar{n}_{T'}^{est} \text{REL}(\bar{k}^{kn'} \text{lck}') \) \( = \bar{n}_{T'}^{est} \)), then let \( \bar{n}_{T'}^{est} = \bar{n}_{T'}^{at} \uparrow_t \)}
Chapter 5. Abstractly Interpreting PPL

146

\[ \mathcal{Taw} \approx \mathcal{Tan} \text{ (cf. Table 5.10).} \] Since \( \mathcal{Taw} \) does not necessarily have to be the case, though. Let \( \mathcal{ck} \) have not yet acquired \( \mathcal{lck} \) (Assumption 5.51). This means that for \( \mathcal{lck} \) taken when \( \mathcal{ACC~TIME} \) is called based on \( \mathcal{taw} \), i.e. \( T_k \mathcal{c} \). \( \mathcal{ck} \) can acquire \( \mathcal{lck} \) and \( \mathcal{c} \) be either \( \mathcal{ck} \) or \( \mathcal{ck} \). To illustrate this, let \( \mathcal{ck} \) be any configuration derived before (i.e. \( \mathcal{c} = \mathcal{ck} \) or \( \mathcal{c} = \mathcal{ck} \)) or inside the \textbf{repeat}-loop (and the corresponding for \( \mathcal{taw} \)), which will now be considered. Note that \( \mathcal{taw} = \mathcal{T} \), is used to exit the loop in case \( \mathcal{DL}(\mathcal{ck}) \approx_\mathcal{Taw} \mathcal{Taw} \). ABSTIME(\( \mathcal{c}, \mathcal{T'} \)) or \( 0 \in \mathcal{y}_\mathcal{T}(\text{ABSTIME}(-\mathcal{c}, \mathcal{T'})) \), where the latter case means that a \( \mathcal{taw} \), such that \( \mathcal{REL}(\mathcal{ck}) \approx_\mathcal{Taw} \mathcal{Taw} \) cannot be derived (cf. Assumption 5.51).

(a) If \( \mathcal{DL}(\mathcal{ck}) \approx_\mathcal{Taw} \mathcal{Taw} \), ABSTIME(\( \mathcal{c}, \mathcal{T'} \)), then it must be that \( \mathcal{taw} \) is a safe estimation of the last point in time when \( \mathcal{T'} \) can acquire \( \mathcal{lck} \) since \( \mathcal{taw} + \mathcal{TIME}(\mathcal{c}, \mathcal{T'}) = \mathcal{DL}(\mathcal{ck}) \) and \( \mathcal{T'} \) acquires \( \mathcal{lck} \) in a transition from \( \mathcal{c} \) (cf. Assumption 5.51 and Lemma 5.53 which means that the total number of iterations of the \textbf{repeat}-loop, and possibly the \textbf{while}-loop from 1, must be greater than or equal to \( \{n_1, n_2, \ldots, n_m \} \)). Thus, it must be that \( \mathcal{taw} + \mathcal{TIME}(\mathcal{c}, \mathcal{T'}) \in \gamma_\mathcal{T}(\mathcal{DL}(\mathcal{ck})) \) and \( \mathcal{T'} \) acquires \( \mathcal{lck} \) in a transition from \( \mathcal{c} \) (cf. Assumption 5.51 and Lemma 5.53).

(b) If \( 0 \in \gamma_\mathcal{T}(\text{ABSTIME}(\mathcal{c}, \mathcal{T'})) \), then it must obviously be that \( \mathcal{Taw} \), ABSTIME(\( \mathcal{c}, \mathcal{T'} \)), where \( \mathcal{t} = (\mathcal{DL}(\mathcal{ck})) \) and \( \mathcal{ck} = \langle [\mathcal{T}, \mathcal{PC}^{\mathcal{Taw}}], \mathcal{ck}, \mathcal{T'} \rangle \), is a safe approximation of the last point in time when \( \mathcal{T'} \) can (or rather, will) acquire \( \mathcal{lck} \) (cf. Assumption 5.51 and Lemma 5.53). Thus, it must be that \( \mathcal{taw} + \mathcal{TIME}(\mathcal{c}, \mathcal{T'}) \in \gamma_\mathcal{T}(\mathcal{DL}(\mathcal{ck})) \) since \( \mathcal{T'} \) acquires \( \mathcal{lck} \) in a transition from \( \mathcal{c} \).
\[
\text{TIME}(c^n, T') = \text{DL}(\| lck') = -\infty \quad \text{and} \quad T' \text{ acquires } lck' \text{ in a transition from } c^n \text{ (cf. Assumption 5.51 and Lemma 5.53 which means that the total number of iterations of the } \text{repeat-loop, and possibly the while-loop from 1, must be greater than or equal to } |\{n_1, n_2, \ldots, n_m, n\}|.
\]

Thus, it has been shown that \( t^n_T + \text{TIME}(c^n, T') \in \gamma_i(\text{ACCTIME}(c @ T', \tau_t, n_T, T_{\text{Thrd}_{ex}}, T')) \), for both the case that \( c' \) is \( c^{k_{prg}} \) (if \( \tilde{T}_{prg}^k \uparrow_t \text{ABSTIME}(c^{k_{prg}}, T') \not\preceq_t \text{REL}(\| lck') \)) and \( c' \) is \( c^{k_{dl}} \) (if \( \tilde{T}_{dl}^k \uparrow_t \text{ABSTIME}(c^{k_{dl}}, T') \not\preceq_t \text{REL}(\| lck') \)), where \( c^{k_{prg}} \) and \( c^{k_{dl}} \) are as defined above. If \( \tilde{T}_{prg}^k \uparrow_t \text{ABSTIME}(c^{k_{prg}}, T') \not\preceq_t \text{REL}(\| lck') \), it is easy to see that

\[
\begin{align*}
\forall lck \in \text{Lck} : (\text{OWN}(\| lck) & = T') \implies (\text{STT}(\| lck) = \text{STT}(\| lck) \land \\
\text{OWN}(\| lck) & = \text{OWN}(\| lck) \land \\
\text{DL}(\| lck) & = \gamma_i(\text{DL}(\| lck)) \land \\
\text{POWN}(\| lck) & = \text{POWN}(\| lck) \land \\
\text{REL}(\| lck) & = \gamma_i(\text{REL}(\| lck)) \land \\
\text{min}(\gamma_i(\text{DL}(\| lck))) & = -\infty)
\end{align*}
\]

since, for \( T' \), the accumulated execution time is the only state affected by the transition \( c^k \xrightarrow{prg} c^{k_{prg}} \) (cf. Table 5.10; this means that, e.g. \( \bar{k}_{prg} lck' = \bar{k}_{dl} lck' \)) and \( \text{TRIM} \) is safe (Lemma 5.28). Thus, for both transition sequences described by \( c^k \xrightarrow{prg} c^{k_{prg}} \xrightarrow{prg} c \) and \( c^k \xrightarrow{prg} c \), where \( c @ ([T, pc_T', \tau_T, \tilde{T}_{prg}^k, T_{\text{Thrd}_{ex}}, T]) \in \text{Conf} \), for the two different cases \( \tilde{T}_{prg}^k \uparrow_t \text{ABSTIME}(c^{k_{prg}}, T') \not\preceq_t \text{REL}(\| lck') \) and \( \tilde{T}_{prg}^k \uparrow_t \).
Thus, it has been shown that

\[ \text{ABSTIME}(\varepsilon^{k'}, T') \not\preceq_{\text{rel}} \text{REL}(\varepsilon^{k'}, lck'), \] respectively, it must be that

\[
\begin{align*}
\text{pc}_{T'} &= \text{pc}_{c'}, \\
\exists x' &\in \gamma_{\text{reg}}(\bar{x}_{T'}), \\
\bar{t}'_{T'} &\in \gamma_{l}(\bar{t}'_{T}), \\
\exists x' &\in \gamma_{\text{var}}(\bar{x}_{T}) : (\forall x \in \text{Var} : ((\bar{x}_{T} x) T') \subseteq ((\bar{x}'_{T} x) T')) \text{ and} \\
\forall lck &\in \text{Lck} : (\text{OWN}(\bar{lck}) = T' \Rightarrow (\text{STT}(\bar{lck}) = \text{ST}(\bar{lck}) \land \\
&\qquad \text{OWN}(\bar{lck}) = \text{OWN}(\bar{lck}) \land \\
&\qquad \text{DL}(\bar{lck}) \in \gamma_{l}(\bar{DL}(\bar{lck}))) \land \\
&\qquad \text{POWN}(\bar{lck}) = \text{POWN}(\bar{lck}) \land \\
&\qquad \text{REL}(\bar{lck}) \in \gamma_{l}(\text{REL}(\bar{lck})) \land \\
&\qquad \min(\gamma_{l}(\bar{DL}(\bar{lck}))) = -\infty)
\end{align*}
\]

where \(\varepsilon^{a} \rightarrow_{\text{as}} c\) for some \(c @ ([T, pc_{T}, t_{T}^{a}]_{T\in\text{Thrd}}, x, []) \in \text{Conf}\), since \(\sim_{\text{as}}\) is a safe approximation of \(\rightarrow_{\text{as}}\) (Lemma 5.50), \(\text{DL}(\bar{lck'}) = t_{T}^{a} + \text{TIME}(\varepsilon^{a}, T')\) (Table 4.2) and TRIM is safe (Lemma 5.28). But then the lemma holds.

Lemma 5.59 states that \(\sim_{\text{prg}}\) 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 \(\text{Thrd}_{\text{exe}}\) 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 161).

**Lemma 5.59 (Soundness of \(\sim_{\text{prg}}\), final state):**

If the valid concrete configurations (cf. Definition 4.4) \(\varepsilon^{0} @ \langle [T, pc_{T}^{0}, t_{T}^{0}]_{T\in\text{Thrd}_{c}}, x^{0}, l^{0} \rangle \in \text{Conf}\) and \(\varepsilon^{a} @ \langle [T, pc_{T}^{a}, t_{T}^{a}]_{T\in\text{Thrd}_{c}}, x^{a}, l^{a} \rangle \in \text{Conf}\) and the abstract configuration \(\hat{\varepsilon}^{0} @ \langle [T, pc_{T}^{\hat{0}}, t_{T}^{\hat{0}}]_{T\in\text{Thrd}_{c}}, \hat{x}^{0}, \hat{l}^{0} \rangle \in \hat{\text{Conf}}\), are such that \(\varepsilon^{0} \in \gamma_{\text{conf}}(\hat{\varepsilon}^{0}), \forall lck \in \text{Lck} : \min(\gamma_{l}(\text{DL}(\bar{lck}))) = -\infty\),

\(\forall lck \in \text{Lck} : \min(\gamma_{l}(\text{DL}(\bar{lck}))) = -\infty\).
\( \forall T \in \text{Thrd}_c : \text{STM}(T, pc^c_T) = [\text{halt}]^{pc^c_T} \)
\[\Rightarrow\]
\( (\exists \tilde{c}^k \in \langle \text{Conf} \rangle \text{Thrd}_c^c, \tilde{x}^k, \tilde{t}^k \rangle \in \text{Conf} : (c^0 \xrightarrow{prg} \ldots \xrightarrow{prg} c^k) \wedge \\
(\forall i \in \{0, \ldots, k-1\} : (|\text{Thrd}^c_{exe}| \neq 1 \vee \\
\{T \in \text{Thrd}^c_{exe} | \exists r \in \text{Reg}_T : \exists x \in \text{Var}_r : \\
\text{STM}(T, pc^c_T) = [\text{load } r \text{ from } x]^{pc^c_T} = \emptyset \}) \wedge \\
(\forall T \in \text{Thrd}_c : (pc^n_T = pc^c_T \wedge \\
\tilde{t}^n_T \in \gamma_{\text{reg}}(\tilde{t}^c_T) \wedge \\
\tilde{t}^c_T \in \gamma_i(\tilde{t}^c_T) \wedge \\
\exists \tilde{x} \in \gamma_{\text{Var}}(\tilde{x}^k) : (\forall x \in \text{Var} : ((\tilde{x}^n x) T) \subseteq ((\tilde{x}^c x) T)) \wedge \\
\forall lck \in \text{Lck} : (\text{STT}(\tilde{l}^n lck) = \text{STT}(\tilde{l}^c lck) \wedge \\
\text{OWN}(\tilde{l}^n lck) = \text{OWN}(\tilde{l}^c lck) \wedge \\
\text{DL}(\tilde{l}^n lck) \in \gamma_i(\tilde{DL}(\tilde{l}^c lck)) \wedge \\
\text{POWN}(\tilde{l}^n lck) = \text{POWN}(\tilde{l}^c lck) \wedge \\
\text{REL}(\tilde{l}^n lck) \in \gamma_i(\text{REL}(\tilde{l}^c lck)) \wedge \\
\min(\gamma_i(\text{DL}(\tilde{l}^c lck))) = -\infty)))) \\
\]

where, for all \( i \in \{0, \ldots, k-1\} \), \( \text{Thrd}^c_{exe} \) is as defined in Table 5.11, 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).

PROOF. Assume that the valid (cf. Definition 4.4) concrete configurations \( c^0 @\langle [T, pc^0_T, \tilde{x}^0_T, \tilde{t}^0_T] \rangle_{T \in \text{Thrd}_c^c}, \tilde{x}^0, \tilde{t}^0 \rangle \in \text{Conf} \) and \( c^n @\langle [T, pc^n_T, \tilde{x}^n_T, \tilde{t}^n_T] \rangle_{T \in \text{Thrd}_c^c}, \tilde{x}^n, \tilde{t}^n \rangle \in \text{Conf} \) and the abstract configuration \( \tilde{c}^0 @\langle [T, pc^0_T, \tilde{x}^0_T, \tilde{t}^0_T] \rangle_{T \in \text{Thrd}_c^c}, \tilde{x}^0, \tilde{t}^0 \rangle \in \text{Conf} \) are such that \( c^0 \in \gamma_{\text{conf}}(c^0) \), \( c^0 \xrightarrow{prg} \ldots \xrightarrow{prg} c^n \), \( \forall lck \in \text{Lck} : \\
\min(\gamma_i(\text{DL}(\tilde{l}^c lck))) = -\infty \) and \( \forall T \in \text{Thrd}_c^c : \text{STM}(T, pc^c_T) = [\text{halt}]^{pc^c_T} \).

Note that since \( \forall T \in \text{Thrd}_c^c : \text{STM}(T, pc^c_T) = [\text{halt}]^{pc^c_T} \), 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 \( \xrightarrow{prg} \) 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 \( c^f @ \langle [T, p_c^f, t_f^f, t_l^f] | T \in \text{Thrd}_c; \tilde{x}^f, \tilde{l}^f \rangle \in \text{Conf} \), \( c^g @ \langle [T, p_c^g, t_f^g, t_l^g] | T \in \text{Thrd}_c; \tilde{x}^g, \tilde{l}^g \rangle \in \text{Conf} \), \( c^h @ \langle [T, p_c^h, t_f^h, t_l^h] | T \in \text{Thrd}_c; \tilde{x}^h, \tilde{l}^h \rangle \in \text{Conf} \) and \( T' \in \text{Thrd}_c \) such that

\[
c^f \xrightarrow{pg} \cdots \xrightarrow{pg} c^g \wedge 0 \leq f < g \leq n \wedge
\forall h \in \{f, \ldots, g-2\} : (T' \not\in \text{Thrd}_{cex}^h \vee
\exists lck \in \text{Lck} : (\text{STM}(T', p_{c_{ch}^h}) = [\text{lock lck}]^{p_{c_{ch}^h}} \wedge \text{OWN}(\tilde{l}^{h+1} \text{ lck}) \neq T')) \wedge
T' \in \text{Thrd}_{cex}^{g-1} \wedge
\forall lck \in \text{Lck} : \text{STM}(T', p_{c_{ch}^{g-1}}) = [\text{lock lck}]^{p_{c_{ch}^{g-1}}} \Rightarrow \text{OWN}(\tilde{l}^{g} \text{ lck}) = T' \wedge
p_{c_{ch}^f} = p_{c_{ch}^g} \wedge
\tilde{x}^g_{T'} \in \gamma_{reg}(\tilde{x}^f_{T'}) \wedge
\tilde{t}_{ch}^g \in \gamma_i(\tilde{t}_{ch}^f) \wedge
\exists x' \in \gamma_{var}(\tilde{x}^f_{T'}) : \forall x \in \text{Var} : \forall T \in \text{Thrd} : (\langle x^f \, T \rangle \subset \langle x^g \, T \rangle) \wedge
\forall lck \in \text{Lck} : ((\text{OWN}(\tilde{l}^f lck) \neq \bot_{thrd}) \Rightarrow (\text{STT}(\tilde{l}^f lck) = s^{\text{STT}(\tilde{l}^f lck)} \wedge
\text{OWN}(\tilde{l}^f lck) = \text{OW}(\tilde{l}^f lck) \wedge
\text{DL}(\tilde{l}^f lck) \in \gamma_i(\text{DL}(\tilde{l}^f lck)) \wedge
\text{POWN}(\tilde{l}^f lck) = \text{POW}(\tilde{l}^f lck) \wedge
\text{REL}(\tilde{l}^f lck) \in \gamma_i(\text{REL}(\tilde{l}^f lck)) \wedge
\text{min}(\gamma_i(\text{DL}(\tilde{l}^f lck))) = -\infty) \wedge
(\text{OWN}(\tilde{l}^f lck) = \bot_{thrd}) \Rightarrow ((\text{OWN}(\tilde{l}^f lck) = \text{OW}(\tilde{l}^f lck) \vee
(\text{OWN}(\tilde{l}^f lck) = T' \wedge
\text{STT}(\tilde{l}^f lck) = \text{unlocked} \wedge
\text{t}_{ch}^g \in \gamma_i(\text{DL}(\tilde{l}^f lck)) \wedge
\text{min}(\gamma_i(\text{DL}(\tilde{l}^f lck))) = -\infty) \wedge
\text{POWN}(\tilde{l}^f lck) = \text{POW}(\tilde{l}^f lck) \wedge
\text{REL}(\tilde{l}^f lck) \in \gamma_i(\text{REL}(\tilde{l}^f lck)))))) \wedge
\forall h \in \{f, \ldots, g-1\} : (|\text{Thrd}_{cex}^h| \neq 1 \vee
\{T \in \text{Thrd}_{cex}^h | \exists r \in \text{Reg}_T : \exists x \in \text{Var}_r : \text{STM}(T, p_{c_{ch}^h} = [\text{load } r \text{ from } x]^{p_{c_{ch}^h}} \} = \emptyset)
\]

where \( \text{Thrd}_{cex} \) is as defined in Table 4.3. This is the induction assumption. Then it is easy to see that there exists a \( c^i @ \langle [T, p_c^i, t_f^i, t_l^i] | T \in \text{Thrd}_c; \tilde{x}^i, \tilde{l}^i \rangle \in \text{Conf} \).
Conf, such that
\[
\overline{c}^i \xrightarrow{\text{prg}} \ldots \xrightarrow{\text{prg}} \overline{c}^j \land \\
\text{pc}_{T'}^g = \text{pc}_{T'}^i \land \\
T_{g}^i \in \gamma_{\text{reg}}(\overline{c}^j) \land \\
T_{g}^i \subset \gamma_{\text{reg}}(\overline{c}^j) \land \\
\exists \overline{x} ' \in \gamma_{\text{var}}(\overline{x}^i) : (\forall x \in \text{Var} : ((\overline{x}^i x) T') \subseteq ((\overline{x} ' x) T')) \land \\
\forall \text{lck} \in \text{Lck} : ((\text{OWN}(\overline{lck}) = T' \lor \\
\text{OWN}(\overline{lck}) = T') \Rightarrow (\text{STT}(\overline{lck}) = \text{STT}(\overline{lck})) \land \\
\text{OWN}(\overline{lck}) = \text{OWN}(\overline{lck}) \land \\
\text{DL}(\overline{lck}) \in \gamma_{\text{DL}}(\overline{lck}) \land \\
\text{POWN}(\overline{lck}) = \text{POWN}(\overline{lck}) \land \\
\text{REL}(\overline{lck}) \in \gamma_{\text{REL}}(\overline{lck}) \land \\
\text{min}(\gamma_{\text{DL}}(\overline{lck})) = -\infty))
\]

(Lemmas 5.56, 5.57 and 5.58). Note that even if for some lock, lck \in \text{Lck}, T' issues 1\text{ock} 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 lck in c^g and \overline{c}^i (cf. Lemmas 5.57 and 5.58).

Now consider the base case for the induction part of the proof. Since \overline{c}^0 \in \gamma_{\text{conf}}(\overline{c}^0), \forall \text{lck} \in \text{Lck} : \min(\gamma_{\text{DL}}(\overline{0} lck))) = -\infty and \overline{c}^0 is valid, it is easy to see that

\[
\forall T \in \text{Thrd}_c : (\text{pc}_T^0 = \text{pc}_T^0 \land \\
\overline{x}^0 \in \gamma_{\text{reg}}(\overline{x}^0) \land \\
T_{g}^0 \in \gamma_{\text{reg}}(\overline{x}^0) \land \\
\exists \overline{x} ' \in \gamma_{\text{var}}(\overline{x}^0) : (\forall x \in \text{Var} : ((\overline{x}^0 x) T) \subseteq ((\overline{x} ' x) T')) \land \\
\forall \text{lck} \in \text{Lck} : (\text{STT}(\overline{0} lck) = \text{STT}(\overline{0} lck) \land \\
\text{OWN}(\overline{0} lck) = \text{OWN}(\overline{0} lck) \land \\
\text{DL}(\overline{0} lck) \in \gamma_{\text{DL}}(\overline{0} lck) \land \\
\text{POWN}(\overline{0} lck) = \text{POWN}(\overline{0} lck) \land \\
\text{REL}(\overline{0} lck) \in \gamma_{\text{REL}}(\overline{0} lck) \land \\
\text{min}(\gamma_{\text{DL}}(\overline{0} lck))) = -\infty)
\]

which means that as long as \forall h \in \{0, \ldots, k - 1\} : \{\overline{\text{Thrd}}_{\text{exe}} \not= 1 \lor \{T \in \overline{\text{Thrd}}_{\text{exe}} \mid \exists r \in \text{Reg}_T : \exists x \in \text{Var}_g : \text{STM}(T, \text{pc}_T^h) = [\text{load} r \text{ from } x]^{\text{pc}_T^h} \} = 0\}, the induction holds for all threads in \overline{\text{Thrd}}_c.
Note that by definition, \( \llbracket^n \text{lck} = \llbracket^0 \text{lck} \) (cf. Tables 4.2 and 4.3) and \( \llbracket^k \text{lck} = \llbracket^0 \text{lck} \) (cf. Tables 5.10 and 5.11) if \( \text{lck} \) is never acquired by any thread, or never released by its initially owning thread (i.e. the owner of \( \text{lck} \) in \( c^0 \) and \( \bar{c}^0 \), 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.
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, $\text{ABS EXE} : (P(C\text{\textcopyright Conf}) \times \text{Time}) \rightarrow (P(C\text{\textcopyright Conf}) \times P(C\text{\textcopyright Conf}) \times P(C\text{\textcopyright Conf}))$, 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, $\text{\tilde{C}} \in P(C\text{\textcopyright Conf})$ and $\text{\tilde{t}}_{to} \in \text{Time}$, is a set of abstract initial configurations (i.e. states) and a timeout value. A configuration, fetched from the worklist, $\text{\tilde{C}_w}$, will not be further considered if the timeout value is exceeded by the accumulated execution times of all threads in that...
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, $\text{ABSEXE} : (\mathcal{P}(\text{Conf}) \times \text{Time}) \rightarrow (\mathcal{P}(\text{Conf}) \times \mathcal{P}(\text{Conf}) \times \mathcal{P}(\text{Conf}))$, 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 $\tilde{t}_{to} \in \text{Time}$, 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 $c_0 \in \text{Conf}$) 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, $c_q \in \text{Conf}$, 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, $\tilde{c}_w \in \text{Conf}$. 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 $\tilde{c}_1, \tilde{c}_2, \ldots, \tilde{c}_{w-1} \in \text{Conf}$ because of problem 1, defined in Chapter 5 on page 109. 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 $\text{Thrd}_{\text{exe}}$, given some considered configuration from the worklist, $\tilde{C}^w$, $\text{ABSEX}E$ 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 5.11). 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, $\tilde{\gamma}_{prg}$ is used to derive a set of succeeding configurations, which are then added to the worklist. This strategy addresses problem 2, defined on page 109.

Problem 3, defined on page 109, is partly addressed in the definition of $\tilde{\gamma}_{prg}$ (cf. Table 5.11), as discussed in the previous chapter. $\text{ABSEX}E$ fully addresses
Algorithm 6.1 Abstract execution

1: function ABSEXE($\tilde{C}, \tilde{t}_0$)
2: $\tilde{C}_w \leftarrow \tilde{C}$; $\tilde{C}_f \leftarrow \emptyset$; $\tilde{C}_d \leftarrow \emptyset$; $\tilde{C}^t \leftarrow \emptyset$
3: while $\tilde{C}_w \neq \emptyset$ do
4: $\tilde{c} @ ([T, pc_T, \tilde{t}_T, \tilde{t}_d^T]_{T \in \text{Thr}_{d}}, \tilde{x}, \tilde{y}) \leftarrow \text{CHOOSE}(\tilde{C}_w)$
5: $\tilde{C}_w \leftarrow \tilde{C}_w \setminus \{\tilde{c}\}$
6: if ISFINAL($\tilde{c}$) then
7: $\tilde{C}_f \leftarrow \tilde{C}_f \cup \{\tilde{c}\}$
8: else if ISDEADLOCK($\tilde{c}$) then
9: $\tilde{C}_d \leftarrow \tilde{C}_d \cup \{\tilde{c}\}$
10: else if ISTIMEOUT($\tilde{c}, \tilde{t}_0$) then
11: $\tilde{C}^t \leftarrow \tilde{C}^t \cup \{\tilde{c}\}$
12: else if ISVALID($\tilde{c}, \tilde{t}_0$) then
13: $\text{Thr}_{\text{load}}^\text{exe} \leftarrow \text{EXELOADTHR}_{\text{exe}}(\tilde{c})$
14: if $\text{Thr}_{\text{load}}^\text{exe} \neq \emptyset \land |\text{EXE}_{\text{exe}}(\tilde{c})| > 1$ then
15: $(\tilde{r}^\text{exe}_T)_{T \in \text{Thr}_{\text{load}}^\text{exe}} \leftarrow (\hat{\tilde{r}}^\text{exe}_T, \text{ABSTIME}(\tilde{c}, T), T \in \text{Thr}_{\text{load}}^\text{exe})$
16: for all $T \in \text{Thr}_{\text{load}}^\text{exe}$ do
17: $x \leftarrow \text{GETVARLOAD}(\text{STM}(T, pc_T))$
18: $r \leftarrow \text{GETREGLOAD}(\text{STM}(T, pc_T))$
19: $\tilde{c}^t \leftarrow ([T, pc_T, \tilde{t}_T, \tilde{t}_d^T]_{T \in \text{Thr}_{d}} \setminus \{T\}, \tilde{x}, \tilde{y})$
20: $(\tilde{C}_f^t, \tilde{C}_d^t, \tilde{C}_w^t) \leftarrow \text{ABSEXE}((\tilde{c}^t), \tilde{r}^\text{exe}_T, T \in \text{Thr}_{d} \cup \{\tilde{c}\})$
21: $\tilde{t}_T \leftarrow \tilde{t}_T + \tilde{t}_T [r \mapsto \tilde{v}_{\text{val}}]$
22: for all $(\tilde{T}, \tilde{x}', \tilde{y}') \in \tilde{C}_f^t \cup \tilde{C}_d^t \cup \tilde{C}_w^t \cup \{\tilde{c}\}$ do
23: $\tilde{x}' \leftarrow \tilde{x}'[r \mapsto (\tilde{T} T \mapsto \tilde{v}_{\text{val}}) \text{READ}(\tilde{x}', x, T, \tilde{r}^\text{exe}_T)]$
24: end for
25: end for
26: $(\{pc_T\}_{T \in \text{Thr}_{\text{load}}^\text{exe}}) \leftarrow (\{pc_T + 1\}_{T \in \text{Thr}_{\text{load}}^\text{exe}})$
27: $(\tilde{r}^\text{exe}_T)_{T \in \text{Thr}_{\text{load}}^\text{exe}} \leftarrow (\hat{\tilde{r}}^\text{exe}_T)_{T \in \text{Thr}_{\text{load}}^\text{exe}}$
28: $\tilde{C}_w \leftarrow \tilde{C}_w \cup \{([T, pc_T, \tilde{t}_T, \tilde{t}_d^T]_{T \in \text{Thr}_{d}}, \tilde{x}, \tilde{y})\}$
29: else
30: $\tilde{C}_w \leftarrow \tilde{C}_w \cup \{\tilde{c}' \in \text{Conf} | \tilde{c} \xrightarrow{\text{exe}} \tilde{c}'\}$
31: end if
32: end if
33: end while
34: return $(\tilde{C}_f, \tilde{C}_d, \tilde{C}^t)$
35: end function
the problem by collecting all the possible transitions (i.e. resulting configurations) and adding them to the worklist. Problem 4, defined on page 110, is addressed by identifying deadlocked configurations and aborting their transitions.

It should be noted that absExe, as defined here, might not terminate for all possible inputs. This matter will be further discussed in Section 8.3.

A configuration is said to be in the final state if all threads are issuing the 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 given by the timeout, $\tau_{to}$, 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 deadlock 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 assignment deadline, reaches a configuration with possibly valid concrete counterparts if the owner issues a lock-statement on (i.e. acquires) the lock before the expiration of the deadline.

If a concrete configuration that is abstracted by some configuration in $\tilde{C}$ could be executed into a final configuration, absExe($\tilde{C}, \tau_{to}$) 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 $\tau_{to}$ for the corresponding abstract transition sequence (whenever the algorithm actually terminates), and thus, $\tilde{C}^f \neq \emptyset$ (cf. Theorem 6.8 on page 168).

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, absExe($\tilde{C}, \tau_{to}$) will either find the corresponding abstract situation (i.e. $\tilde{C}^d \neq \emptyset$) or reach a timeout for the corresponding abstract transition sequence (whenever the algorithm actually terminates), and thus, $\tilde{C}^f \neq \emptyset$ (cf. Theorem 6.8).

It is also the case that if $\tilde{C}^d \cup \tilde{C}^f = \emptyset$ (whenever the algorithm actually terminates), then all concrete configurations represented by the abstract configurations 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).
\textsc{absexe}(\tilde{C}, \tilde{t}_{\text{to}}) \text{ 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}, \text{ up until } \tilde{t}_{\text{to}}, \text{ which corresponds to the concrete time-point } t_{\text{to}} = \max(y(\tilde{t}_{\text{to}})) \text{, 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 \textsc{absexe} are further discussed here. \textsc{choose}(S), defined in Algorithm 6.2, gives a deterministically chosen element from the set S.

\begin{algorithm}
\caption{Choose an element}
\begin{algorithmic}[1]
\Function{choose}{S}
\State \textbf{Require: } S \neq \emptyset
\State \Return a deterministically chosen elements of S
\EndFunction
\end{algorithmic}
\end{algorithm}

Given a graph, \((V, E)\), \textsc{cycle}(V, E), as defined in Algorithm 6.3, means that \((V, E)\) contains at least one cycle (Lemma 6.1).

\begin{algorithm}
\caption{Determine if graph has cycles}
\begin{algorithmic}[1]
\Function{cycle}{V, E}
\State \(V' \leftarrow V\)
\State \(E' \leftarrow E\)
\While{\(V' \neq \emptyset\)}
\State \(V'' \leftarrow \{v \in V' \mid \not\exists v' \in V' : (v', v) \in E'\}\)
\If{\(V'' = \emptyset\)}
\State \Return \text{true}
\Else
\State \(E' \leftarrow E' \setminus \{(v, v') \in E' \mid v \in V'' \land v' \in V'\}\)
\State \(V' \leftarrow V' \setminus V''\)
\EndIf
\EndWhile
\State \Return \text{false}
\EndFunction
\end{algorithmic}
\end{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 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, \(\bar{c} \in \text{Conf}, \text{EXEThrd}(\bar{c})\), as defined in Algorithm 6.4, is an over-approximation of \(\text{Thrd}_{\text{exe}}\) as defined in Table 5.11 (Lemma 6.2).

Lemma 6.2 (Soundness of EXEThrd):
Given \(\bar{c} \in \text{Conf}, \text{Thrd}_{\text{exe}}^\bar{c} \subseteq \text{EXEThrd}(\bar{c})\), where \(\text{Thrd}_{\text{exe}}^\bar{c}\) is as defined in Table 5.11.

PROOF. Based on \(\bar{c} \in \text{Conf}, \text{EXEThrd}(\bar{c})\), assume that \(\bar{t}\) is as defined in Algorithm 6.4 and that \(\bar{t}'\) is defined as \(\bar{t}\) given by Table 5.11. It is easy to see that \(\text{Thrd}_{\text{hold}}\) as given by Algorithm 6.4 is a superset of \(\text{Thrd}_{\text{hold}}\) as given by Table 5.11 since in the latter case, a lock might have been assigned to some thread, which will exclude that thread from \(\text{Thrd}_{\text{hold}}\). Thus it must be that \(\min(\gamma_\bar{c}(\bar{t}')) \leq \min(\gamma_\bar{t}(\bar{t}))\) and \(\max(\gamma_\bar{c}(\bar{t}')) \leq \max(\gamma_\bar{t}(\bar{t}))\) (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
6.1 Abstract Execution

Algorithm 6.4 Threads to execute in an abstract configuration

1: function EXETHRD(\(\bar{c}@([T,pct,T,\bar{t}_T]_{T\in\text{Thrd}_\gamma},\bar{x},\bar{\bar{y}})]\))
2: \(\text{Thrd}_{\text{hold}} \leftarrow \{T \in \text{Thrd}_\gamma : \text{STM}(T,pc_T) = [\text{halt}]^{pc_T} \lor \exists lck \in \text{Lck} : (\text{STM}(T,pc_T) = [\text{lock } lck]^{pc_T} \land \text{O\textsc{wn}}(\lp lck \rp) \neq T)\}\)
3: \(\langle [\bar{c}]_{T\in\text{Thrd}\setminus\text{Thrd}_{\text{hold}}},\bar{x},\bar{\bar{y}}\rangle \leftarrow (\langle \text{ABSTIME}(\bar{c},T)\rangle_{T\in\text{Thrd}\setminus\text{Thrd}_{\text{hold}}})\)
4: \(t_{\text{min}} \leftarrow \min\{\min(\gamma(T,\bar{t}_T+\bar{r}_T)) : T \in \text{Thrd}_\gamma \setminus \text{Thrd}_{\text{hold}}\}\)
5: \(t_{\text{max}} \leftarrow \min\{\max(\gamma(T,\bar{t}_T+\bar{r}_T)) : T \in \text{Thrd}_\gamma \setminus \text{Thrd}_{\text{hold}}\}\)
6: \(\bar{t} \leftarrow \alpha(\{t_{\text{min}},t_{\text{max}}\})\)
7: return \(\{T \in \text{Thrd}_\gamma \setminus \text{Thrd}_{\text{hold}} | \bar{t} \cap \gamma(T,\bar{t}_T+\bar{r}_T) \neq \bot_T\} \cup\)
\(\{T \in \text{Thrd}_{\text{hold}} | \exists lck \in \text{Lck} : (\text{STM}(T,pc_T) = [\text{lock } lck]^{pc_T} \land \text{O\textsc{wn}}(\lp lck \rp) = \bot_{\text{thrd}})\}\)
8: end function

O\textsc{wn}(\lp lck \rp) = \bot_{\text{thrd}}). Thus it must be that \(\text{Thrd}^x_{\text{exe}} \subseteq \text{EXETHRD}(\bar{c})\), where \(\text{Thrd}^x_{\text{exe}}\) is as defined in Table 5.11, since EXETHRD(\(\bar{c}\)) is derived based on \(\bar{t}\) but also includes all threads issuing \text{lock } lck where \(lck \in \text{Lck}\) and O\textsc{wn}(\lp lck \rp) = \bot_{\text{thrd}}.

Given a set of threads, \(\text{Thrd}_\gamma \subseteq \text{Thrd}\), GLOBALVAR(\(\text{Thrd}_\gamma\)), as defined in Algorithm 6.5, is the set of variables that could transfer data between some of the threads in \(\text{Thrd}_\gamma\) (Lemma 6.3).

Algorithm 6.5 Global variables in an abstract configuration

1: function GLOBALVAR(\(\text{Thrd}_\gamma\))
2: \(\langle [\text{Var}^\text{load}_T]_{T\in\text{Thrd}_\gamma} \rangle \leftarrow\)
\(\langle [\{x \in \text{Var} \mid \exists r \in \text{Reg}_T : \exists l \in \text{Lbl}_T : \text{STM}(T,l) = [\text{load } r \text{ from } x]_l\}]_{T\in\text{Thrd}_\gamma}\rangle\)
3: \(\langle [\text{Var}^\text{store}_T]_{T\in\text{Thrd}_\gamma} \rangle \leftarrow\)
\(\langle [\{x \in \text{Var} \mid \exists r \in \text{Reg}_T : \exists l \in \text{Lbl}_T : \text{STM}(T,l) = [\text{store } r \text{ to } x]_l\}]_{T\in\text{Thrd}_\gamma}\rangle\)
4: return \(\{x \in \text{Var} \mid \exists T,T' \in \text{Thrd}_\gamma : (T \neq T' \land x \in \text{Var}^\text{load}_T \land x \in \text{Var}^\text{store}_{T'})\}\)
5: end function

Lemma 6.3 (Soundness of GLOBALVAR):
GLOBALVAR(\(\text{Thrd}_\gamma\)) 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 \(\text{Thrd}_\gamma\).

PROOF. Assume that \(x \in \text{Var}\). First note that
Given a configuration \( \bar{c} @ \langle [T, pc_T, \bar{T}, \bar{\bar{T}}]_{T \in \text{Thr}_\emptyset}, \bar{x}, \bar{l} \rangle \) ∈ \( \text{Conf} \), \( \{ T \in \text{Thr}_\emptyset | \exists r \in \text{Reg}_T : \exists x \in \text{GlobalVAR}(\text{Thr}_\emptyset) : \text{STM}(T, pc_T) = \langle \text{load r from x} \rangle \} \subseteq \text{EXELOADTHRD}(\bar{c}) \), where \( \text{Thr}_\emptyset \text{exe} \) is defined as in Table 5.11.

\[ \]
Algorithm 6.6

Given a configuration \( \tilde{c} \@ \langle [T, pc_T, \tilde{x}_T, \tilde{y}_T \mid T \in Thrd_\tilde{c}, \tilde{x}, \tilde{y} \rangle \rangle \in Conf \) and that \( Thrd_{exe}^\tilde{c} \) is defined as in Table 5.11. The proof follows directly from the fact that \( Thrd_{exe}^\tilde{c} \subseteq EXEThrd(\tilde{c}) \) (Lemma 6.2).

Given an abstract configuration, \( \tilde{c} \in Conf \), \( ISFINAL(\tilde{c}) \), as defined in Algorithm 6.7, means that \( \tilde{c} \) is in the final state; i.e. all threads issue the \texttt{halt}-statement.

Algorithm 6.7 Final abstract configuration

1: function \( ISFINAL([T, pc_T, \tilde{x}_T, \tilde{y}_T \mid T \in Thrd_\tilde{c}, \tilde{x}, \tilde{y}]) \)
2: return \( \forall T' \in Thrd_\tilde{c} \; STM(T, pc_T) = [\texttt{halt}]^{pc_T} \)
3: end function

Given an abstract configuration, \( \tilde{c} \in Conf \), \( ISDEADLOCK(\tilde{c}) \), as defined in Algorithm 6.8, means that \( \tilde{c} \) cannot reach a final state according to the abstract semantic rules (Lemma 6.5). Note that \( ISDEADLOCK \) is not guaranteed to identify all such cases, though.

Algorithm 6.8 Deadlocked abstract configuration

1: function \( ISDEADLOCK(\tilde{c} @ [T, pc_T, \tilde{x}_T, \tilde{y}_T \mid T \in Thrd_\tilde{c}, \tilde{x}, \tilde{y}]) \)
2: Require: \( \neg ISFINAL(\tilde{c}) \)
3: \( Thrd_{lock} \leftarrow \{ T \in Thrd_\tilde{c} \mid \exists lck \in Lck : (STM(T, pc_T) = \langle lock lck \rangle^{pc_T} \land \right.
\left. OWN(lck) \notin \{ \bot_{thrd}, T \} \wedge STM(lck) = locked \} \}
4: \( E \leftarrow \{ (T, T') \in Thrd_{lock} \times Thrd_{lock} \mid \exists lck \in Lck : \\
STM(T, pc_T) = \langle lock lck \rangle^{pc_T} \land \\
STM(lck) = locked \wedge \\
STM(OWN(lck), pc_{OWN(lck)}) = [\texttt{halt}]^{pc_{OWN(lck)}} \} \}
5: end function

Lemma 6.5 (Soundness of \textbf{ISDEADLOCK}):

Given a configuration \( \tilde{c} @ [T, pc_T, \tilde{x}_T, \tilde{y}_T \mid T \in Thrd_\tilde{c}, \tilde{x}, \tilde{y} \rangle \in Conf \), such that \( \exists T \in Thrd_\tilde{c} : STM(T, pc_T) \neq [\texttt{halt}]^{pc_T} \), \( ISDEADLOCK(\tilde{c}) \Rightarrow \forall c \in \gamma_{conf}(\tilde{c}) : \\
\neg \exists c' @ [T, pc_T, \tilde{x}_T, \tilde{y}_T \mid T \in Thrd_\tilde{c}, \tilde{x}, \tilde{y} \rangle \in Conf : \\
(c \longrightarrow_{pc} \ldots \longrightarrow_{pc} c' \wedge \forall T \in Thrd_\tilde{c} : \\
STM(T, pc_T') = [\texttt{halt}]^{pc_T'} \rangle \), where \( c \) and \( c' \) are valid concrete configurations (cf. Definition 4.4); i.e. if \( ISDEADLOCK(\tilde{c}) \), then \( \tilde{c} \) does not represent any concrete configuration that can possibly reach a final state. \[ \square \]
PROOF. Assume that \( \tilde{c} \in \text{Conf} \), such that \( \exists T \in \text{Thrd}_\tilde{c} : \text{STM}(T, pc_T) \neq [\text{halt}]^{pc_T} \) (note that this assumption fulfills \( \neg\text{isFINAL}(\tilde{c}) \)) and \( \text{isDEADLOCK}(\tilde{c}) \). Note that it must be that \( \text{Thrd}_\tilde{c} = \text{Thrd} \) (otherwise, \( \neg\text{isDEADLOCK}(\tilde{c}) \)).

Since \( \text{Thrd}_{\text{lock}} = \{ T \in \text{Thrd}_\tilde{c} \mid \exists lck \in \text{Lck} : (\text{STM}(T, pc_T) = [\text{lock} lck]^{pc_T} \land \text{OWN}(lck) \notin \{\bot_{\text{thrd}, T}\} \land \text{STT}(lck) = \text{locked} \} \) and \( E = \{(T, T') \mid T, T' \in \text{Thrd}_{\text{lock}} \land \exists lck \in \text{Lck} : (\text{STM}(T, pc_T) = [\text{lock} lck]^{pc_T} \land \text{OWN}(lck) = T') \} \), 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 \( \text{CYCLE}(\text{Thrd}_{\text{lock}}, E) \), then for all \( c \in \gamma_{\text{conf}}(\tilde{c}) \) (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, \( \forall c \in \gamma_{\text{conf}}(\tilde{c}) : \neg\exists c' @ [\langle T, pc'_T, \bar{r}_T, f'_T \rangle]_{\text{Thrd}_\tilde{c}} : \tilde{x}', \tilde{l}' \rangle \in \text{Conf} : (c \xrightarrow{prg} \ldots \xrightarrow{prg} c' \land \forall T \in \text{Thrd}_\tilde{c} : \text{STM}(T, pc'_T) = [\text{halt}]^{pc'_T} \) (Note that if for some thread, \( T \in \text{Thrd}_\tilde{c} \), and lock, \( lck \in \text{Lck} \), \( \text{STM}(T, pc_T) = [\text{lock} lck]^{pc_T} \), \( \text{OWN}(lck) \notin \{\bot_{\text{thrd}, T}\} \) and \( \text{STT}(lck) = \text{unlocked} \), then \( T \notin \text{Thrd}_{\text{lock}} \) since \( \text{OWN}(lck) \) has not yet acquired \( lck \).)

If \( \exists T \in \text{Thrd}_\tilde{c} : \exists lck \in \text{Lck} : (\text{STM}(T, pc_T) = [\text{lock} lck]^{pc_T} \land \text{STT}(lck) = \text{locked} \land \text{OWN}(lck) \neq \bot_{\text{thrd}, T} \land \text{STM}(\text{OWN}(lck), pc_{\text{OWN}(lck)}) = [\text{halt}]^{pc_{\text{OWN}(lck)}} \), it is easy to see that \( \text{OWN}(lck) \) will never issue \( \text{unlock} lck \) (cf. Tables 5.10 and 5.11) and thus \( \forall c \in \gamma_{\text{conf}}(\tilde{c}) : \neg\exists c' @ [\langle T, pc'_T, r'_T, f'_T \rangle]_{\text{Thrd}_\tilde{c}} \), \( \tilde{x}', \tilde{l}' \rangle \in \text{Conf} : (c \xrightarrow{prg} \ldots \xrightarrow{prg} c' \land \forall T \in \text{Thrd}_\tilde{c} : \text{STM}(T, pc'_T) = [\text{halt}]^{pc'_T} \).

This concludes the proof.

Given an abstract configuration, \( \tilde{c} \in \text{Conf} \), and a timeout, \( \tilde{t}_{\text{to}} \in \text{Ti\&ime} \), \( \text{isTIMEOUT}(\tilde{c}, \tilde{t}_{\text{to}}) \), as defined in Algorithm 6.9, means that \( \tilde{c} \) cannot reach a final state 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.

**Lemma 6.6 (Soundness of \text{isTIMEOUT}):**

Given a configuration, \( \tilde{c} @ [\langle T, pc_T, \bar{r}_T, \bar{f}_T \rangle]_{\text{Thrd}_\tilde{c}} \), \( \tilde{x}', \tilde{l}' \rangle \in \text{Conf} \), and timeout, \( \tilde{t}_{\text{to}} \in \text{Ti\&ime} \), such that \( \exists T \in \text{Thrd}_\tilde{c} : \text{STM}(T, pc_T) \neq [\text{halt}]^{pc_T} \) and \( \neg\text{isDEADLOCK}(\tilde{c}) \), \( \text{isTIMEOUT}(\tilde{c}, \tilde{t}_{\text{to}}) \Rightarrow \forall c \in \gamma_{\text{conf}}(\tilde{c}) : \neg\exists c' @ [\langle T, pc'_T, r'_T, f'_T \rangle]_{\text{Thrd}_\tilde{c}} \), \( \tilde{x}', \tilde{l}' \rangle \in \text{Conf} : (c \xrightarrow{prg} \ldots \xrightarrow{prg} c' \land \forall T \in \text{Thrd}_\tilde{c} : (\text{STM}(T, pc'_T) = [\text{halt}]^{pc'_T} \land f'_T \leq \max(\gamma_{t}(\tilde{t}_{\text{to}}))) \), where \( c \) and \( c' \) are valid concrete configurations (cf. Definition 4.4); i.e. \( \text{isTIMEOUT}(\tilde{c}, \tilde{t}_{\text{to}}) \), then \( \tilde{c} \)
Algorithm 6.9 Timed-out abstract configuration

1: function ISTIMEOUT(\@([T,pc_T,\tilde{\xi}_T,\tilde{\rho}_T]|_{\Theta} Thrd_c, \tilde{\xi}, \tilde{\rho}_T, \tilde{t}_o))

Require: \text{\neg ISFINAL}(\tilde{\xi}) \land \text{\neg ISDEADLOCK}(\tilde{\xi})

2: return \( \forall T \in \text{Thrd}_c : (\text{STM}(T,pc_T) \neq [\text{halt}]^{pct} \rightarrow \\
(\tilde{t}_o \preceq t_{pc} + ABSTIME(\tilde{\xi}, T)) \lor \\
(\text{Thrd}_c \subseteq \text{Thrd} \land \exists lck \in \text{Lck} : (\text{STM}(T,pc_T) = [\text{lock } lck]^{pct} \land \\
\text{OWN}(\tilde{lck}) \notin \{\bot_{\text{thrhd}}, T\}))) \}

3: end function

does not represent any concrete configuration that can possibly reach a final state before the given timeout (i.e. before \(\gamma_t(\tilde{t}_o)\)). □

PROOF. Assume that \(\tilde{\xi}@([T,pc_T,\tilde{\xi}_T,\tilde{\rho}_T]|_{\Theta} Thrd_c, \tilde{\xi}, \tilde{\rho}_T, \tilde{t}_o) \in \text{Conf} \) and \(\tilde{t}_o \in \text{Time} \) are such that \(\exists T \in \text{Thrd}_c : \text{STM}(T,pc_T) \neq [\text{halt}]^{pct}, \text{\neg ISDEADLOCK}(\tilde{\xi}) \) and \(\text{ISTIMEOUT}(\tilde{\xi}, \tilde{t}_o)\).

Since ISTIMEOUT(\tilde{\xi}, \tilde{t}_o), it must be that \(\forall T \in \text{Thrd}_c : (\text{STM}(T,pc_T) \neq [\text{halt}]^{pct} \rightarrow (\tilde{t}_o \preceq t_{pc} + ABSTIME(\tilde{\xi}, T)) \lor (\text{Thrd}_c \subseteq \text{Thrd} \land \exists lck \in \text{Lck} : (\text{STM}(T,pc_T) = [\text{lock } lck]^{pct} \land \text{OWN}(\tilde{lck}) \notin \{\bot_{\text{thrhd}}, T\}))) \). For all threads, \(T \in \text{Thrd}_c\), such that \(\text{STM}(T,pc_T) \neq [\text{halt}]^{pct} \land \tilde{t}_o \preceq t_{pc} + ABSTIME(\tilde{\xi}, T),\) it is easy to see that \(\neg \exists c'@([T,pc_c',\tilde{\xi}'_T,\tilde{\rho}'_T]|_{\Theta} Thrd_c, \tilde{\xi}', \tilde{\rho}'_T) \in \text{Conf} : (c_{pc}' \rightarrow \ldots \rightarrow c' \land \text{STM}(T,pc_c') = [\text{halt}]^{pct} \land \tilde{t}_o' \leq \max(\gamma_t(\tilde{t}_o)))\) (cf. Assumptions 4.1 and 5.51). Thus, for all other threads, \(T \in \text{Thrd}_c\), such that \(\exists lck \in \text{Lck} : (\text{STM}(T,pc_T) = [\text{lock } lck]^{pct} \land \text{OWN}(\tilde{lck}) \notin \{\bot_{\text{thrhd}}, T\})\), it must be that \(\neg \exists c'@([T,pc_c',\tilde{\xi}'_T,\tilde{\rho}'_T]|_{\Theta} Thrd_c, \tilde{\xi}', \tilde{\rho}'_T) \in \text{Conf} : (c_{pc}' \rightarrow \ldots \rightarrow c' \land \text{STM}(T,pc_c') = [\text{halt}]^{pct} \land \tilde{t}_o' \leq \max(\gamma_t(\tilde{t}_o)))\) since the respective locks cannot possibly be released at any time, \(t\), such that \(t \leq \max(\gamma_t(\tilde{t}_o))\) (cf. Assumptions 4.1 and 5.51).

This concludes the proof. □

Given an abstract configuration, \(\tilde{\xi} \in \text{Conf}\), and a timeout, \(\tilde{t}_o \in \text{Time}\), \(\text{\neg ISVALID}(\tilde{\xi}, \tilde{t}_o)\), where ISVALID(\(\tilde{\xi}, \tilde{t}_o)\) is as defined in Algorithm 6.10, means that \(\tilde{\xi}\) cannot reach a configuration that could represent at least one valid (cf. Definition 4.4) concrete configuration (Lemma 6.7). Note that ISVALID might not identify all possible such cases, though.

**Lemma 6.7 (Soundness of ISVALID):**

*Given a configuration \(\tilde{\xi}@([T,pc_T,\tilde{\xi}_T,\tilde{\rho}_T]|_{\Theta} Thrd_c, \tilde{\xi}, \tilde{\rho}_T) \in \text{Conf} \) and a time, \(\tilde{t}_o \in \text{Time}\), such that \(\exists T \in \text{Thrd}_c : \text{STM}(T,pc_T) \neq [\text{halt}]^{pct}, \text{\neg ISDEADLOCK}(\tilde{\xi})\) and...*
Algorithm 6.10 Valid abstract configuration

1: function ISVALID($\vec{c} @ \langle [T, pc_T, \tilde{x}, \tilde{t}_i], T \in \text{Thrd}_\vec{x}, \tilde{x}, \tilde{t}_i \rangle$)

Require: $\neg \text{ISFINAL}(\vec{c}) \land \neg \text{ISDEADLOCK}(\vec{c}) \land \neg \text{ISTIMEOUT}(\vec{c}, \tilde{t}_i)$

2: $\text{Thrd}_{\text{lock}} \leftarrow \{ T \in \text{Thrd}_\vec{x} \mid \exists lck \in \text{Lck} : (\text{STM}(T, pc_T) = [\text{lock} lck]_{\text{pc}_T} \land \text{OWN}(\tilde{lck}) \not\in \{ \bot_{\text{thrd}}, T \}) \}$

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]_{\text{pc}_T} \land \text{OWN}(\tilde{lck}) = T') \}$

4: return $(\text{Thrd}_\vec{x} = \text{Thrd} \Rightarrow \neg \text{CYCLE}(\text{Thrd}_{\text{lock}}, E)) \land \forall lck \in \text{Lck} : (\text{STM}(T, pc_T) = [\text{lock} lck]_{\text{pc}_T} \land \text{OWN}(\tilde{lck}) = \bot_{\text{thrd}}) \Rightarrow$

(\text{STM}(T, pc_T) = [\text{lock} lck]_{\text{pc}_T} \land \text{OWN}(\tilde{lck}) = \bot_{\text{thrd}}) \}$

5: end function

and $\neg \text{ISTIMEOUT}(\vec{c}, \tilde{t}_i), \neg \text{ISVALID}(\vec{c}, \tilde{t}_i) \Rightarrow \neg \exists \vec{c}' \in \text{Conf} : (c \xrightarrow{\text{prg}} \ldots \xrightarrow{\text{prg}} \vec{c}' \land \exists c @ \langle [T, pc_T, \tilde{x}, \tilde{t}_i], T \in \text{Thrd}_\vec{x}, \tilde{x}, \tilde{t}_i \rangle$) $\in \gamma_{\text{conf}}(\vec{c}') : (\exists c' \in \text{Conf} : c' \xrightarrow{\text{prg}} \ldots \xrightarrow{\text{prg}} c \land \forall lck \in \text{Lck} : ((\text{STM}(T, pc_T) = [\text{lock} lck]_{\text{pc}_T} \land \text{OWN}(\tilde{lck}) = \bot_{\text{thrd}})) \Rightarrow \text{OWN}(\tilde{lck}) = \bot_{\text{thrd}}\})$; i.e. $\vec{c}$ can never lead to a configuration that could represent at least one valid concrete configuration (cf. Definition 4.4).

□

Proof. Assume that $\vec{c} @ \langle [T, pc_T, \tilde{x}, \tilde{t}_i], T \in \text{Thrd}_\vec{x}, \tilde{x}, \tilde{t}_i \rangle \in \text{Conf}$ and $\tilde{t}_i$ to $\text{Time}$ are such that $\exists T \in \text{Thrd}_\vec{x} : \text{STM}(T, pc_T) \neq [\text{halt}]_{\text{pc}_T}, \neg \text{ISDEADLOCK}(\vec{c}), \neg \text{ISTIMEOUT}(\vec{c}, \tilde{t}_i)$, and $\neg \text{ISVALID}(\vec{c}, \tilde{t}_i)$. Then it must be that

1. $\neg (\text{Thrd}_\vec{x} = \text{Thrd} \Rightarrow \neg \text{CYCLE}(\text{Thrd}_{\text{lock}}, E))$ (i.e. $\text{Thrd}_\vec{x} = \text{Thrd} \land \text{CYCLE}(\text{Thrd}_{\text{lock}}, E)$), where $\text{Thrd}_{\text{lock}} = \{ T \in \text{Thrd}_\vec{x} \mid \exists lck \in \text{Lck} : (\text{STM}(T, pc_T) = [\text{lock} lck]_{\text{pc}_T} \land \text{OWN}(\tilde{lck}) \not\in \{ \bot_{\text{thrd}}, T \}) \}$ and $E = \{ (T, T') \mid T, T' \in \text{Thrd}_{\text{lock}} \land \exists lck \in \text{Lck} : (\text{STM}(T, pc_T) = [\text{lock} lck]_{\text{pc}_T} \land \text{OWN}(\tilde{lck}) = T') \}$, or

2. $\forall lck \in \text{Lck} : \forall T \in \text{Thrd}_\vec{x} : ((\text{STM}(T, pc_T) = [\text{lock} lck]_{\text{pc}_T} \land \text{OWN}(\tilde{lck}) = \bot_{\text{thrd}}) \Rightarrow (\text{STM}(T, pc_T) = [\text{lock} lck]_{\text{pc}_T} \land \text{OWN}(\tilde{lck}) = \bot_{\text{thrd}}) \})$.

If $\text{Thrd}_\vec{x} = \text{Thrd} \land \text{CYCLE}(\text{Thrd}_{\text{lock}}, E)$, 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 $\neg \text{ISDEADLOCK}(\vec{c})$, it must be that this cycle involves at least one lock, $lck \in \text{Lck}$, such that for some thread, $T \in \text{Thrd}$, $\text{OWN}(\tilde{lck}) \not\in \{ \bot_{\text{thrd}}, T \}$ and $\text{STM}(T, pc_T) = [\text{lock} lck]_{\text{pc}_T} \land \text{OWN}(\tilde{lck}) = \bot_{\text{thrd}}$ (cf. Algorithm 6.8). But then it is easy to see that $\neg \exists \vec{c}' \in \text{Conf} : (c \xrightarrow{\text{prg}} \ldots \xrightarrow{\text{prg}} \vec{c}' \land \exists c @ \langle [T, pc_T, \tilde{x}, \tilde{t}_i], T \in \text{Thrd}_\vec{x}, \tilde{x}, \tilde{t}_i \rangle$)
\[ \gamma_{\text{conf}}(\bar{c}') : (\exists c' \in \text{Conf} : c' \xrightarrow{\text{prg}} \ldots \xrightarrow{\text{prg}} c \land \forall \text{lck} \in \text{Lck} : (\text{STT}(lck) = \text{unlocked} \Rightarrow \text{OWN}(lck) = \bot_{\text{thrd}})) \] (cf. Tables 5.10 and 5.11 and Lemma 4.5).

Note that if \( \neg \forall \text{lck} \in \text{Lck} : \forall T \in \text{Thrd}_{\bar{c}} : ((\text{OWN}(lck) = T \land \text{STT}(lck) = \text{unlocked}) \Rightarrow (\text{STM}(T,pc_T) \neq [\text{halt}]^{\text{pcT}} \land \text{DL}(lck) \not\preceq_t (i^a_{T} \xrightarrow{t} \text{ABSTIME}(\bar{c},T)))) \) then it must be that \( \exists \text{lck} \in \text{Lck} : \exists T \in \text{Thrd}_{\bar{c}} : (\text{OWN}(lck) = T \land \text{STT}(lck) = \text{unlocked} \land (\text{STM}(T,pc_T) = [\text{halt}]^{\text{pcT}} \lor \text{DL}(lck) \preceq_t (i^a_{T} \xrightarrow{t} \text{ABSTIME}(\bar{c},T)))) \).

If \( \exists \text{lck} \in \text{Lck} : \exists T \in \text{Thrd}_{\bar{c}} : (\text{OWN}(lck) = T \land \text{STT}(lck) = \text{unlocked} \land \text{STM}(T,pc_T) = [\text{halt}]^{\text{pcT}}) \), then it is easy to see that \( \neg \exists \exists c' \in \text{Conf} : (c' \xrightarrow{\text{prg}} \ldots \xrightarrow{\text{prg}} c' \land \exists c \in \text{Conf} : \langle T,pc_T,t_{T}^{a},t_{T}^{b},T_{T}^{c},t_{T}^{d},\text{Thrd}_{\bar{c}},x,\|$] \rangle \in \gamma_{\text{conf}}(\bar{c}') : (\exists c' \in \text{Conf} : \langle c' \xrightarrow{\text{prg}} \ldots \xrightarrow{\text{prg}} c \land \forall \text{lck} \in \text{Lck} : (\text{STT}(lck) = \text{unlocked} \Rightarrow \text{OWN}(lck) = \bot_{\text{thrd}})) \rangle \rangle \). since, for the given lock, the owner will be T but the state will remain unlocked for all configurations following \( \bar{c} \) (cf. Tables 5.10 and 5.11).

If \( \exists \text{lck} \in \text{Lck} : \exists T \in \text{Thrd}_{\bar{c}} : (\text{OWN}(lck) = T \land \text{STT}(lck) = \text{unlocked} \land \text{DL}(lck) \preceq_t (i_{T}^{a} \xrightarrow{t} \text{ABSTIME}(\bar{c},T))) \), then it is easy to see that \( \neg \exists \exists c' \in \text{Conf} : (c' \xrightarrow{\text{prg}} \ldots \xrightarrow{\text{prg}} c' \land \exists c \in \text{Conf} : \langle T,pc_T,\text{Thrd}_{\bar{c}},x,\|$] \rangle \in \gamma_{\text{conf}}(\bar{c}') : (\exists c' \in \text{Conf} : \langle c' \xrightarrow{\text{prg}} \ldots \xrightarrow{\text{prg}} c \land \forall \text{lck} \in \text{Lck} : (\text{STT}(lck) = \text{unlocked} \Rightarrow \text{OWN}(lck) = \bot_{\text{thrd}})) \rangle \rangle \).

This concludes the proof.

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.

**Algorithm 6.11 Get variable in \text{load}-statement**

1: \textbf{function} \text{GETVARLOAD}[\text{load} \text{r} \text{from} \text{x}][l]
2: \hspace{1cm} return x
3: \textbf{end function}

Theorem 6.8 states that, as discussed above, \text{ABSEX}(\bar{c},i_0)\) gives a sound approximation of the timing behavior of all configurations in the collecting
Theorem 6.8 (Soundness of ABS EXE):

The structure of the algorithm is such that, for any considered configuration on any level of recursion have

The overall structure of the algorithm is of the work-list type; i.e. given

Algorithm 6.12 Get register in load-statement

1: function GETREGLOAD([load r from x]')
2: return r
3: end function

semantics of the concrete initial configurations corresponding to the abstract configurations found in \(\hat{C}\) up until \(t_{fo} = \max(\gamma(\tilde{t}_{fo}))\) whenever it terminates. Assuming that the algorithm terminates, the theorem also states some important properties of the output sets; \(\hat{C}', \hat{C}d\) and \(\hat{C}'\). If \(\hat{C}'\) is not empty, then it could be that the program deadlocks for some of the initial states. And, if \(\hat{C}'\) 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 \(t_{fo}\)).

Theorem 6.8 (Soundness of ABS EXE):

If the sets of valid configurations \(C \in \mathcal{P}(Conf)\) (cf. Definition 4.4) and \(\hat{C} \in \mathcal{P}(\hat{Conf})\), are such that \(\forall c@\langle[T,pc_{T},\gamma_{T},\tilde{r}_{T}]_{T\in Thrd_{1}},\tilde{x},\tilde{l}\rangle \in C:\langle\forall\langle[T,pc_{T}^{\hat{C}},\tilde{x},\tilde{l}]_{T\in Thrd_{2}},\hat{C} : (\text{Thrd}_{1} = \text{Thrd}_{2} = \text{Thrd}) \land \exists \tilde{c} \in \hat{C} : c \in Y_{conf}(\tilde{c}) \rangle \land |\text{Thrd}| < \infty \land \forall \tilde{c}@\langle[T,pc_{T}^{\hat{C}},\tilde{x},\tilde{l}]_{T\in Thrd^{\hat{C}}},\hat{C} : \forall \text{Lck} \in Lck : \min(\gamma(\text{DL}(\tilde{Lck})) = -\infty)\rangle\), then given \(\tilde{t}_{fo} \in \text{Time}^{\hat{C}}\), \((\hat{C}', \hat{C}d, \hat{C}')@\text{ABS EXE}(\hat{C},\tilde{t}_{fo})\) is such that

\[
\forall c \in C : \forall c'@\langle[T,pc_{T}^{c'},\gamma_{T},\tilde{r}_{T}]_{T\in Thrd^{c'}},\tilde{x'},\tilde{l}\rangle \in \text{Conf} : \ (c_{prg} \ldots c_{prg} c'_{prg} \land \forall T \in \text{Thrd} : \text{STM}(T,pc_{T}^{c'}) = [\text{halt}]^{pc_{T}^{c'}}) \Rightarrow \\
(\hat{C}' \neq \emptyset \lor \exists \tilde{c} @ \langle[T,pc_{T}^{\hat{C}},\tilde{x},\tilde{l}]_{T\in Thrd^{\hat{C}}},\hat{C} : \forall T \in \text{Thrd} : \ (pc_{T}^{\hat{C}} = pc_{T}^{c'} \land r_{T}^{\hat{C}} = \gamma(\tilde{r}_{T}^{\hat{C}}))) \rangle \\
\forall c \in C : \forall c'@\langle[T,pc_{T}^{c'},\gamma_{T},\tilde{r}_{T}]_{T\in Thrd^{c'}},\tilde{x'},\tilde{l}\rangle \in \text{Conf} : \\
((c_{prg} \ldots c_{prg} c'_{prg} c'_{prg} \land (\text{CYCLE}(\text{Thrd}_{lock}^{c'},E'^{c'}) \lor \\
\exists T \in \text{Thrd} : \exists \text{Lck} \in Lck : \ (\text{STM}(T,pc_{T}^{c'}) = [\text{lock lck}]^{pc_{T}^{c'}} \land \\
\text{OWN}([lck]'') \not\subset \{\perp_{thrd},T\} \land \\
\text{STM}(\text{OWN}([lck]''),pc_{T}^{c}'_{own}) = [\text{halt}]^{pc_{T}^{c'}_{own}}(\tilde{lck}''))) \Rightarrow \\
(\hat{C}' \neq \emptyset \lor \hat{C}d \neq \emptyset))
\]
where $\text{Thrd}_{\text{lock}}' = \{ T \in \text{Thrd} \mid \exists \text{lck} \in \text{Lck} : (\text{STM}(T, pc_T') = [\text{lck lck}]^{pc_T'} \land \\
onumber \text{OWN}(\text{lck}')) \notin \{ \bot_{\text{thrd}}, T \}) \}$ and $E_c' = \{(T, T') \mid T, T' \in \text{Thrd}_{\text{lock}}' \land \exists \text{lck} \in \text{Lck} : (\text{STM}(T, pc_T') = [\text{lck lck}]^{pc_T'} \land \text{OWN}(\text{lck}')) = T' \}$, whenever it terminates.

Furthermore, if $\bar{C}^d \cup \bar{C}^f = \emptyset$, then:

$$
\forall c \in C : \forall c' @ ([T, pc_T', r_T', t_T', T_{\text{Thrd}'}], x', y') \in \text{Conf} : \\
(c \xrightarrow{\text{prg}} \ldots \xrightarrow{\text{prg}} c' \land \forall T \in \text{Thrd} : \text{STM}(T, pc_T') = [\text{halt}]^{pc_T'} \Rightarrow \\
\exists ([T', pc_{T'}', r_{T'}, t_{T'}, T_{\text{Thrd}'}], x', y') \in \bar{C}^f : (pc_{T'} = pc'_T \land t_{T'}' \in \gamma'(\bar{I}_{T'}'))) \\
\square
$$

PROOF. Given $\bar{i}_{to}$ in $\text{Time}$, assume that the sets of valid configurations $C \in \mathcal{P}(\text{Conf})$ and $\bar{C} \in \mathcal{P}(\text{Conf})$ are such that $\forall c @ ([T, pc_T', r_T', t_T', T_{Thrd'}], x', y') \in C : (\forall ([T, pc_{T'}, r_{T'}, t_{T'}, T_{\text{Thrd}'}], x', y') \in \bar{C} : (\text{Thrd}_1 = \text{Thrd}_2 = \text{Thrd}) \land \exists \bar{c} \in \bar{C} : c \in \gamma(\bar{c}) \land |\text{Thrd}| < \infty \land \forall c @ ([T, pc_T', r_T', t_T', T_{\text{Thrd}'}], x', y') \in \bar{C} : \forall \text{lock} \in \text{Lck} : \\
\min(\gamma(\bar{I}_{\text{lck}})) = -\infty))$, and that $(\bar{C}^d, \bar{C}^f, \bar{C}^f) = \text{ABSEXEC}(\bar{C}, \bar{i}_{to})$.

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 $\bar{c}$, $\text{Thrd}_{\text{lock}} = \text{Thrd}$) 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 work-list type; i.e. given an item (abstract configuration in this case) that is extracted from a work-list, new items are generated, based on some rules, and are either added to the work-list (and will thus eventually be extracted themselves) or saved as output items if some condition is fulfilled. When the work-list is empty, the algorithm terminates.

- Since $(\bar{C}^f, \bar{C}^d, \bar{C}^f) = \text{ABSEXEC}(\bar{C}, \bar{i}_{to})$, i.e. $\text{ABSEXEC}(\bar{C}, \bar{i}_{to})$ results in a tuple of three sets, it must be that the algorithm terminates for the particular input (i.e. $\bar{C}$ and $\bar{i}_{to}$).

- The structure of the algorithm is such that, for any $\bar{c} \in \text{Conf}$ and $\bar{i}_{to} \in \text{Time}$, $\text{ISDEADLOCK}(\bar{c})$ is only issued when $\neg \text{ISFINAL}(\bar{c})$, $\text{ISTIMEOUT}(\bar{c}, \bar{i}_{to})$ is only issued when $\neg \text{ISDEADLOCK}(\bar{c})$, and $\text{ISVALID}(\bar{c}, \bar{i}_{to})$ is only issued when $\neg \text{ISTIMEOUT}(\bar{c}, \bar{i}_{to})$. 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 \textsc{Abs Time} (Assumption 5.51).

• When the considered level of recursion, \( i \), is greater than 0, it is easy to see that \( \text{Thrd}_i \subseteq \text{Thrd} \), 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 \(|\text{EXE Thrd}(\tilde{c})| > 1\) for recursion to occur and \(|\text{Thrd}| \geq |\text{EXE Thrd}(\tilde{c})|\) for any \( \tilde{c} \in \text{Conf} \) (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 \( \max(\gamma_i(\tilde{t}_{to}^i)) \leq \max(\gamma_i(\tilde{t}_{to}^{i-1})) \) since \( \tilde{t}_{to}^i = \tilde{t}_{to}^i \cap_T (\tilde{t}_{to}^{i-1} \cup_T \alpha_T(\{-\infty\})) \), where \( T \in \text{Thrd}_\tilde{c} \) is the thread that will be removed from the configurations on recursion level \( i \). This means that the timeout cannot be shifted into the future when recursion occurs.

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 \( \tilde{t}_{to}^{i+1} = \tilde{t}_{to}^i \cap_T \text{ABSTime}(\tilde{c}^{i-1}, T_{i-1}) \).

• Assume that a thread, \( T \in \text{Thrd} \), issues a \texttt{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 \texttt{LOAD}-statement at recursion level \( i - 1 \), then a new recursion level, \( i \), will be created to determine a safe write history before the \texttt{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 \( \tilde{t}_{to}^i = \tilde{t}_{to}^i \cap_T (\tilde{t}_{to}^{i-1} \cup_T \alpha_T(\{-\infty\})) \) 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).
Figure 6.2: Illustration of how the timeout, $\tilde{t}_{to}$, for a new level of recursion in ABSEXE is determined.
- 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 1oad-statement) gives that all possible cases, for which order 1oad-statements in different threads can be issued, are considered.

Assume that, given some configuration, \( \bar{\gamma} \circ \langle [T, pc_T^i, \bar{x}_T, \bar{i}_T]_{T \in \text{Thrd}}, \bar{x}; \bar{l} \rangle \), and timeout, \( \bar{i}_{to} \), on recursion level \( i \), where \( 0 \leq i < n \), some thread, \( T_i \in \text{Thrd} \), issues a possibly unsafe 1oad-statement (which means that a deeper recursion level, \( i + 1 \), will exist) that cannot be affected by any event occurring after \( \bar{i}_{to} \). 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 \( \bar{\gamma} \) 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 1oad-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 \( \bar{\gamma} \) safely approximate the possible concrete thread states and variable values given by the configuration at the end of the corresponding concrete transition sequences and all 1oad-statements on recursion levels \( i + 1 \) to \( n \) are safely approximated, it must be that all \( \bar{\gamma}' \in C_\text{Conf} \), such that \( \bar{\gamma} \xrightarrow{\text{prg}} \ldots \xrightarrow{\text{prg}} \bar{\gamma}' \), safely approximate the possible concrete thread states and variable values given the considered program point and the corresponding concrete transition sequences since \( \xrightarrow{\text{prg}} \) is used to approximate the execution of all statements except 1oad-statements that are possibly unsafe (cf. Lemmas 5.56, 5.57 and 5.58). Thus, it must be that \( (\bar{C}_{i+1}, \bar{C}_{i+1}^d, \bar{C}_{i+1}^t) \) @ ABSEXE\( \{\langle [T, pc_T^i, \bar{x}_T, \bar{i}_T]_{T \in \text{Thrd}\backslash\{T_i\}}, \bar{x}; \bar{l} \rangle, (\bar{i}_T^a, \top_t, \text{ABSTIME}(\bar{\gamma}, T_i)) \cap_1 (\bar{i}_{to} \cup \alpha_i(\{-\infty\})) \rangle \) is such that \( \bigcup_{\text{val}} \{\text{READ}(\bar{x}', x, T_i, \bar{i}_T^a, \top_t, \text{ABSTIME}(\bar{\gamma}, T_i)) \mid \langle \bar{T}, \bar{x}', \bar{l}' \rangle \in \bar{C}_{i+1} \cup \bar{C}_{i+1}^d \cup \bar{C}_{i+1}^t \cup \{\bar{\gamma}\} \} \) safely approximates all possible concrete values that could be read by the 1oad-statement in \( T_i \) for the corresponding concrete transition sequence, since (the ABSEXE instance mentioned above, corresponding to recursion level \( i + 1 \) is considered)

1. \( \{\bar{\gamma}'' \in C_\text{Conf} \mid \bar{\gamma}' \xrightarrow{\text{prg}} \bar{\gamma}'' \} \) safely collects all transition possibilities for any given configuration, \( \bar{\gamma}' \in C_\text{Conf} \), or rather, thread, for which no possibly unsafe 1oad-statements are approximated by the transition (cf. Lemmas 5.56, 5.57 and 5.58),

2. TRIM is not used to remove old writes from the write history since \( i + 1 > 0 \) (cf. Table 5.11),
3. (note that \( \text{Thrd}_{i+1} \subseteq \text{Thrd} \)) \( \forall \tilde{e}' \in \{[T, pc_T', \tilde{x}_T', \tilde{t}_T'] | T \in \text{Thrd}' \} \in \text{Conf} : \)

\( (\text{Thrd}' \subseteq \text{Thrd} \Rightarrow \neg \text{ISDEADLOCK}(\tilde{e}')) \); i.e. even if a deadlock exists in \( \tilde{e}' \), 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 \( \tilde{e}' \in \text{Coñf} \) and \( \tilde{t}_{t_0} \in \text{Time} \), \( \text{ISTIMEOUT}(\tilde{e}', \tilde{t}_{t_0}) \Rightarrow \forall c \in \gamma_{\text{conf}}(\tilde{e}') : \neg \exists c^c @ ([T, pc_T, \tilde{x}_T, t_T] | T \in \text{Thrd}', \tilde{x}', \tilde{t}' \}) \in \text{Conf} : (c \rightarrow \ldots \rightarrow c') \land \forall T \in \text{Thrd} : (\text{STM}(T, pc_T) = |\text{halt}|^{pc_T} \land t_T^c \leq \max(\gamma_T(\tilde{t}_{t_0}))) \)

where \( c \) and \( c' \) are valid concrete configurations (cf. Definition 4.4); i.e. if \( \text{ISTIMEOUT}(\tilde{e}', \tilde{t}_{t_0}) \), then \( \tilde{e}' \) 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 \( \tilde{e}' \) can affect the system state so that the effects are visible at or before \( \tilde{t}_{t_0} \) (cf. Algorithm 6.9 and Assumption 5.5).

5. (note that \( \text{Thrd}_{i+1} \subseteq \text{Thrd} \)) \( \forall \tilde{e}' \in \{[T, pc_T', \tilde{x}_T', \tilde{t}_T'] | T \in \text{Thrd}' \} \in \text{Coñf} : (\text{Thrd}' \subset \text{Thrd} \land \neg \text{ISVALID}(\tilde{e}')) \Rightarrow \neg \exists \text{lock} \in \text{Lck} : \forall T \in \text{Thrd}' : (\text{OWN}(\tilde{t}' \text{lock}) = T \land \text{STM}(t_T') = \text{unlocked}) \Rightarrow (\text{STM}(T, pc_T) \neq |\text{halt}|^{pc_T} \land \tilde{t}_T \neq \text{ABSTIME}(\tilde{e}, T))) \)

which follows directly from Algorithm 6.10 and means that there is no possibility that \( \tilde{e}' \) 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 5.10 and 5.11 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 \( \tilde{t}^d_{T_i} = \tilde{t}_{T_i} + \text{ABSTIME}(\tilde{e}, T_i) \), \( pc_{T_i} = pc_{T_i} + 1 \) and \( \tilde{x}_{T_i} = \tilde{x}_{T_i} [r \mapsto \text{READ}(\tilde{x}, x, T_i, \tilde{t}_{T_i}, \tilde{t}_{T_i}) + \text{ABSTIME}(\tilde{e}, T_i)] \) (assuming that the possibly unsafe load-statement issued by \( T_i \) is \( \text{load} \ r \ from \ x \) \( p_{T_i} \) for some \( r \in \text{Reg}_{T_i} \) and \( x \in \text{Var} \)), it must thus be that the load-statement in thread \( T_i \) on recursion level \( i \) is safely approximated and
that the new configuration, which is added to the work-list 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 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 load-statement has yet been approximated. Since no potentially unsafe load-statement has yet been approximated and \(\forall c \in C : \exists \tilde{c} \in \tilde{C} : c \in \gamma_{conf}(\tilde{c})\), 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 \(\tilde{c} \xrightarrow{\text{prog}}\) has been safely used for all transitions and \(\{c' \in \text{Conf} \mid \tilde{c} \xrightarrow{\text{prog}} c'\}\) 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, \(\tilde{c} @ \langle [T, pc_T, \tilde{\tau}_T, \tilde{\tau}_T]' T \in \text{Thrd}_t \rangle \in \text{Conf}\), at level \(n\), \(\exists \text{EXEHLD} \text{Thrd}(\tilde{c})\) \(\neq 1\) or \(\text{EXEHLD} \text{Thrd}(\tilde{c}) = 0\). But since for any given configuration \(\tilde{c} @ \langle [T, pc_T, \tilde{\tau}_T, \tilde{\tau}_T]' T \in \text{Thrd}_t \rangle \in \text{Conf}\), \(\text{Thrd}^\text{exe} \subseteq \text{EXEHLD} \text{Thrd}(\tilde{c})\) (Lemma 6.2) and \(\{T \in \text{Thrd}^\text{exe} \mid \exists r \in \text{Reg}_T : \exists x \in \text{GLOBALVAR}(\text{Thrd}) : \text{STM}(T, pc_T) = [\text{load } r \text{ from } x]^{pc_T} \} \subseteq \text{EXEHLD} \text{Thrd}(\tilde{c})\) (Lemma 6.4), where \(\text{Thrd}^\text{exe}\) is as defined in Table 5.11, it must thus be that \(\text{Thrd}^\text{exe} \neq 1\) \(\lor\) \(\{T \in \text{Thrd}^\text{exe} \mid \exists r \in \text{Reg}_T : \exists x \in \text{Var}_g : \text{STM}(T, pc_T) = [\text{load } r \text{ from } x]^{pc_T} \} = \emptyset\) for all \(\tilde{c} \in \text{Conf}\) at recursion level \(n\). Thus, it must be that \(\{c' \in \text{Conf} \mid \tilde{c} \xrightarrow{\text{prog}} c'\}\) 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 \text{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 \(\forall c \in C : \exists \tilde{c} \in \tilde{C} : c \in \gamma_{conf}(\tilde{c})\). 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 \(\tilde{C}'\), \(\tilde{C}^d\) and \(\tilde{C}^e\) are created. Given \(c \in C\) and \(c' @ \langle [T, pc_T', \tilde{\tau}'_T, \tilde{\tau}'_T]' T \in \text{Thrd}_t \rangle \in \text{Conf}\), the following
6.1 Abstract Execution

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 $c \rightarrow_{\text{prg}} \ldots \rightarrow_{\text{prg}} c' \land \forall T \in \text{Thrd} : \text{stm}(T, pc_T^c) = [\text{halt}]^{pc_T^c}$.

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, $\tilde{c} \in \text{Cnf}$, on that trace are such that $\neg \text{isDeadlock}(\tilde{c})$ (cf. Algorithm 6.8 and Lemma 6.5) and $\text{isValid}(\tilde{c}, \tilde{t}_0)$ (cf. Algorithm 6.10 and Lemma 6.7). It must also be that, eventually, a configuration, $\tilde{c} \in \{ [T, pc_T^c, \tilde{t}_T, \tilde{t}_T', \tilde{\theta}, \tilde{\eta}]_{T \in \text{Thrd}} : \tilde{x}, \tilde{\eta} \} \in \text{Cnf}$, for which either $\forall T \in \text{Thrd} : \text{stm}(T, pc_T^c) = [\text{halt}]^{pc_T^c}$ or $\forall T \in \text{Thrd} : (\text{stm}(T, pc_T^c) \neq [\text{halt}]^{pc_T^c} \Rightarrow \tilde{t}_0 \prec \tilde{t}_T' \land \text{timeout}(\tilde{c}, T))$ is derived along the corresponding (over-approximating) abstract trace of transitions.

If $\forall T \in \text{Thrd} : \text{stm}(T, pc_T^c) = [\text{halt}]^{pc_T^c}$, then it is easy to see that $\text{isFinal}(\tilde{c})$ (cf. Algorithm 6.7), which means that $\tilde{c} \in \tilde{C}_T$. Thus, it must be that $\exists c' @ (\{ [T, pc_T^c, \tilde{t}_T', \tilde{t}_T', \tilde{\theta}, \tilde{\eta}]_{T \in \text{Thrd}} : \tilde{x}', \tilde{\eta}' \} \in \tilde{C}_T : \forall T \in \text{Thrd} : (pc_T^c = pc_T^c' \land \tilde{t}_T' \in \gamma(\tilde{\eta}'))$.

If $\exists T \in \text{Thrd} : \text{stm}(T, pc_T^c) \neq [\text{halt}]^{pc_T^c} \land \forall T \in \text{Thrd} : (\text{stm}(T, pc_T^c) \neq [\text{halt}]^{pc_T^c} \Rightarrow \tilde{t}_0 \prec \tilde{t}_T' \land \text{timeout}(\tilde{c}, T))$, 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 $\text{isTimeout}(\tilde{c}, \tilde{t}_0)$ (cf. Algorithms 6.7 and 6.9), which means that $\tilde{c} \in \tilde{C}_T$. Thus, it must be that $\tilde{C}_T \neq \emptyset$.

2. Assume that $c \rightarrow_{\text{prg}} \ldots \rightarrow_{\text{prg}} c' \land (\text{CYCLE}((\text{Thrd}_{\text{lock}}^c'E', T')) \lor \exists T \in \text{Thrd} : \exists \text{lock} \in \text{Lck} : (\text{stm}(T, pc_T^c) = [\text{lock lck}]^{pc_T^c} \land \text{own}(\tilde{t}_c', \text{lock}) \notin \{ \text{\_thrd}, T \} \land \text{stm}(\text{own}(\tilde{t}_c, \text{lock}), pc_{\text{own}(\tilde{t}_c, \text{lock})}) = [\text{halt}]^{pc_{\text{own}(\tilde{t}_c, \text{lock})}}))$, where $\text{Thrd}_{\text{lock}}^c'E' = \{ T \in \text{Thrd} | \exists \text{lock} \in \text{Lck} : (\text{stm}(T, pc_T^c) = [\text{lock lck}]^{pc_T^c} \land \text{own}(\tilde{t}_c, \text{lock}) \notin \{ \text{\_thrd}, T \} ) \}$ and $E' = \{ (T, T') | T, T' \in \text{Thrd}_{\text{lock}}^c \land \exists \text{lock} \in \text{Lck} : (\text{stm}(T, pc_T^c) = [\text{lock lck}]^{pc_T^c} \land \text{own}(\tilde{t}_c, \text{lock}) = T') \}$ (remember that $\text{own}(\tilde{t}_c, \text{lock}) \neq \text{\_thrd} \Rightarrow \text{own}(\tilde{t}_c, \text{lock}) = \text{locked}$ since $c$ is valid and $\rightarrow_{\text{prg}}$ 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 \( \text{ISVALID}(\tilde{c}, \tilde{t}_0) \) (cf. Algorithm 6.10 and Lemma 6.7). It must also be that, eventually, a configuration, \( \tilde{c} \oplus \langle [T, p\tilde{c}_T, \tilde{r}_T, \tilde{t}_T] \mid \tilde{c} \in \text{Conf} \rangle \), will be derived (along the corresponding, over-approximating abstract trace of transitions) for which either (\( \text{CYCLE}(\text{Thr\ddash lock}^\tilde{c}, E^\tilde{c}) \lor \exists T \in \text{Thr} : \exists lck \in \text{Lck} : (\text{STM}(T, p\tilde{c}_T) = [\text{lock} \ lck]\ p\tilde{c}_T \land \text{STM}(\tilde{lck}) = \text{locked} \land \text{OWN}(\tilde{lck}) \neq \bot_{\text{thrd}} \land \text{STM}(\text{OWN}(\tilde{lck}), p\tilde{c}_\text{OWN}(\tilde{lck})) = \text{halt}^{p\tilde{c}_\text{OWN}(\tilde{lck})}) \)), where
\[
\text{Thr\ddash lock}^\tilde{c} = \{ T \in \text{Thr} \mid \exists lck \in \text{Lck} : (\text{STM}(T, p\tilde{c}_T) = [\text{lock} \ lck]\ p\tilde{c}_T \land \text{OWN}(\tilde{lck}) \not\subseteq \{ \bot_{\text{thrd}}, T \} \land \text{STM}(\tilde{lck}) = \text{locked}) \} \land E^\tilde{c} = \{ (T, T') \mid T, T' \in \text{Thr}\text{lock} \land \exists lck \in \text{Lck} : (\text{STM}(T, p\tilde{c}_T) = [\text{lock} \ lck]\ p\tilde{c}_T \land \text{OWN}(\tilde{lck}) = T') \}, \forall T \in \text{Thr} : (\text{STM}(T, p\tilde{c}_T) \neq [\text{lock} \ lck]\ p\tilde{c}_T \land \text{OWN}(\tilde{lck}) = T'), \exists lck \in \text{Lck} : (\text{STM}(T, p\tilde{c}_T) = [\text{lock} \ lck]\ p\tilde{c}_T \land \text{STM}(\text{OWN}(\tilde{lck}), p\tilde{c}_\text{OWN}(\tilde{lck})) = \text{halt}^{p\tilde{c}_\text{OWN}(\tilde{lck})}) \}
\]

If (\( \text{CYCLE}(\text{Thr\ddash lock}^\tilde{c}, E^\tilde{c}) \lor \exists T \in \text{Thr} : \exists lck \in \text{Lck} : (\text{STM}(T, p\tilde{c}_T) = [\text{lock} \ lck]\ p\tilde{c}_T \land \text{STM}(\text{OWN}(\tilde{lck}), p\tilde{c}_\text{OWN}(\tilde{lck})) = \text{halt}^{p\tilde{c}_\text{OWN}(\tilde{lck})}) \)), then it is easy to see that \( \neg \text{ISFINAL}(\tilde{c}) \) and \( \text{ISDEADLOCK}(\tilde{c}) \) (cf. Algorithm 6.8 and Lemma 6.5), which means that \( \tilde{C} \neq \emptyset \).

If (\( \forall T \in \text{Thr} : (\text{STM}(T, p\tilde{c}_T) \neq [\text{lock} \ lck]\ p\tilde{c}_T \land \text{STM}(\text{OWN}(\tilde{lck}), p\tilde{c}_\text{OWN}(\tilde{lck})) = \text{halt}^{p\tilde{c}_\text{OWN}(\tilde{lck})}) \)), then it is easy to see that \( \neg \text{ISFINAL}(\tilde{c}), \neg \text{ISDEADLOCK}(\tilde{c}) \) and \( \text{ISTIMEOUT}(\tilde{c}, \tilde{t}_0) \) (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 \( \tilde{C} \cup \tilde{C}' = \emptyset \). Since (\( \forall [T, p\tilde{c}_T, \tilde{r}_T, \tilde{t}_T] \in \tilde{C} : \forall T \in \text{Thr} : \text{STM}(T, p\tilde{c}_T) = [\text{lock} \ lck]\ p\tilde{c}_T \) (cf. Algorithm 6.7) and \( \neg \text{ISVALID}(\tilde{c}, \tilde{t}_0) \) 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 $c' @ \langle [T, p\tau T', T'] \in \text{Thrd}; \bar{x}', \bar{y}' \rangle \in \text{Conf}$ are such that $c \xrightarrow{pc} \ldots \xrightarrow{pc} c' \wedge \forall T \in \text{Thrd} : \text{STM}(T, p\tau T') = [\text{halt}]p\tau T'$. Since $\bar{C} \cup \bar{C}' = \emptyset$ and $\forall c \in C : \exists \bar{c} \in \bar{C} : c \in \gamma_{\text{conf}}(\bar{c})$, it is easy to see that $\exists \langle [T, p\tau T', T'] \in \text{Thrd}; \bar{x}, \bar{y} \rangle \in \bar{C}^f : (p\tau T, p\tau T' \wedge \tau T' \in \gamma(\bar{y}))$ since all possible concrete transition sequences are safely approximated. This concludes the proof.

**Note.** \textsc{absex}e has not been proven to terminate for all inputs. However, when it does terminate, it safely approximates the transition sequences for the corresponding concrete input set.

One case for which \textsc{absex}e 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 (Definition 6.9) of a program, given some initial system states, are safely approximated by \textsc{analysis}, which is defined in Algorithm 6.13, whenever it terminates (Theorem 6.10). The algorithm simply derives a safe approximation of the timing behavior of the concrete collecting semantics of a given set of initial configurations using \textsc{absex}e (defined in Algorithm 6.1 on page 157). Then the smallest BCET and the largest WCET among the resulting configurations are found. Note that \textsc{choose} was defined in Algorithm 6.2 on page 159 and that it gives a deterministically chosen element from the considered set.

**Definition 6.9 (BCET and WCET):**
The Best-Case Execution Time, BCET, and the Worst-Case Execution Time, WCET, of a given configuration, $\langle [T, p\tau T', T'] \in \text{Thrd}; \bar{x}, \bar{y} \rangle \in \text{Conf}$ are defined as:

\[
\begin{align*}
\text{BCET} &= \max \{ \min(\gamma_i(\bar{y})) \mid T \in \text{Thrd}_c \} \\
\text{WCET} &= \max \{ \max(\gamma_i(\bar{y})) \mid T \in \text{Thrd}_c \}
\end{align*}
\]
Algorithm 6.13 BCET/WCET analysis

1: \textbf{function} ANALYSIS($\hat{C}, \tilde{t}_{t_{0}}$)
2: \hspace{1em} ($\hat{C}^f, \hat{C}^d, \hat{C}^l$) $\leftarrow$ ABSEXE($\hat{C}, \tilde{t}_{t_{0}}$)
3: \hspace{1em} if $\hat{C}^l \cup \hat{C}^d \neq \emptyset$ then
4: \hspace{2em} $BCET \leftarrow \min(\{\min(\gamma_i(\tilde{t}_T^d)) \mid T' \in \text{Thrd} \land \langle [ T, pc_T, \tilde{t}_T^d ]_{T' \in \text{Thrd}} : \tilde{x}, \tilde{I} \rangle \in \hat{C} \})$
5: \hspace{2em} return $BCET$, $\infty$
6: \hspace{1em} end if
7: \hspace{1em} $BCET \leftarrow \infty$
8: \hspace{1em} $WCET \leftarrow -\infty$
9: \hspace{1em} \textbf{while} $\hat{C}^l \neq \emptyset$ do
10: \hspace{2em} $\tilde{c} @ \langle [ T, pc_T, \tilde{t}_T^d ]_{T' \in \text{Thrd}} : \tilde{x}, \tilde{I} \rangle \leftarrow \text{CHOOSE}(\hat{C}^l)$
11: \hspace{2em} $\hat{C}^l \leftarrow \hat{C}^l \setminus \{\tilde{c}\}$
12: \hspace{2em} $BCET_{\tilde{c}} \leftarrow \max(\{\min(\gamma_i(\tilde{t}_T^d)) \mid T \in \text{Thrd} \})$
13: \hspace{2em} $WCET_{\tilde{c}} \leftarrow \max(\{\max(\gamma_i(\tilde{t}_T^d)) \mid T \in \text{Thrd} \})$
14: \hspace{2em} if $BCET > BCET_{\tilde{c}}$ then
15: \hspace{3em} $BCET \leftarrow BCET_{\tilde{c}}$
16: \hspace{1em} end if
17: \hspace{2em} if $WCET < WCET_{\tilde{c}}$ then
18: \hspace{3em} $WCET \leftarrow WCET_{\tilde{c}}$
19: \hspace{1em} end if
20: \hspace{1em} \textbf{end while}
21: \hspace{1em} return $BCET$, $WCET$
22: \textbf{end function}
Algorithm 6.13

178 Chapter 6. Safe Execution Time Analysis by Abstract Execution

22: return

17: end while

18: if

14: if

2: \( (T, pc_T, \tau_T, t_{\tau_T}) \in \text{Thrd}_1 \) and \( \exists \tilde{e} \in \tilde{C} : c \in \gamma_{\text{conf}}(\tilde{e}) \) then \( |\text{Thrd}| < \infty \) and \( \forall \tilde{e} @ \langle [T, pc_T, \tau_T, t_{\tau_T}] \rangle \in \tilde{C} : \forall \text{lck} @ \text{Lck} : \min(\gamma_t(\text{Dl}(l_{\text{lck}}) = -\infty)) \), then given \( \tilde{i}_0 \in \text{Ti\~n}e \), \((BCET, WCET) \) @ ANALYSIS\( (\tilde{C}, \tilde{i}_0) \) is such that

\[ \forall c \in C : \forall c' @ \langle [T, pc'_T, \tau'_T, t_{\tau'_T}] \rangle \in \text{Conf} : \]

\[ (((c \rightarrow_p \ldots \rightarrow_p c') \land C) \land \forall T @ \text{Thrd} : \text{STM}(T, pc'_T) = [\text{halt}]^{T, pc_T} ) \Rightarrow \]

\[ \forall T @ \text{Thrd} : BCET \leq t_{\tau'_T} \leq WCET) \land \]

\[ (c \rightarrow_p \ldots \rightarrow_p c' \Rightarrow t_{\tau'_T} \leq WCET) \]

given that the algorithm terminates.

\[ \square \]

PROOF. Given \( \tilde{i}_0 \in \text{Ti\~n}e \), assume that the sets of valid concrete configurations \( C \in \mathcal{P}(\text{Conf}) \) (cf. Definition 4.4) and abstract configurations \( \tilde{C} \in \mathcal{P}(\text{Conf}) \) and abstract configurations \( \tilde{C} \in \mathcal{P}(\text{Conf}) \) are such that \( \forall c @ \langle [T, pc_T, \tau_T, t_{\tau_T}] \rangle \in C : \forall \langle [T, pc_T, \tau_T, t_{\tau_T}] \rangle \in \tilde{C} : \text{Thrd}_1 = \text{Thrd}_2 = \text{Thrd} \) and \( \exists \tilde{e} \in \tilde{C} : c \in \gamma_{\text{conf}}(\tilde{e}) \) then \( |\text{Thrd}| < \infty \) and \( \forall \tilde{e} @ \langle [T, pc_T, \tau_T, t_{\tau_T}] \rangle \in \tilde{C} : \forall \text{lck} @ \text{Lck} : \min(\gamma_t(\text{Dl}(l_{\text{lck}}) = -\infty)), \]

and that \((BCET, WCET) \) = ANALYSIS\( (\tilde{C}, \tilde{i}_0) \).

Since \((BCET, WCET) \) = ANALYSIS\( (\tilde{C}, \tilde{i}_0) \), it must be that \((\tilde{C}', \tilde{C}'', \tilde{C}') @ \text{ABSEXE}(\tilde{C}, \tilde{i}_0) \) terminates at some point and that

\[ \forall c \in C : \forall c' @ \langle [T, pc'_T, \tau'_T, t_{\tau'_T}] \rangle \in \text{Conf} : \]

\[ (((c \rightarrow_p \ldots \rightarrow_p c') \land C) \land \forall T @ \text{Thrd} : \text{STM}(T, pc'_T) = [\text{halt}]^{T, pc_T} ) \Rightarrow \]

\[ (\tilde{C}' \neq \emptyset) \lor \]

\[ \exists \tilde{e} @ \langle [T, pc_T, \tau_T, t_{\tau_T}] \rangle \in \text{Thrd}_1 : \tilde{e} \in \tilde{C}' : \forall T @ \text{Thrd} : \]

\[ (pc_T = pc_{\tau'_T} \land t_{\tau'_T} \in \gamma_t(\tilde{t}_{\tau'_T})) \) \)

\[ \forall c \in C : \forall c' @ \langle [T, pc'_T, \tau'_T, t_{\tau'_T}] \rangle \in \text{Conf} : \]

\[ (((c \rightarrow_p \ldots \rightarrow_p c') \land \text{CYCLE}(\text{Thrd}_{\text{lock}}, E') ) \land \]

\[ \exists T @ \text{Thrd} : \exists \text{lck} @ \text{Lck} : \]

\[ (\text{STM}(T, pc_T) = [\text{lock lck}]^{pc_{\tau'_T}} \land \]

\[ \text{OWN}(\text{lck}') \notin \{\uparrow_{\text{thrd}}, T\} \land \]

\[ \text{STM}(\text{OWN}(\text{lck}'), pc_{\text{OWN}(\text{lck}')}) = \]

\[ [\text{halt}]^{pc_{\text{OWN}(\text{lck}')}}) \)

\[ (\tilde{C}' \neq \emptyset \lor \tilde{C}'' \neq \emptyset) \)
It is thus apparent that if $\tilde{C}^d \cup \tilde{C}^f \neq \emptyset$, there might exist an infinite transition sequence in the concrete case. However, it is easy to see that $\min(\{\min(\gamma_i(\tilde{x}_i)) \mid T' \in \text{Thrd} \land (\langle[T, pc_T, \tilde{x}_T, \tilde{r}_T]_{T \in \text{Thrd}}, \tilde{x}, \tilde{l} \rangle \in \tilde{C})\})$ 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 $\tilde{C}^d \cup \tilde{C}^f = \emptyset$, then all concrete transition sequences are of finite length and $\forall c \in C : \forall c' @ (\langle[T, pc_T', \tilde{x}_T', \tilde{r}_T']_{T \in \text{Thrd}}, \tilde{x}', \tilde{l}' \rangle \in \text{Conf} : ((c \xrightarrow{\text{prg}} \ldots \text{prg} c') \wedge \forall T \in \text{Thrd} : \text{STM}(T, pc_T') = [\text{halt}]^{pc_T}) \Rightarrow \exists(\langle[T, pc_T, \tilde{x}_T, \tilde{r}_T]_{T \in \text{Thrd}}, \tilde{x}, \tilde{l} \rangle \in \tilde{C}^f : (pc_T = pc_T' \wedge \tilde{r}_T' \in \gamma_i(\tilde{x}_i))$ (Theorem 6.8). Thus, since the structure of the algorithm trivially gives that the smallest possible estimation of the BCET, BCET, and the largest possible estimation of the WCET, WCET, among the derived final abstract configurations in $\tilde{C}^f$ are found (cf. Definition 6.9), it must be that $\forall c \in C : \forall c' @ (\langle[T, pc_T', \tilde{x}_T', \tilde{r}_T']_{T \in \text{Thrd}}, \tilde{x}', \tilde{l}' \rangle \in \text{Conf} : ((c \xrightarrow{\text{prg}} \ldots \text{prg} c') \Rightarrow \forall T \in \text{Thrd} : \text{BCET} \leq \tilde{r}_T' \leq \text{WCET})$. But, then it must also be that $\forall c \in C : \forall c' @ (\langle[T, pc_T, \tilde{x}_T, \tilde{r}_T]_{T \in \text{Thrd}}, \tilde{x}', \tilde{l}' \rangle \in \text{Conf} : (c \xrightarrow{\text{prg}} \ldots \text{prg} c' \Rightarrow \forall T \in \text{Thrd} : \tilde{r}_T' \leq \text{WCET})$ since time only moves forward (Lemma 4.2 and Assumption 5.51), which concludes the proof.
Chapter 7

Examples

To clarify and explain the analysis defined in Chapters 5 and 6, this chapter instantiates it for some example PPL programs.

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, Thrd = \{T_1, T_2, T_3\}, defined in Table 7.1, it is easy to see that Reg_{T_1} = \{r\}, Reg_{T_2} = \{r\}, Reg_{T_3} = \{r\}, Var = \{x, y, z\} and Lck = \emptyset. Note that r represents local memory within each thread; i.e. the register-name r can refer to three different memory locations – what location it refers to

<table>
<thead>
<tr>
<th>Table 7.1: Communication – Program.</th>
</tr>
</thead>
<tbody>
<tr>
<td>T_1 @ (1, {load r from x}_1; [store r to y]_2; [halt]_3)</td>
</tr>
<tr>
<td>T_2 @ (2, {load r from y}_1; [store r to z]_2; [halt]_3)</td>
</tr>
<tr>
<td>T_3 @ (3, {if r &lt;= 3 goto 4}_1; [store r to x]_2; [skip]_3; [halt]_4)</td>
</tr>
</tbody>
</table>
depends on which thread is considered.

Assume that $\text{ABSTIME}(\bar{c}, T)$, i.e. the abstracted timing model, where $\bar{c} @ \langle[T, pc_T, \bar{x}_T, T]_{T \in \text{Thrd}_T} \bar{c}, \bar{c}, \bar{c} \rangle \in \text{Conf}$ and $T \in \text{Thrd}_T$, is such that for any $\bar{c}$, 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, $\bar{c}_0 @ \langle[T, pc_T, \bar{x}_T, T]_{T \in \text{Thrd}_T} \bar{c}, \bar{c}, \bar{c} \rangle$, 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 $\text{ABSEXE}(\{c^0_0\}, [-\infty, \infty])$ during the analysis described by $\text{ANALYSIS}(\{c^0_0\}, [-\infty, \infty])$. 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 $\text{ABSEXE}$ are created, note that when $\text{Thrd}_T = \{T_1, T_2, T_3\}$, then $\text{Var}_g = \{x, y\}$; when $\text{Thrd}_T = \{T_1, T_3\}$, then $\text{Var}_g = \{x\}$; and when $\text{Thrd}_T = \{T_2, T_3\}$, then $\text{Var}_g = \emptyset$.

It is apparent that $\text{ABSEXE}(\{c^0_0\}, [-\infty, \infty]) = (\{c^0_0, c^0_23\}, \emptyset, \emptyset)$; i.e. $c^0_11$ and $c^0_23$ are final-state configurations and there are no deadlocked or timed-out configurations. Note that $c^1_{12}, c^1_{22}, c^1_{11}, c^2_{22}, c^2_{22}$ and $c^2_{22}$ only exist within the recursively called $\text{ABSEXE}$-instances. According to Algorithm 6.13, it is thus easy to see that the estimated timing bounds are:

\[
\begin{align*}
\text{BCET} & = \min(\{\max(\{\min(\gamma_T(h^T)) \mid T \in \text{Thrd}\}) \mid \langle[T, pc_T, \bar{x}_T, T]_{T \in \text{Thrd}_T} \bar{c}, \bar{c}, \bar{c} \rangle \in \{c^0_11, c^0_23\}\}) = 4 \\
\text{WCET} & = \max(\{\min(\gamma_T(h^T)) \mid T \in \text{Thrd}\}) \mid \langle[T, pc_T, \bar{x}_T, T]_{T \in \text{Thrd}_T} \bar{c}, \bar{c}, \bar{c} \rangle \in \{c^0_11, c^0_23\}\}) = 11
\end{align*}
\]
Table 7.3: Communication – Configurations (first half).

<table>
<thead>
<tr>
<th>$\bar{c}$</th>
<th>$pc_{T_1}$</th>
<th>$pc_{T_2}$</th>
<th>$pc_{T_3}$</th>
<th>$\bar{x}_{T_1}$</th>
<th>$\bar{x}_{T_2}$</th>
<th>$\bar{x}_{T_3}$</th>
<th>$\bar{x}_{T_1}$</th>
<th>$\bar{x}_{T_2}$</th>
<th>$\bar{x}_{T_3}$</th>
<th>$(\bar{x} \ x) \ T_3$</th>
<th>$(\bar{x} \ y) \ T_1$</th>
<th>$(\bar{x} \ z) \ T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\bar{c}_0$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>$[0,0]$</td>
<td>$[0,0]$</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>${(\mathbb{I}_{val},\mathbb{I}_r)}$</td>
</tr>
<tr>
<td>$\bar{c}_1$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>$[2,4]$</td>
<td>$[0,0]$</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>${(\mathbb{I}_{val},\mathbb{I}_r)}$</td>
</tr>
<tr>
<td>$\bar{c}_2$</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>$[5,5]$</td>
<td>$[2,3]$</td>
<td>$[2,6]$</td>
<td>$[1,4]$</td>
<td>$[1,4]$</td>
<td>$[1,4]$</td>
<td>${(1,1),[0,0]}$</td>
<td>${(5,5),[0,0]}$</td>
<td>${(\mathbb{I}_{val},\mathbb{I}_r)}$</td>
</tr>
<tr>
<td>$\bar{c}_3$</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>$[5,5]$</td>
<td>$[4,9]$</td>
<td>$[4,8]$</td>
<td>$[4,8]$</td>
<td>$[4,8]$</td>
<td>$[4,8]$</td>
<td>${(1,1),[0,0]}$</td>
<td>${(5,5),[0,0]}$</td>
<td>${(\mathbb{I}_{val},\mathbb{I}_r)}$</td>
</tr>
<tr>
<td>$\bar{c}_4$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>$[2,4]$</td>
<td>$[0,0]$</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>${(\mathbb{I}_{val},\mathbb{I}_r)}$</td>
</tr>
<tr>
<td>$\bar{c}_5$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>$[2,4]$</td>
<td>$[0,0]$</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>${(\mathbb{I}_{val},\mathbb{I}_r)}$</td>
</tr>
<tr>
<td>$\bar{c}_6$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>$[2,4]$</td>
<td>$[0,0]$</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>${(\mathbb{I}_{val},\mathbb{I}_r)}$</td>
</tr>
<tr>
<td>$\bar{c}_7$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>$[2,4]$</td>
<td>$[0,0]$</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>${(\mathbb{I}_{val},\mathbb{I}_r)}$</td>
</tr>
</tbody>
</table>

(continued on next page)
Table 7.4: Communication – Configurations (second half).

<table>
<thead>
<tr>
<th>$\tilde{c}$</th>
<th>$pc_{T_1}$</th>
<th>$pc_{T_2}$</th>
<th>$pc_{T_3}$</th>
<th>$\tilde{r}_{T_1}$</th>
<th>$\tilde{r}_{T_2}$</th>
<th>$\tilde{r}_{T_3}$</th>
<th>$\tilde{r}_{T_3}^a$</th>
<th>$\tilde{r}_{T_3}^a$</th>
<th>$\tilde{r}_{T_3}^a$</th>
<th>$(\tilde{\varphi} \ x) \ T_3$</th>
<th>$(\tilde{\varphi} \ y) \ T_1$</th>
<th>$(\tilde{\varphi} \ z) \ T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\tilde{c}_1^{12}$</td>
<td>2</td>
<td>1</td>
<td>[1, 4]</td>
<td>[2, 4]</td>
<td>[1, 5]</td>
<td>[0, 0]</td>
<td>${(1, 1), (0, 0)}$</td>
<td>${(5, 5), (0, 0)}$</td>
<td>${(\overline{I}_{val}, \overline{I}_r)}$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\tilde{c}_1^{11}$</td>
<td>3</td>
<td>4</td>
<td>[1, 4]</td>
<td>[2, 3]</td>
<td>[2, 8]</td>
<td>[1, 4]</td>
<td>${(1, 1), (0, 0)}$</td>
<td>${(5, 5), (0, 0)}$</td>
<td>$(1, 4), (2, 8)$</td>
<td>${(\overline{I}_{val}, \overline{I}_r)}$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\tilde{c}_2^{11}$</td>
<td>3</td>
<td>2</td>
<td>[1, 4]</td>
<td>[4, 4]</td>
<td>[2, 8]</td>
<td>[1, 4]</td>
<td>${(1, 1), (0, 0)}$</td>
<td>${(5, 5), (0, 0)}$</td>
<td>$(1, 4), (2, 8)$</td>
<td>${(\overline{I}_{val}, \overline{I}_r)}$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\tilde{c}_2^{12}$</td>
<td>3</td>
<td>3</td>
<td>[1, 4]</td>
<td>[4, 4]</td>
<td>[2, 8]</td>
<td>[4, 8]</td>
<td>${(1, 1), (0, 0)}$</td>
<td>${(5, 5), (0, 0)}$</td>
<td>$(1, 4), (2, 8)$</td>
<td>${(\overline{I}_{val}, \overline{I}_r)}$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\tilde{c}_1^0$</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>[1, 4]</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>$(\overline{I}_{val}, \overline{I}_r)$</td>
<td></td>
</tr>
<tr>
<td>$\tilde{c}_1^{11}$</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>[1, 4]</td>
<td>[2, 3]</td>
<td>[2, 8]</td>
<td>[4, 9]</td>
<td>[1, 4]</td>
<td>${(1, 1), (0, 0)}$</td>
<td>${(5, 5), (0, 0)}$</td>
<td>$(1, 4), (2, 8)$</td>
<td>${(\overline{I}_{val}, \overline{I}_r), (1, 5), (4, 9)}$</td>
</tr>
<tr>
<td>$\tilde{c}_2^{11}$</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>${(5, 5), (0, 0)}$</td>
<td>$(1, 4), (2, 8)$</td>
</tr>
<tr>
<td>$\tilde{c}_2^{12}$</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>${(1, 1), (0, 0)}$</td>
<td>${(5, 5), (0, 0)}$</td>
<td>$(1, 4), (2, 8)$</td>
</tr>
<tr>
<td>$\tilde{c}_2^{13}$</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>${(1, 1), (0, 0)}$</td>
<td>${(5, 5), (0, 0)}$</td>
<td>$(1, 4), (2, 8)$</td>
</tr>
</tbody>
</table>

$\tilde{c}$ | $pc_{T_1}$ | $pc_{T_2}$ | $pc_{T_3}$ | $\tilde{r}_{T_1}$ | $\tilde{r}_{T_2}$ | $\tilde{r}_{T_3}$ | $\tilde{r}_{T_3}^a$ | $\tilde{r}_{T_3}^a$ | $\tilde{r}_{T_3}^a$ | $(\tilde{\varphi} \ x) \ T_3$ | $(\tilde{\varphi} \ y) \ T_1$ | $(\tilde{\varphi} \ z) \ T_2$ |
Figure 7.5: Communication – Configuration relations.
Table 7.6: Synchronization (Deadlock) – Program.

\[
\begin{align*}
T_1 @ (1, [\text{lock } la]^1; [\text{lock } lb]^2; [\text{unlock } la]^3; [\text{unlock } lb]^4; [\text{halt}]^5) \\
T_2 @ (2, [\text{lock } la]^1; [\text{lock } lb]^2; [\text{halt}]^3)
\end{align*}
\]

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>( \text{ABSTIME}(\tilde{c}, T_1) )</td>
<td>[2, 2]</td>
<td>[1, 2]</td>
<td>[1, 1]</td>
<td>[1, 1]</td>
</tr>
<tr>
<td>( \text{ABSTIME}(\tilde{c}, T_2) )</td>
<td>[1, 2]</td>
<td>[1, 2]</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 \( \text{Lck} = \{la, lb\} \).

Assume that the abstracted timing model, \( \text{ABSTIME}(\tilde{c}, T) \), where \( \tilde{c} @ (\{T, pc_T, \tilde{f}_T, \tilde{\tau}_T\} | T \in \text{Thrd}, \tilde{x}, \tilde{\mu}) \in \text{Conf} \) and \( T \in \text{Thrd} \), 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, \( c_0^0 @ (\{T, pc_T, \tilde{f}_T, \tilde{\tau}_T\} | T \in \text{Thrd}, \tilde{x}, \tilde{\mu}) \), is as described in Table 7.8. Table 7.8 also collects all the configurations derived by \( \text{ABSExe}(\{c_0^0\}, [-\infty, \infty]) \) during the analysis described by \( \text{ANALYSIS}(\{c_0^0\}, [-\infty, \infty]) \). 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 \( c_4^2 \) occurs since \( T_2 \) has been waiting to acquire \( la \) and is now assigned it; \( T_2 \)'s accumulated execution time is updated to account for the concrete spin-waiting (cf. the proof of Lemma 5.58).

It is apparent that \( \text{ABSExe}(\{c_0^0\}, [-\infty, \infty]) = (\{c_2^2\}, \{c_4^2\}, \emptyset) \); i.e. \( c_4^2 \) is a final-state configuration, \( c_4^3 \) is a deadlocked configuration, and there are no timed-out configurations.
discontinues deadlocked configurations that lack concrete counterparts.

This case shows how ABS EXE identifies deadlocked configurations and how it

timed-out configurations.

<table>
<thead>
<tr>
<th>$\tilde{c}$</th>
<th>$pc_{T_1}$</th>
<th>$pc_{T_2}$</th>
<th>$\tilde{t}_{T_1}^a$</th>
<th>$\tilde{t}_{T_2}^a$</th>
<th>$\tilde{1}_a$</th>
<th>$\tilde{1}_b$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\tilde{c}_0^0$</td>
<td>1</td>
<td>1</td>
<td>[0,0]</td>
<td>[0,0]</td>
<td>(unlocked, $\perp_{thrd}$, $\tilde{1}_t$)</td>
<td>(unlocked, $\perp_{thrd}$, $\tilde{1}_t$)</td>
</tr>
<tr>
<td>$\tilde{c}_1^1$</td>
<td>2</td>
<td>1</td>
<td>[2,2]</td>
<td>[0,0]</td>
<td>(locked, $T_1$, $\neginfty$, $\perp_{thrd}$, $\tilde{1}_t$)</td>
<td>(unlocked, $\perp_{thrd}$, $\tilde{1}_t$)</td>
</tr>
<tr>
<td>$\tilde{c}_2^2$</td>
<td>2</td>
<td>1</td>
<td>[2,2]</td>
<td>[0,0]</td>
<td>(locked, $T_1$, $\neginfty$, $\perp_{thrd}$, $\tilde{1}_t$)</td>
<td>(unlocked, $T_2$, $\neginfty$, $\perp_{thrd}$, $\tilde{1}_t$)</td>
</tr>
<tr>
<td>$\tilde{c}_3^3$</td>
<td>3</td>
<td>1</td>
<td>[3,4]</td>
<td>[0,0]</td>
<td>(locked, $T_1$, $\neginfty$, $\perp_{thrd}$, $\tilde{1}_t$)</td>
<td>(locked, $T_1$, $\neginfty$, $\perp_{thrd}$, $\tilde{1}_t$)</td>
</tr>
<tr>
<td>$\tilde{c}_4^4$</td>
<td>4</td>
<td>1</td>
<td>[4,5]</td>
<td>[0,0]</td>
<td>(unlocked, $\perp_{thrd}$, $\neginfty$, $T_1$, [4,5])</td>
<td>(locked, $T_1$, $\neginfty$, $\perp_{thrd}$, $\tilde{1}_t$)</td>
</tr>
<tr>
<td>$\tilde{c}_5^5$</td>
<td>5</td>
<td>1</td>
<td>[5,6]</td>
<td>[0,0]</td>
<td>(unlocked, $T_1$, $\neginfty$, $\perp_{thrd}$, $\tilde{1}_t$)</td>
<td>(unlocked, $\perp_{thrd}$, [5,6])</td>
</tr>
<tr>
<td>$\tilde{c}_6^6$</td>
<td>4</td>
<td>1</td>
<td>[4,5]</td>
<td>[1,2]</td>
<td>(locked, $T_2$, $\neginfty$, $\perp_{thrd}$, $\tilde{1}_t$)</td>
<td>(locked, $T_1$, $\neginfty$, $\perp_{thrd}$, $\tilde{1}_t$)</td>
</tr>
<tr>
<td>$\tilde{c}_7^7$</td>
<td>4</td>
<td>2</td>
<td>[4,5]</td>
<td>[1,2]</td>
<td>(locked, $T_2$, $\neginfty$, $\perp_{thrd}$, $\tilde{1}_t$)</td>
<td>(locked, $T_1$, $\neginfty$, $\perp_{thrd}$, $\tilde{1}_t$)</td>
</tr>
<tr>
<td>$\tilde{c}_8^8$</td>
<td>5</td>
<td>2</td>
<td>[5,6]</td>
<td>[1,2]</td>
<td>(locked, $T_2$, $\neginfty$, $\perp_{thrd}$, $\tilde{1}_t$)</td>
<td>(locked, $T_1$, $\neginfty$, $\perp_{thrd}$, $\tilde{1}_t$)</td>
</tr>
<tr>
<td>$\tilde{c}_9^9$</td>
<td>5</td>
<td>3</td>
<td>[5,6]</td>
<td>[5,18]</td>
<td>(locked, $T_2$, $\neginfty$, $\perp_{thrd}$, $\tilde{1}_t$)</td>
<td>(locked, $T_1$, $\neginfty$, $\perp_{thrd}$, $\tilde{1}_t$)</td>
</tr>
<tr>
<td>$\tilde{c}_{10}^{10}$</td>
<td>1</td>
<td>2</td>
<td>[0,0]</td>
<td>[1,2]</td>
<td>(locked, $T_2$, $\neginfty$, $\perp_{thrd}$, $\tilde{1}_t$)</td>
<td>(locked, $T_1$, $\neginfty$, $\perp_{thrd}$, $\tilde{1}_t$)</td>
</tr>
<tr>
<td>$\tilde{c}_{11}^{11}$</td>
<td>1</td>
<td>2</td>
<td>[0,0]</td>
<td>[1,2]</td>
<td>(locked, $T_2$, $\neginfty$, $\perp_{thrd}$, $\tilde{1}_t$)</td>
<td>(locked, $T_1$, $\neginfty$, $\perp_{thrd}$, $\tilde{1}_t$)</td>
</tr>
<tr>
<td>$\tilde{c}_{12}^{12}$</td>
<td>1</td>
<td>3</td>
<td>[0,0]</td>
<td>[2,4]</td>
<td>(locked, $T_2$, $\neginfty$, $\perp_{thrd}$, $\tilde{1}_t$)</td>
<td>(locked, $T_1$, $\neginfty$, $\perp_{thrd}$, $\tilde{1}_t$)</td>
</tr>
</tbody>
</table>
Figure 7.9: Synchronization (Deadlock) – Configuration relations.
7.3 Synchronization – Deadline Miss

Table 7.10: Synchronization (Deadline miss) – Program.

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_1$@</td>
<td>$(1, [\text{lock } l]^1; [\text{halt}]^2)$</td>
<td></td>
</tr>
<tr>
<td>$T_2$@</td>
<td>$(2, [\text{lock } l]^1; [\text{halt}]^2)$</td>
<td></td>
</tr>
</tbody>
</table>

Table 7.11: Synchronization (Deadline miss) – Timing model.

<table>
<thead>
<tr>
<th></th>
<th>$pc_T$ $(T \in \text{Thrd})$</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABS TIME</td>
<td>1</td>
</tr>
</tbody>
</table>

According to Algorithm 6.13, it is thus easy to see that the estimated timing bounds are:

$$
\begin{align*}
BCET &= 0 \\
WCET &= \infty
\end{align*}
$$

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, $\text{Thrd} = \{T_1, T_2\}$, defined in Table 7.10, it is easy to see that $\text{Reg}_{T_1} = \emptyset$, $\text{Reg}_{T_2} = \emptyset$, $\text{Var} = \emptyset$ and $\text{Lck} = \{1\}$.

Assume that $\text{ABS TIME}(\tilde{c}, T)$, i.e. the abstracted timing model, where $\tilde{c} \in \langle [T, pc_T, \tilde{\pi}_T, \tilde{\tau}_T]_{T \in \text{Thrd}}; \tilde{x}, \tilde{\pi} \rangle \in \text{Conf}$ 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, $\tilde{c}_0 \in \langle [T, pc_T, \tilde{\pi}_T, \tilde{\tau}_T]_{T \in \text{Thrd}}; \tilde{x}, \tilde{\pi} \rangle$, 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 $\text{ABS EXE}(\{\tilde{c}_0\}, [-\infty, \infty])$ during the analysis described by $\text{ANALYSIS}(\{\tilde{c}_0\}, [-\infty, \infty])$. Figure 7.13 shows the relation between the derived configurations. In the figure, deadlocked con-
There are no final-state or timed-out configurations, and can refer to two different memory locations – what location it refers to depends on which thread is considered. It is easy to see that the estimated timing bounds are:

\[
\begin{align*}
BCET &= 0 \\
WCET &= \infty
\end{align*}
\]

### 7.4 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

<table>
<thead>
<tr>
<th>( \tilde{c} )</th>
<th>( pc_{T_1} )</th>
<th>( pc_{T_2} )</th>
<th>( \tilde{t}^a_{T_1} )</th>
<th>( \tilde{t}^a_{T_2} )</th>
<th>( \tilde{1} )</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>(\text{unlocked}, \perp_{\text{thrd}}, \perp_t, \perp_{\text{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>(\text{locked}, T_1, [\infty, 5], \perp_{\text{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>(\text{unlocked}, T_2, [\infty, 5], \perp_{\text{thrd}}, \perp_{t} )</td>
</tr>
</tbody>
</table>

Figures 7.13: Synchronization (Deadline miss) – Configuration relations.
The purpose of the program in Table 7.14 is to increment the value of the variable \( \tilde{c} \) at the end of the loop.

### Table 7.14: Parallel loop – Program.

\[
\begin{align*}
T_1 @ (1, [p := p + 1]^1; & \quad [r := r + 2 \times p + 3]^2; [\text{if } p < 2 \text{ goto } 1]^3; [\text{lock } l]^4; \\
& \quad [\text{load } p \text{ from } x]^5; [p := p + r]^6; [\text{store } p \text{ to } x]^7; [\text{unlock } l]^8; \\
& \quad [\text{halt}]^9)
\end{align*}
\]

\[
\begin{align*}
T_2 @ (2, [p := p + 1]^1; & \quad [r := r + 2 \times p + 3]^2; [\text{if } p < 4 \text{ goto } 1]^3; [\text{lock } l]^4; \\
& \quad [\text{load } p \text{ from } x]^5; [p := p + r]^6; [\text{store } p \text{ to } x]^7; [\text{unlock } l]^8; \\
& \quad [\text{halt}]^9)
\end{align*}
\]

### Table 7.15: 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 \( \text{Thrd}_\ell = \{T_1, T_2\} \) and that there are no global variables when \( \text{Thrd}_\ell = \{T_1\} \) or \( \text{Thrd}_\ell = \{T_2\} \).

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, \( \bar{c}_0 @ ([T, pc_T, \tilde{s}_T, \tilde{t}_T]_{T \in \text{Thrd}}; \bar{x}, \bar{s}_0) \), is as described in Tables 7.16, 7.17 and 7.18. Note that \( p \) and \( r \) for \( T_1 \), and \( x \) 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 \( \text{ABSEXEC}(\{\bar{c}_0\}, [-\infty, \infty]) \). 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 ‘\( \times \)’, and discontinued (invalid) configurations are crossed out. \( \bar{c}_1 \) is discontinued since the timing constraints given by \( \tilde{t}_{T_2} \) (\( \times \)), \( \text{ABSTime}(\bar{c}_1, T_2) = [10, 11] \) (\( \times \)), \([4, 5] = [14, 16] \) and the lock owner assignment deadline, \([\infty, 12]\), give that \( T_2 \) cannot acquire \( 1 \) before \( T_1 \). \( \bar{c}_{12} \) 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 $\hat{t}_T^q \rightarrow_T \hat{t}_T^q \rightarrow_T$.

It is apparent that $\text{ABS EXE}(\{\tilde{c}_0\}, [-\infty, \infty]) = (\{\tilde{c}_{16}\}, 0, 0)$; i.e. $\tilde{c}_{16}$ is a final-state configuration and there are no deadlocked or timed-out configurations. Note that $\tilde{c}_{72}^2$ only exists within the recursively called $\text{ABS EXE}$-instance. According to Algorithm 6.13, it is thus easy to see that the estimated timing bounds are:

$$
\begin{align*}
BCET &= \min\left(\left\{\max\left(\left\{\min\left(\gamma_i(\tilde{t}_T^q)\right) \mid T \in \text{Thrd}\right)\right\} \mid \langle T, p_c T, \tilde{c}_T, \tilde{t}_T^q, \tilde{T}_T, \tilde{a}_T, \tilde{b}_T \rangle \in \{\tilde{c}_{16}\}\right\}\right) = 27 \\
WCET &= \max\left(\left\{\max\left(\left\{\min\left(\gamma_i(\tilde{t}_T^q)\right) \mid T \in \text{Thrd}\right)\right\} \mid \langle T, p_c T, \tilde{c}_T, \tilde{t}_T^q, \tilde{T}_T, \tilde{a}_T, \tilde{b}_T \rangle \in \{\tilde{c}_{16}\}\right\}\right) = 42
\end{align*}
$$
7.4 Parallel Loop 193

Table 7.16: Parallel loop – Configurations (thread-local states).

<table>
<thead>
<tr>
<th>( \tilde{c} )</th>
<th>( p_c T_1 )</th>
<th>( p_c T_2 )</th>
<th>( \tilde{r}_{T_1} )</th>
<th>( \tilde{r}_{T_1} )</th>
<th>( \tilde{r}_{T_2} )</th>
<th>( \tilde{r}_{T_2} )</th>
<th>( \tilde{r}^a_{T_1} )</th>
<th>( \tilde{r}^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}_{71} )</td>
<td>–</td>
<td>3</td>
<td>–</td>
<td>–</td>
<td>[4,4]</td>
<td>[20,20]</td>
<td>–</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>
</tbody>
</table>

\( \tilde{c} \) | \( p_c T_1 \) | \( p_c T_2 \) | \( \tilde{r}_{T_1} \) | \( \tilde{r}_{T_1} \) | \( \tilde{r}_{T_2} \) | \( \tilde{r}_{T_2} \) | \( \tilde{r}^a_{T_1} \) | \( \tilde{r}^a_{T_2} \)
Table 7.17: Parallel loop – Configurations (variable states).

<table>
<thead>
<tr>
<th>( \tilde{c} )</th>
<th>((\tilde{x} \times) \ T_1)</th>
<th>((\tilde{x} \times) \ T_2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \tilde{c}_0 )</td>
<td>{([0,0],[0,0])}</td>
<td>{([0,0],[0,0])}</td>
</tr>
<tr>
<td>( \tilde{c}_1 )</td>
<td>{([0,0],[0,0])}</td>
<td>{([0,0],[0,0])}</td>
</tr>
<tr>
<td>( \tilde{c}_2 )</td>
<td>{([0,0],[0,0])}</td>
<td>{([0,0],[0,0])}</td>
</tr>
<tr>
<td>( \tilde{c}_3 )</td>
<td>{([0,0],[0,0])}</td>
<td>{([0,0],[0,0])}</td>
</tr>
<tr>
<td>( \tilde{c}_4 )</td>
<td>{([0,0],[0,0])}</td>
<td>{([0,0],[0,0])}</td>
</tr>
<tr>
<td>( \tilde{c}_5 )</td>
<td>{([0,0],[0,0])}</td>
<td>{([0,0],[0,0])}</td>
</tr>
<tr>
<td>( \tilde{c}_6 )</td>
<td>{([0,0],[0,0])}</td>
<td>{([0,0],[0,0])}</td>
</tr>
<tr>
<td>( \tilde{c}_7 )</td>
<td>{([0,0],[0,0])}</td>
<td>{([0,0],[0,0])}</td>
</tr>
<tr>
<td>( \tilde{c}_8 )</td>
<td>{([0,0],[0,0])}</td>
<td>{([0,0],[0,0])}</td>
</tr>
<tr>
<td>( \tilde{c}_9 )</td>
<td>{([0,0],[0,0])}</td>
<td>{([0,0],[0,0])}</td>
</tr>
<tr>
<td>( \tilde{c}_{10} )</td>
<td>{([0,0],[0,0]), ([12,12],[14,19])}</td>
<td>{([0,0],[0,0])}</td>
</tr>
<tr>
<td>( \tilde{c}_{11} )</td>
<td>{([0,0],[0,0]), ([12,12],[14,19])}</td>
<td>{([0,0],[0,0])}</td>
</tr>
<tr>
<td>( \tilde{c}_{12} )</td>
<td>{([0,0],[0,0]), ([12,12],[14,19])}</td>
<td>{([0,0],[0,0])}</td>
</tr>
<tr>
<td>( \tilde{c}_{13} )</td>
<td>{([12,12],[14,19])}</td>
<td>{([\tilde{\uparrow}_{val}, \tilde{\downarrow}_r])}</td>
</tr>
<tr>
<td>( \tilde{c}_{14} )</td>
<td>{([12,12],[14,19])}</td>
<td>{([\tilde{\uparrow}_{val}, \tilde{\downarrow}_r])}</td>
</tr>
<tr>
<td>( \tilde{c}_{15} )</td>
<td>{([12,12],[14,19])}</td>
<td>{([32,32],[25,39])}</td>
</tr>
<tr>
<td>( \tilde{c}_{16} )</td>
<td>{([12,12],[14,19])}</td>
<td>{([32,32],[25,39])}</td>
</tr>
</tbody>
</table>

\( \tilde{c} \) | \((\tilde{x} \times) \ T_1\) | \((\tilde{x} \times) \ T_2\) |
Table 7.18: Parallel loop – Configurations (lock states).

<table>
<thead>
<tr>
<th>( \bar{c} )</th>
<th>( \bar{1} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \bar{c}_0 )</td>
<td>( \text{unlocked, } \perp_{\text{thrd}}, \bar{1}<em>t, \perp</em>{\text{thrd}}, \bar{1}_r )</td>
</tr>
<tr>
<td>( \bar{c}_1 )</td>
<td>( \text{unlocked, } \perp_{\text{thrd}}, \bar{1}<em>t, \perp</em>{\text{thrd}}, \bar{1}_r )</td>
</tr>
<tr>
<td>( \bar{c}_2 )</td>
<td>( \text{unlocked, } \perp_{\text{thrd}}, \bar{1}<em>t, \perp</em>{\text{thrd}}, \bar{1}_r )</td>
</tr>
<tr>
<td>( \bar{c}_3 )</td>
<td>( \text{unlocked, } \perp_{\text{thrd}}, \bar{1}<em>t, \perp</em>{\text{thrd}}, \bar{1}_r )</td>
</tr>
<tr>
<td>( \bar{c}_4 )</td>
<td>( \text{unlocked, } \perp_{\text{thrd}}, \bar{1}<em>t, \perp</em>{\text{thrd}}, \bar{1}_r )</td>
</tr>
<tr>
<td>( \bar{c}_5 )</td>
<td>( \text{unlocked, } \perp_{\text{thrd}}, \bar{1}<em>t, \perp</em>{\text{thrd}}, \bar{1}_r )</td>
</tr>
<tr>
<td>( \bar{c}_6 )</td>
<td>( \text{unlocked, } \perp_{\text{thrd}}, \bar{1}<em>t, \perp</em>{\text{thrd}}, \bar{1}_r )</td>
</tr>
<tr>
<td>( \bar{c}_7 )</td>
<td>( \text{unlocked, } T_2, [-\infty, 12], \perp_{\text{thrd}}, \bar{1}_r )</td>
</tr>
<tr>
<td>( \bar{c}_7^1 )</td>
<td>( \text{locked, } T_1, [-\infty, 12], \perp_{\text{thrd}}, \bar{1}_r )</td>
</tr>
<tr>
<td>( \bar{c}_7^2 )</td>
<td>( \text{locked, } T_1, [-\infty, 12], \perp_{\text{thrd}}, \bar{1}_r )</td>
</tr>
<tr>
<td>( \bar{c}_7^{12} )</td>
<td>( \text{locked, } T_1, [-\infty, 12], \perp_{\text{thrd}}, \bar{1}_r )</td>
</tr>
<tr>
<td>( \bar{c}_8 )</td>
<td>( \text{locked, } T_1, [-\infty, 12], \perp_{\text{thrd}}, \bar{1}_r )</td>
</tr>
<tr>
<td>( \bar{c}_9 )</td>
<td>( \text{locked, } T_1, [-\infty, 12], \perp_{\text{thrd}}, \bar{1}_r )</td>
</tr>
<tr>
<td>( \bar{c}_{10} )</td>
<td>( \text{locked, } T_1, [-\infty, 12], \perp_{\text{thrd}}, \bar{1}_r )</td>
</tr>
<tr>
<td>( \bar{c}_{11} )</td>
<td>( \text{unlocked, } \perp_{\text{thrd}}, [-\infty, 12], T_1, [16, 22] )</td>
</tr>
<tr>
<td>( \bar{c}_{12}^1 )</td>
<td>( \text{unlocked, } T_1, [-\infty, 28], T_1, [16, 22] )</td>
</tr>
<tr>
<td>( \bar{c}_{12}^2 )</td>
<td>( \text{locked, } T_2, [-\infty, 28], T_1, [16, 22] )</td>
</tr>
<tr>
<td>( \bar{c}_{13} )</td>
<td>( \text{locked, } T_2, [-\infty, 28], T_1, [16, 22] )</td>
</tr>
<tr>
<td>( \bar{c}_{14} )</td>
<td>( \text{locked, } T_2, [-\infty, 28], T_1, [16, 22] )</td>
</tr>
<tr>
<td>( \bar{c}_{15} )</td>
<td>( \text{locked, } T_2, [-\infty, 28], T_1, [16, 22] )</td>
</tr>
<tr>
<td>( \bar{c}_{16} )</td>
<td>( \text{unlocked, } \perp_{\text{thrd}}, [-\infty, 28], T_2, [27, 42] )</td>
</tr>
</tbody>
</table>
In this chapter, some distinguishing properties of the defined analysis will be discussed. Feedback on the Research Questions and issues to be further considered and investigated will also be given.

8.1 The Underlying Architecture

The analysis is defined for an arbitrary underlying architecture (that is, however, restricted to the constraints in 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 [55,102]. The lock- and unlock-statements also trivially correspond to higher level spin-locking primitives, such as those provided by...
Chapter 8

Conclusions

In this chapter, some distinguishing properties of the defined analysis will be discussed. Feedback on the Research Questions and issues to be further considered and investigated will also be given.

8.1 The Underlying Architecture

The analysis is defined for an arbitrary underlying architecture (that is, however, restricted to the constraints in 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 [55, 102]. The `lock`- and `unlock`-statements also trivially correspond to higher level spin-locking primitives, such as those provided by
the POSIX thread library [16, 53].

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 \text{store}, then the \text{lock}- and \text{unlock}-statements could be regarded as macros without timing that should encapsulate all \text{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 or PowerPC architectures [6, 56]. 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 8.3 for a discussion on more nonterminating cases.

### 8.2 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 5.10 and 5.11). The advantage of this approach (i.e. abstracting time using intervals) in conjunction with the defined domain for variable states (cf. Section 5.5) is that a relatively high precision is achieved. And, when \( |\text{Thrd}| = 1 \), the analysis result will be equivalent to that of the corresponding sequential ana-
lysis (cf. [40]). 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.

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 can cause complexity problems. The result of not abstracting some parts of a state is that (at least) all the concrete counterparts must be evaluated. Any reasonably precise abstractions of the parts of the lock states that are currently kept concrete have not been found. In case of complexity issues, candidate domains for such abstractions must be further investigated.

It should be apparent that a given (abstract) configuration could result in two or more configurations for each thread issuing an \texttt{if} - or a \texttt{lock}-statement in a transition (cf. Tables 5.10 and 5.11). 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 [37]. 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. As discussed above, domains for abstracting the currently concrete parts of the configuration could be further investigated in case of complexity issues. Abstracting more parts of the configurations would also increase the possibility that merging could be more frequently performed.
It is very important to note that several configurations that lack valid concrete counterparts (cf. Definition 4.4) are added to the work-list 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 8.1 (cf. the example in Section 7.3). T1 is obviously the thread issuing lock lck first in any considered case. 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 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 8.2a, assuming that the timing of the first lock-statement in the two threads overlap (cf. the case study 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 \( \tilde{c} \in \tilde{\text{Conf}} \) and that \( pc_{T_1} = pc_{T_2} = 1 \). The resulting lock-owner assignments (i.e. configurations) are given in Figure 8.2b. Obviously, only \( \tilde{c}_{11} \) and \( \tilde{c}_{22} \) have valid concrete counterparts. \( \tilde{c}_{12} \) and \( \tilde{c}_{21} \) will be discontinued (i.e. removed from the work-list) 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 \( \tilde{c}_{12} \) and \( \tilde{c}_{21} \) were not discontinued, the analysis would itself deadlock.

An important point to notice from the above discussion is that well-
$$T_1 : [\text{lock } lck]^1; [\text{lock } lck']^2; \ldots$$
$$T_2 : [\text{lock } lck]^1; [\text{lock } lck']^2; \ldots$$

(a) The program described by $\tilde{c}$.

$$\text{OWN}(\tilde{l}_1 \ lck) = T_1$$
$$\text{OWN}(\tilde{l}_2 \ lck) = T_2$$

$$\tilde{c}_1$$

$$\tilde{c}_2$$

$$\text{OWN}(\tilde{l}_{11} \ lck') = T_1$$
$$\text{OWN}(\tilde{l}_{12} \ lck') = T_2$$
$$\text{OWN}(\tilde{l}_{21} \ lck') = T_1$$
$$\text{OWN}(\tilde{l}_{22} \ lck') = T_2$$

(b) Resulting configurations.

Figure 8.2: Lock owner assignments based on $\tilde{c} \in \text{Conf}$ resulting in two valid and two invalid (i.e. falsely deadlocked) configurations.

structured concurrent programs are less complex to analyze compared to less well-structured concurrent programs [69]. 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.

Another important point to notice is that the complexity is lowered by keeping a high precision in the calculation of the accumulated 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. 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 hard-
ware 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.

### 8.3 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($\tilde{c}, \tilde{t}_{to}$), given that time moves forward and the timeout is finite; i.e. it cannot be that $0 \in \text{ABS TIME}(\tilde{c}', T)$ for all $\tilde{c}' \in \text{Conf}$ occurring in the loop in $T$ and $\max(\gamma_l(\tilde{t}_{to})) = \infty$. If ABSTIME includes 0 for all statements of an infinite loop in some thread, then the algorithm will not terminate.

To avoid part of this problem, another timeout variable could be added to the analysis. This timeout could be used to identify that the upper bound of a single thread’s accumulated execution time has reached a limit. However, this does not resolve the case that, for all $\tilde{c}' \in \text{Conf}$ in the loop, ABSTIME($\tilde{c}', T$) = [0,0].

To address this case, a transition counter could be used. There could be one counter for each thread individually and/or one counter for all threads combined. The counter(s) could either count all transitions or only transitions that are consecutively done in [0,0] amount of 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.

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.

### 8.4 The Research Questions

**Question 1:** "How can safe and tight bounds on the execution time of a concurrent program 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. Further evaluation, preferably based on an implementation of the analysis, must be performed before the tightness of the approach used in this thesis can be commented upon.

**Question 2:** "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 and further abstraction might have to be incorporated.

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.

To completely answer this question, further investigation is required.

**Question 3:** "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. Further techniques should be investigated and could most probably be derived based on an implementation and evaluation of the analysis. One such example, as discussed above, is to include a second timeout variable in the analysis. Another example, as also discussed above, is to include a transition counter.
8.4 The Research Questions

**Question 1:** “How can safe and tight bounds on the execution time of a concurrent program 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. Further evaluation, preferably based on an implementation of the analysis, must be performed before the tightness of the approach used in this thesis can be commented upon.

**Question 2:** “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 and further abstraction might have to be incorporated.

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.

To completely answer this question, further investigation is required.

**Question 3:** “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. Further techniques should be investigated and could most probably be derived based on an implementation and evaluation of the analysis. One such example, as discussed above, is to include a second timeout variable in the analysis. Another example, as also discussed above, is to include a transition counter.
The techniques discussed above (a second timeout combined with a transition counter) should basically guarantee termination of the analysis even if the analyzed program might not terminate; provided that suitable timeout-limits are chosen. But note that this is still an open question for the analysis presented in this thesis.

**Question 4:** “*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.

### 8.5 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 timeout input argument. There are many methods and tools capable of finding deadlocks in concurrent programs available [11, 19, 22, 32, 50, 51, 80, 103, 114], but none of them includes the timing properties of the analyzed program as far as the author knows. 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.
8.6 Future Work

Some concrete tasks that currently are being performed, and will be performed in the near future, are to implement and evaluate the analysis presented in this thesis. The analysis is currently being implemented in the ERLANG programming language [7, 62].

The evaluation will be performed on some suitable benchmark suite of concurrent programs. Such a suite is currently being established within the TACLe EU COST Action [108] (a European network of leading researchers within the field of WCET analysis) and will include concurrent versions of some of the programs in the Mälardalen WCET Benchmark suite [36]. The benchmark suite should include different types of concurrent programs, each of them stressing the analysis in a different way.

It is expected that the evaluation will result in hints pointing to some parts of the analysis that suffer from complexity problems. Thus, improvements and strategies for complexity reduction for these parts should be derived and implemented.

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 [75], LLVM [109], ALF [38, 39], ARM [6] and PowerPC [56]. It could also be possible to make the implementation a flexible framework which could allow the analyzed instruction set to be switched. This could be done by dividing instructions into special classes.

Some model of the underlying architecture (i.e. the function ABSTIME) must also be derived. Since the focus of this thesis has excluded a formal definition of ABSTIME, some very simple, and perhaps even non-realistic, timing model will most probably be used. Several different timing models should be evaluated to investigate how the characteristics of their definitions affect the complexity of the analysis.

Another interesting topic is to investigate how using a more well-structured (i.e. restricted) programming model could help alleviate the potential complexity and precision problems. One such example could be to replace the lock and unlock primitives with a barrier synchronization primitive which could be used to synchronize all (or a subset of the) threads in a program.


Bibliography


208 Bibliography


[37] J. Gustafsson and A. Ermedahl. Merging techniques for faster derivation of WCET flow information using abstract execution. In R. Kirner,


[77] MERASA. MERASA project, 2013.


Appendix A: Notation & Nomenclature

Exp 1 and Exp 2 denote the same thing, often a short and a long notation for a configuration.

\[ b : \text{Exp} 1, \text{otherwise Exp} 2. \]

\( (o_1, ..., o_n) \) Ordinary tuple containing \( n \) elements.

\[ \langle o_1, ..., o_n \rangle \] Special tuple containing \( n \) elements. Used to denote complete lattices, Galois connections, configurations, etc.

\[ [o_1, ..., o_n] \in \{e_1, ..., e_m\} \] Expands to \( o_11, ..., o_1n, ..., o_m1, ..., o_mn \); i.e. one instance of \( o_1, ..., o_n \) for each \( e \in \{e_1, ..., e_m\} \). Used inside special tuples.

\( S \) An arbitrary set (capitalized, italic notation).

\( S \) A standard set (capitalized, blackboard bold notation); e.g. \( \mathbb{Z} \).

Set \( A \) set of analysis-specific elements (first letter capitalized, bold notation); e.g. \( \text{Thrd} \).

\( P(S) \) The powerset of \( S \); i.e. \( \{S' | S' \subseteq S\} \).

\( S \times S' \) The Cartesian product; i.e. \( \{(e, e') | e \in S \wedge e' \in S'\} \).

\( \prod_{e \in \{e_1, ..., e_m\}} (\text{Exp}(e)) \) Expands to \( \text{Exp}(e_1) \times ... \times \text{Exp}(e_m) \).
Appendix A

Notation & Nomenclature

\( \exp_1 @ \exp_2 \) \quad \exp_1 \) and \( \exp_1 \) denote the same thing, often a short and a long notation for a configuration.

\( b ? \exp_1 : \exp_2 \) \quad If \( b \), then \( \exp_1 \), otherwise \( \exp_2 \).

\( (o_1, \ldots, o_n) \) \quad Ordinary tuple containing \( n \) elements.

\( \langle o_1, \ldots, o_n \rangle \) \quad Special tuple containing \( n \) elements. Used to denote complete lattices, Galois connections, configurations, etc.

\[ o_1, \ldots, o_n \]_{e \in \{e_1, \ldots, e_m\}} \quad Expands to \( o_1^e, \ldots, o_n^e \), i.e. one instance of \( o_1, \ldots, o_n \) for each \( e \in \{e_1, \ldots, e_m\} \). Used inside special tuples.

\( S \) \quad An arbitrary set (capitalized, italic notation).

\( \mathbb{S} \) \quad A standard set (capitalized, blackboard bold notation); e.g. \( \mathbb{Z} \).

\( \textbf{Set} \) \quad A set of analysis-specific elements (first letter capitalized, bold notation); e.g. \( \textbf{Thrd} \).

\( \mathcal{P}(S) \) \quad The powerset of \( S \); i.e. \( \{S' \mid S' \subseteq S\} \).

\( S \times S' \) \quad The Cartesian product; i.e. \( \{(e, e') \mid e \in S \land e' \in S'\} \).

\( \Pi_{e \in \{e_1, \ldots, e_m\}}(\exp(e)) \) \quad Expands to \( \exp(e_1) \times \ldots \times \exp(e_m) \).
\( e, e' \in S \) Short for \( e \in S \wedge e' \in S \).

\( \lambda e \in S. \text{exp} \) Lambda notation: a function from \( e \), which is an element of \( S \), to \( \text{exp} \), 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.

\( \exists \) Denotes a state (i.e. a function/mapping from elements to values); e.g. \( \pi \).

\( \exists[s' \mapsto \text{exp}] \) Remapping. Defined as: \( \exists[s' \mapsto \text{exp}] s = \begin{cases} \text{exp} & \text{if } s = s' \\ \exists s & \text{otrwh} \end{cases} \)

\( ((\exists s_1) s_2) \leftarrow \ldots \) A shorthand for \( f \leftarrow \exists[s_1 \mapsto (\exists s_1)[s_2 \mapsto \ldots]] \)

\text{ALG.Func} A function defined in a table or algorithm.

\( T \) One of the threads defined in the analyzed program.

\( \Pi, \text{Thrd} \) 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 variables defined in the program.

\( lck \) Lock (shared resource).

\( \text{Lck} \) The locks defined in the program.

\( pc \) Program counter (unique for each thread).

\( \bar{f} \) \( f \) defined in some abstract domain; i.e. an abstraction of \( f \).

\( \tau, \bar{\tau} \) Mapping from registers to their values (unique for each thread).

\( t^a, \bar{t}^a \) Accumulated execution time (unique for each thread).

\( \chi, \bar{\chi} \) 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 .................................. 51
4.3 TIME is non-zero when spin-locking .................. 52
5.51 ABSTIME is safe and non-negative .................... 105
Appendix C  
List of Definitions

3.1 Monotone function ....................... 22
3.2 Completely additive function .................. 23
3.3 Completely multiplicative function ............... 23
3.9 Galois connection ........................ 26
3.10 Galois insertion ......................... 27
3.11 Induced function ........................ 27
3.12 Adjunction ........................... 27
3.26 Partial order ........................... 38
3.27 Greatest lower bound ...................... 38
3.28 Least upper bound ........................ 38
3.29 Abstraction function, \(\alpha\) ..................... 38
3.30 Alternative definition – Concretization function, \(\gamma\) .... 38
3.31 Interval ............................. 39
3.32 Concretization of interval .................... 39
3.33 Partial order for intervals .................... 39
3.34 Greatest lower bound for intervals ............... 39
3.35 Least upper bound for intervals ................. 40
3.36 Abstraction to interval ..................... 40
4.4 Valid concrete configuration .................. 55
4.7 Collecting semantics ...................... 58
5.1 Concretization of an abstract register state ........... 60
5.2 Partial order for abstract register states ............. 62
### Appendix C

#### List of Definitions

<table>
<thead>
<tr>
<th>Section</th>
<th>Definition</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Monotone function</td>
<td>22</td>
</tr>
<tr>
<td>3.2</td>
<td>Completely additive function</td>
<td>23</td>
</tr>
<tr>
<td>3.3</td>
<td>Completely multiplicative function</td>
<td>23</td>
</tr>
<tr>
<td>3.9</td>
<td>Galois connection</td>
<td>26</td>
</tr>
<tr>
<td>3.10</td>
<td>Galois insertion</td>
<td>27</td>
</tr>
<tr>
<td>3.11</td>
<td>Induced function</td>
<td>27</td>
</tr>
<tr>
<td>3.12</td>
<td>Adjunction</td>
<td>27</td>
</tr>
<tr>
<td>3.26</td>
<td>Partial order</td>
<td>38</td>
</tr>
<tr>
<td>3.27</td>
<td>Greatest lower bound</td>
<td>38</td>
</tr>
<tr>
<td>3.28</td>
<td>Least upper bound</td>
<td>38</td>
</tr>
<tr>
<td>3.29</td>
<td>Abstraction function, $\alpha$</td>
<td>38</td>
</tr>
<tr>
<td>3.30</td>
<td>Alternative definition – Concretization function, $\gamma$</td>
<td>38</td>
</tr>
<tr>
<td>3.31</td>
<td>Interval</td>
<td>39</td>
</tr>
<tr>
<td>3.32</td>
<td>Concretization of interval</td>
<td>39</td>
</tr>
<tr>
<td>3.33</td>
<td>Partial order for intervals</td>
<td>39</td>
</tr>
<tr>
<td>3.34</td>
<td>Greatest lower bound for intervals</td>
<td>39</td>
</tr>
<tr>
<td>3.35</td>
<td>Least upper bound for intervals</td>
<td>40</td>
</tr>
<tr>
<td>3.36</td>
<td>Abstraction to interval</td>
<td>40</td>
</tr>
<tr>
<td>4.4</td>
<td>Valid concrete configuration</td>
<td>55</td>
</tr>
<tr>
<td>4.7</td>
<td>Collecting semantics</td>
<td>58</td>
</tr>
<tr>
<td>5.1</td>
<td>Concretization of an abstract register state</td>
<td>60</td>
</tr>
<tr>
<td>5.2</td>
<td>Partial order for abstract register states</td>
<td>62</td>
</tr>
</tbody>
</table>
### Appendix C. List of Definitions

<table>
<thead>
<tr>
<th>Section</th>
<th>Definition</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.3</td>
<td>Greatest lower bound of abstract register states</td>
<td>62</td>
</tr>
<tr>
<td>5.4</td>
<td>Least upper bound of abstract register states</td>
<td>62</td>
</tr>
<tr>
<td>5.5</td>
<td>Abstraction of a set of register states</td>
<td>62</td>
</tr>
<tr>
<td>5.7</td>
<td>Abstract evaluation of arithmetic expressions</td>
<td>63</td>
</tr>
<tr>
<td>5.8</td>
<td>Boolean restriction</td>
<td>63</td>
</tr>
<tr>
<td>5.9</td>
<td>Concretization of an abstract variable state</td>
<td>74</td>
</tr>
<tr>
<td>5.10</td>
<td>Abstraction of a set of variable states</td>
<td>74</td>
</tr>
<tr>
<td>5.12</td>
<td>Partial order of writes</td>
<td>76</td>
</tr>
<tr>
<td>5.13</td>
<td>Least upper bound of writes</td>
<td>76</td>
</tr>
<tr>
<td>5.14</td>
<td>Time precedence</td>
<td>77</td>
</tr>
<tr>
<td>5.15</td>
<td>Partial order for abstract variable states</td>
<td>77</td>
</tr>
<tr>
<td>5.16</td>
<td>Greatest lower bound of abstract variable states</td>
<td>77</td>
</tr>
<tr>
<td>5.17</td>
<td>Least upper bound of abstract variable states</td>
<td>78</td>
</tr>
<tr>
<td>5.18</td>
<td>Time of most recent write</td>
<td>78</td>
</tr>
<tr>
<td>5.19</td>
<td>Safe write history</td>
<td>78</td>
</tr>
<tr>
<td>5.20</td>
<td>Safe value of $x$ as seen by thread T</td>
<td>79</td>
</tr>
<tr>
<td>5.21</td>
<td>Safe partial order of abstract variable states</td>
<td>79</td>
</tr>
<tr>
<td>5.22</td>
<td>Safe lower bound of abstract variable states</td>
<td>82</td>
</tr>
<tr>
<td>5.23</td>
<td>Safe upper bound of abstract variable states</td>
<td>82</td>
</tr>
<tr>
<td>5.29</td>
<td>Concretization of an abstract lock state</td>
<td>91</td>
</tr>
<tr>
<td>5.30</td>
<td>Abstraction of a set of lock states</td>
<td>91</td>
</tr>
<tr>
<td>5.31</td>
<td>Partial order of abstract lock states</td>
<td>92</td>
</tr>
<tr>
<td>5.32</td>
<td>Greatest lower bound of abstract lock states</td>
<td>93</td>
</tr>
<tr>
<td>5.33</td>
<td>Least upper bound of abstract lock states</td>
<td>93</td>
</tr>
<tr>
<td>5.36</td>
<td>Concretization of an abstract configuration</td>
<td>96</td>
</tr>
<tr>
<td>5.37</td>
<td>Partial ordering of two abstract configurations</td>
<td>96</td>
</tr>
<tr>
<td>5.39</td>
<td>Greatest lower bound for two abstract configurations</td>
<td>97</td>
</tr>
<tr>
<td>5.40</td>
<td>Least upper bound for two abstract configurations</td>
<td>97</td>
</tr>
<tr>
<td>5.41</td>
<td>Abstraction of a set of configurations</td>
<td>97</td>
</tr>
<tr>
<td>5.43</td>
<td>Abstraction of a set of axiom input configurations</td>
<td>100</td>
</tr>
<tr>
<td>5.44</td>
<td>Concretization of an abstract axiom input configuration</td>
<td>100</td>
</tr>
<tr>
<td>5.45</td>
<td>Abstraction of a set of axiom output configurations</td>
<td>100</td>
</tr>
<tr>
<td>5.46</td>
<td>Concretization of an abstract axiom output configuration</td>
<td>100</td>
</tr>
<tr>
<td>5.49</td>
<td>Soundness of the abstract axiom transition relation</td>
<td>101</td>
</tr>
<tr>
<td>6.9</td>
<td>BCET and WCET</td>
<td>177</td>
</tr>
</tbody>
</table>
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 Galois connection between concrete and abstract domains. .. 10

4.6 Illustration of how \texttt{Thrd}_{\texttt{exe}} is determined. .......... 53

5.8 The timestamps of the writes in \tilde{x} considered by READ. ... 88
5.12 Abstract lock state transitions. ................................ 123

6.1 Relation between concrete and abstract transitions .......... 156
6.2 Timeout for recursion in ABSEXES. .............................. 171

7.5 Communication – Configuration relations. ....................... 185
7.9 Synchronization (Deadlock) – Configuration relations. ...... 188
7.13 Synchronization (Deadline miss) – Configuration relations. 190
7.19 Parallel loop – Configuration relations. ......................... 196

8.1 Lock owner assignments based on \tilde{c} \in \text{	exttt{C}}_{\text{	exttt{Conf}}} resulting in one valid and one invalid configuration. . 200
8.2 Lock owner assignments based on \tilde{c} \in \text{	exttt{C}}_{\text{	exttt{Conf}}} resulting in two valid and two invalid (i.e. falsely deadlocked) configurations. . 201
Appendix E
List of Tables

4.1 The Syntax of PPL ................................ 44
4.2 Semantics of concrete axiom transitions ............ 48
4.3 Semantics of concrete program transitions ........... 49
4.4 Definition of STM and LABELS ................. 50
4.5 Definition of STT, OWN, DL, POWN and REL .. 50
4.7 Semantics of concrete evaluation of arithmetic expressions .. 53
4.8 Semantics of concrete evaluation of boolean expressions ... 54
5.1 PPL operators defined for interval arguments ......... 61
5.1 Cont. PPL operators defined for interval arguments ..... 62
5.2 The abstract function evaluating arithmetic expressions. .... 64
5.3 Boolean restriction for intervals ................ 65
5.4 Arithmetic restriction for intervals ............... 66
5.5 Multiplication operator for inverting interval division expressions. 69
5.6 Division operator for inverting interval multiplication expressions. 70
5.7 Division operator for inverting interval division expressions. 72
5.9 Definition of ˜STT, ˜OWN, ˜DL, ˜POWN and ˜REL ......... 92
5.10 Semantics of abstract axiom transitions ............ 102
5.11 Semantics of abstract program transitions ........... 106
7.1 Communication – Program ................... 181
7.2 Communication – Timing model ................ 182

231
Appendix E

List of Tables

4.1 The Syntax of PPL. ........................................ 44
4.2 Semantics of concrete axiom transitions. ............... 48
4.3 Semantics of concrete program transitions. ............... 49
4.4 Definition of STM and LABELS. .......................... 50
4.5 Definition of STT, OWN, DL, POWN and REL. ......... 50
4.7 Semantics of concrete evaluation of arithmetic expressions. .. 53
4.8 Semantics of concrete evaluation of boolean expressions. . . 54

5.1 PPL operators defined for interval arguments. .......... 61
5.1 Cont. PPL operators defined for interval arguments. .... 62
5.2 The abstract function evaluating arithmetic expressions. . . 64
5.3 Boolean restriction for intervals. ........................ 65
5.4 Arithmetic restriction for intervals. ...................... 66
5.5 Multiplication operator for inverting interval division expressions. ........................................ 69
5.6 Division operator for inverting interval multiplication expressions. ........................................ 70
5.7 Division operator for inverting interval division expressions. .. 72
5.9 Definition of STT, OWN, DL, POWN and REL. ....... 92
5.10 Semantics of abstract axiom transitions. ................ 102
5.11 Semantics of abstract program transitions. .............. 106

7.1 Communication – Program. .............................. 181
7.2 Communication – Timing model ........................... 182
7.3 Communication – Configurations (first half) ............ 183
7.4 Communication – Configurations (second half) .......... 184
7.6 Synchronization (Deadlock) – Program .................. 186
7.7 Synchronization (Deadlock) – Timing model .......... 186
7.8 Synchronization (Deadlock) – Configurations .......... 187
7.10 Synchronization (Deadline miss) – Program .......... 189
7.11 Synchronization (Deadline miss) – Timing model .... 189
7.12 Synchronization (Deadline miss) – Configurations .... 190
7.14 Parallel loop – Program .......................... 191
7.15 Parallel loop – Timing model .................. 191
7.16 Parallel loop – Configurations (thread-local states) .. 193
7.17 Parallel loop – Configurations (variable states) .... 194
7.18 Parallel loop – Configurations (lock states) .......... 195
Appendix F

List of Algorithms

5.1 Partial order of abstract variable states .................................. 80
5.2 Earliest write for a thread ......................................................... 81
5.3 Meeting two abstract variable states ............................................. 83
5.4 Joining two abstract variable states .............................................. 84
5.5 Write to variable ................................................................. 85
5.6 Read from variable ............................................................. 86
5.7 Time of most recent write ......................................................... 86
5.8 Time of most recent write in thread ........................................... 86
5.9 Trim variable state ............................................................... 89
5.10 Split set of writes ............................................................. 89
5.11 Determine deadline for lock owner assignment ...................... 107
5.12 Determine accumulated execution time ................................ 107
5.12 Cont. Determine accumulated execution time .................... 108
6.1 Abstract execution .......................................................... 157
6.2 Choose an element .......................................................... 159
6.3 Determine if graph has cycles .................................................. 159
6.4 Threads to execute in an abstract configuration .................... 161
6.5 Global variables in an abstract configuration ....................... 161
6.6 Threads executing a possibly unsafe load-statement ............ 162
6.7 Final abstract configuration .................................................. 163
6.8 Deadlocked abstract configuration ....................................... 163
6.9 Timed-out abstract configuration ......................................... 165
Appendix F. List of Algorithms

6.10 Valid abstract configuration ........................................... 166
6.11 Get variable in load-statement ...................................... 167
6.12 Get register in load-statement ...................................... 168
6.13 BCET/WCET analysis .................................................... 178
Appendix G

List of Lemmas

3.4 Completely multiplicative functions ....................... 23
3.14 Relation between $\alpha$ and $\gamma$ .......................... 28
3.15 Galois connection – Existence .................................. 29
3.18 Monotonicity of $\alpha_{\theta}$ ...................................... 31
3.19 Monotonicity of $\gamma_{\theta}$ ...................................... 32
3.23 Monotonicity of $\gamma_s$ ......................................... 34
3.37 Monotonicity of $\gamma_{\text{int}}$ ................................... 40
3.38 Monotonicity of $\alpha_{\text{int}}$ ................................... 40

4.2 Time only moves forward ....................................... 51
4.5 $\rightarrow_{prg}$ preserves lock state validity ...................... 56
4.6 Properties of $\bar{Y}'$ ............................................... 57

5.24 Soundness of WRITE ........................................... 82
5.25 Soundness of MOSTRECENTWRITEWITEMTHREAD .... 85
5.26 Soundness of MOSTRECENTWRITETIME .................. 85
5.27 Soundness of READ ............................................... 87
5.28 Soundness of TRIM .............................................. 88
5.34 Monotonicity of $\gamma_{\text{lock}}$ ................................. 94
5.38 Monotonicity of $\gamma_{\text{conf}}$ ................................. 96
5.50 Soundness of $\overrightarrow{ax}$ .................................. 101
5.52 Time accumulation ............................................ 111
5.53 Thread isolation ............................................... 111
Appendix G. List of Lemmas

5.54 Soundness of DLLOCK ........................................ 113
5.55 Partial soundness of ACCTIME ................................. 118
5.56 Properties of owner assignment for lock-transitions .......... 123
5.57 Soundness of \( \frac{\text{prg}}{\text{pg}} \), no frozen thread .......................... 127
5.58 Soundness of \( \frac{\text{prg}}{\text{pg}} \), frozen thread .......................... 138
5.59 Soundness of \( \frac{\text{prg}}{\text{pg}} \), final state .................................. 149

6.1 Soundness of CYCLE ............................................. 159
6.2 Soundness of EXETHRD .......................................... 160
6.3 Soundness of GLOBALVAR ...................................... 161
6.4 Soundness of EXELOADTHRD ................................... 162
6.5 Soundness of ISDEADLOCK ..................................... 163
6.6 Soundness of ISTIMEOUT ....................................... 164
6.7 Soundness of ISVALID .......................................... 165
Appendix H

List of Theorems

3.5 Complete lattice – Lifting .................................. 24
3.6 Complete lattice – Cartesian product ......................... 24
3.7 Complete lattice – Total function space ...................... 25
3.8 Complete lattice – Monotone function space ................. 26
3.13 Adjunctions and Galois connections ......................... 27
3.16 Galois connection – Independent attribute method ...... 30
3.17 Galois connection – Lifted independent attribute method .. 30
3.20 Galois connection – Double lifting .......................... 32
3.21 Not a Galois connection – Double lifting .................... 33
3.22 Galois connection – Function space ......................... 34
3.24 Galois connection – Lifted function space ................. 35
3.25 Galois connection – Indexing ................................ 36
3.39 Galois insertion – Intervals ................................ 41
5.6 Galois connection – Register states .......................... 63
5.11 Galois connection – Variable states ......................... 74
5.35 Galois connection – Lock states ............................ 94
5.42 Galois connection – Configurations ......................... 98
5.47 Galois connection – Axiom input configurations .......... 100
5.48 Galois connection – Axiom output configurations ........ 101
6.8 Soundness of ABSExe ...................................... 168
6.10 Soundness of ANALYSIS ................................ 177
Index

abstract domain, 9, 26

Abstract execution, 12, 14, 16, 203

Abstract interpretation, 9, 16, 18, 21, 46, 203

abstraction, 9, 12, 14, 26, 54, 59, 63, 155, 197, 199, 222

anti-symmetric relation, see relation

BCET, BCET, 3–5, 7, 11–13, 16, 177, 179, 180, 182, 189, 190, 224

Best-Case Execution Time, see BCET, BCET

bottom element, 22

bounds lower, 22

least, 22

greatest, 22

upper, 22

least upper bound, see bounds

Galois connection, 9, 26

global memory, see variable

halting-problem, 7

high-level analysis, 5

flow analysis, 5

function completely additive, 22

completely multiplicative, 22

monotone, 22

partial, 22

total, 22

Greatest lower bound, see bounds

lock, 43

low-level analysis, 5

lower bound, see bounds

Mälardalen WCET Benchmark suite, 205

Model checking, 1, 8, 16, 18

Bounded, 18

local memory, see register

Model checking, 1, 8, 16, 18

Bounded, 18

Mälardalen WCET Benchmark suite, 205

Model checking, 1, 8, 16, 18

Bounded, 18

Mälardalen WCET Benchmark suite, 205

Model checking, 1, 8, 16, 18

Bounded, 18

Mälardalen WCET Benchmark suite, 205

Model checking, 1, 8, 16, 18

Bounded, 18

Mälardalen WCET Benchmark suite, 205

Model checking, 1, 8, 16, 18

Bounded, 18

Mälardalen WCET Benchmark suite, 205

Model checking, 1, 8, 16, 18

Bounded, 18

Mälardalen WCET Benchmark suite, 205

Model checking, 1, 8, 16, 18

Bounded, 18

Mälardalen WCET Benchmark suite, 205

Model checking, 1, 8, 16, 18

Bounded, 18

Mälardalen WCET Benchmark suite, 205

Model checking, 1, 8, 16, 18

Bounded, 18

Mälardalen WCET Benchmark suite, 205

Model checking, 1, 8, 16, 18

Bounded, 18

Mälardalen WCET Benchmark suite, 205
Index

abstract domain, 9, 26
Abstract execution, 12, 14, 16, 203
Abstract interpretation, 9, 16, 18, 21, 46, 203
abstraction, 9, 12, 14, 26, 54, 59, 63, 155, 197, 199, 222
anti-symmetric relation, see relation
BCET, BCET, 3–5, 7, 11–13, 16, 177, 179, 180, 182, 189, 190, 224
BCET, BCET, 178
Best-Case Execution Time, see BCET, BCET
bottom element, 22
bounds
  lower, 22
  greatest, 22
  upper, 22
  least, 22
calculation, 5
completely additive function, see function
completely multiplicative function, see function
concrete domain, 26
COST Action, 205
dynamic analysis, 5
embedded system, 1
estimation
  safe, 3, 5
  tight, 4, 5, 10
fixed-point calculation, 109
flow analysis, 5
function
  completely additive, 22
  completely multiplicative, 22
  monotone, 22
  partial, 22
  total, 22
Galois connection, 9, 26
global memory, see variable
greatest lower bound, see bounds
halting-problem, 7
high-level analysis, 5
least upper bound, see bounds
local memory, see register
lock, 43
low-level analysis, 5
lower bound, see bounds
Mälardalen WCET Benchmark suite, 205
Model checking, 1, 8, 16, 18
  Bounded, 18
Symbolic, 18
monotone function, see function
multi-core, 2, 201
multi-core CPU, 2, 4, 11, 16, 17, 19, 43, 59, 197

Note, 21, 44–46, 60, 76, 82, 118, 155, 177

partial function, see function
partial ordering, 22
processor-behavior analysis, 5

real-time system, 1–3, 8, 12
   hard, 2, 3
   soft, 2
reflexive relation, see relation
register, 43
relation, 22
   anti-symmetric, 22
   reflexive, 22
   transitive, 22

safe estimation, see estimation
shared memory, 3, 5, 7, 11, 17, 19, 43, 198
single-core CPU, 8, 19
static analysis, 5

TACLe, see COST Action
tight estimation, see estimation
top element, 22
total function, see function
transitive relation, see relation

UPPAAL, 8, 16
upper bound, see bounds

variable, 43