Data Dependency Analysis in Industrial Systems

Mälardalen University

School of Innovation, Design and Engineering

Azra Čaušević

DVA423 Thesis for the Degree of Master of Science (60 credits) in Computer Science with Specialization in Software Engineering, 15 credits

2015-10-28

Examiner: Jan Carlson

Supervisors: Luka Lednicki and Frank Lüders
Abstract

Software in modern industrial systems may have complex data dependencies. As a result of this, it can be hard for system developers to understand the system’s behavior, if they cannot explicitly see these dependencies. This thesis addresses this problem, with an emphasis on dependencies among data in systems built using the IEC 61499 standard. An analysis method was developed, with which we are able to extract data dependency information from basic and composite function blocks. The first step of handling this problem is to investigate how the data dependencies occur in IEC 61499. The second step is to create a formal definition of IEC 61499 elements that were needed in order to formulate the analysis method. Next, we define a dependency matrix, in which we store the information regarding dependencies between input and output data ports. Later, we formulate the necessary algorithms for data dependency analysis in basic and composite function blocks. Finally, the last piece of the puzzle is to develop a plug-in for Framework for Distributed Industrial Automation and Control – Integrated Development Environment. This plug-in is used to show that the analysis method is efficient and that the proposed analysis is applicable to IEC 61499 systems.
Contents

Abstract ......................................................................................................................... 1
List of Figures ............................................................................................................... 3
1. Introduction ............................................................................................................. 4
  1.1 Problem formulation ......................................................................................... 4
2. Background ............................................................................................................ 6
  2.1. The IEC 61131-3 standard ............................................................................ 6
  2.2. The IEC 61499 standard ................................................................................. 8
  2.3 IEC 61499 elements ......................................................................................... 8
    2.3.1 Basic Function Block .............................................................................. 9
    2.3.2 Service Interface Function Blocks .......................................................... 10
    2.3.3 Composite Function Block ......................................................................... 10
    2.3.4 Function Block Network ............................................................................ 10
  2.4. The 4DIAC initiative ...................................................................................... 11
  2.5. State Of The Art ............................................................................................. 11
3. Method .................................................................................................................... 12
4. Formal definition of IEC 61499 ........................................................................... 13
5. Results ..................................................................................................................... 15
  5.1 Data dependency analysis in the algorithm level ............................................ 15
    5.1.1 Data Dependency Type 1 ......................................................................... 15
    5.1.2 Data Dependency Type 2 ......................................................................... 16
    5.1.3 Data Dependency Type 5 ......................................................................... 16
    5.1.4 Data Dependency Type 6 ......................................................................... 17
  5.2 Data dependency analysis in the FB level ....................................................... 17
    5.2.1 Data Dependency type 3 .......................................................................... 17
    5.2.2 Data Dependency type 4 .......................................................................... 18
    5.2.3 Data Dependency type 7 .......................................................................... 18
    5.2.4 Data Dependency type 8 .......................................................................... 18
  5.3 Basic Function Block Dependency Analysis Method ....................................... 19
  5.4 Composite Function Block Dependency Analysis Method ........................... 25
  5.5 The 4DIAC-IDE plug-in .................................................................................... 27
6. Evaluation ................................................................................................................ 30
7. Related Research .................................................................................................... 36
8. Conclusion ............................................................................................................... 39
  8.1 Future work ....................................................................................................... 39
References .................................................................................................................... 40
Abbreviations .............................................................................................................. 42
List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>PL defined in IEC 61131-3 implementing a simple function</td>
<td>7</td>
</tr>
<tr>
<td>2.2</td>
<td>A Basic Function Block</td>
<td>9</td>
</tr>
<tr>
<td>2.3</td>
<td>A Function Block Network</td>
<td>10</td>
</tr>
<tr>
<td>3.1</td>
<td>Research Method</td>
<td>12</td>
</tr>
<tr>
<td>5.1</td>
<td>ECC of FB_FT_TN16 FB</td>
<td>17</td>
</tr>
<tr>
<td>5.2</td>
<td>(a) BFB interface (b) ECC BFB</td>
<td>19</td>
</tr>
<tr>
<td>5.3</td>
<td>Stages of the Dependency Matrix</td>
<td>21</td>
</tr>
<tr>
<td>5.4</td>
<td>Stages of the Dependency Matrix</td>
<td>21</td>
</tr>
<tr>
<td>5.5</td>
<td>CFB analysis example</td>
<td>25</td>
</tr>
<tr>
<td>5.6</td>
<td>Dependency matrices of BFBs F_1 and F_2</td>
<td>25</td>
</tr>
<tr>
<td>5.7</td>
<td>Stages of updating dependency matrix of CFB</td>
<td>26</td>
</tr>
<tr>
<td>5.8</td>
<td>Executing the plug-in</td>
<td>28</td>
</tr>
<tr>
<td>5.9</td>
<td>BFB AlignValues (a) interface (b) ECC (c) dependency matrix</td>
<td>28</td>
</tr>
<tr>
<td>5.10</td>
<td>Launching the 4DIAC-IDE</td>
<td>29</td>
</tr>
<tr>
<td>6.1</td>
<td>Interface of FB_FT_TN1 BFB</td>
<td>30</td>
</tr>
<tr>
<td>6.2</td>
<td>ECC of FB_FT_TN1 BFB</td>
<td>30</td>
</tr>
<tr>
<td>6.3</td>
<td>Dependency matrix of the BFB FB_FT_TN1</td>
<td>31</td>
</tr>
<tr>
<td>6.4</td>
<td>Dependency Matrix extended with internal variables</td>
<td>32</td>
</tr>
<tr>
<td>6.5</td>
<td>Interface of Int_To_LReal</td>
<td>32</td>
</tr>
<tr>
<td>6.6</td>
<td>Composite Network of Int_To_LREAL CFB</td>
<td>33</td>
</tr>
<tr>
<td>6.7</td>
<td>Results of Data Dependency Analysis in CFB INT_To_RLeal</td>
<td>33</td>
</tr>
<tr>
<td>6.8</td>
<td>Interface of LReal_To_Int CFB</td>
<td>34</td>
</tr>
<tr>
<td>6.9</td>
<td>Composite Network of CFB LReal_To_Int</td>
<td>34</td>
</tr>
<tr>
<td>6.10</td>
<td>Results of the analysis performed by the plug-in</td>
<td>35</td>
</tr>
</tbody>
</table>
1. Introduction
Over the last years manufacturers of distributed automation systems use programming languages that are defined in the IEC 61131-3 standard, in order to program the fundamental controller of the systems, namely programmable logic controllers (PLCs). The increased usage of controllers, in conjunction with the rapid improvement of the network performance, had as a result the increased complexity of the distributed applications [20]. In order to tackle this, IEC established the 61499 standard. The purpose of this standard is to deal with new requirements of the distributed applications, such as configurability interoperability and others [20]. Nowadays the standard is commonly used in order to define software components of automated systems [20].

The fundamental software unit defined in the 61999 standard is the function block [21]. According to the events that are triggered, the function block executes specific algorithms which leads to the production of new events and data. The data transfer is achieved through the data inputs and outputs that are defined on the interface of each function block [21].

Despite the fact that the research regarding the standard is quite extensive, a field that has not been investigated thoroughly is related to the dependency between data inputs and outputs of a function block.

This thesis will deal with this matter.

1.1 Problem formulation
The main goal of the thesis is to develop the theory of an analysis method, with which we will be able to investigate the dependencies between data in industrial software systems. Based on this theory we will be able to examine software data flow. This will allow us to study how program outputs depend on program inputs. In addition, the theory will provide us a way for exploring which inputs a selected output depends on. Furthermore, the formulated theory will illustrate the way we should follow in order to identify how many and which outputs depend on a specific input.

In this thesis we will consider systems that are built using the IEC 61499 standard.

Investigating the answers to the following research questions will provide us with the needed information in order to reach the above mentioned goal:

RQ1: In which ways can output data be dependent on input data based on the IEC 61499 standard?

RQ2: Which method can we use for calculating data dependency?

The second main goal of this thesis is to implement a plug-in for the 4DIAC-IDE tool. 4DIAC is an open and free framework that gives the opportunity to establish an automation and control environment based on the three main targets, namely portability, configurability and interoperability. The implemented prototype will be used in order to validate the theoretical analysis definition we formulated previously.
This thesis report is organized as follows. In Section 2 we provide the necessary theoretical information regarding used standard and framework that is needed in order to understand the contributions of the report. In Section 3 we describe the research method that was used to retrieve the results of the work, while in Section 4 we provide formal definition of IEC 61499 elements that are required to present the results of the work. Section 5 presents formulated data dependency analysis method and guidelines for developed plug-in usage. In Section 6 we evaluate the results of the research by executing formulated analysis on existing industrial systems. Finally in Section 7 we summarize the results of the research and we discuss future work.

Foundation for this thesis is a previous project, partly funded by ABB, where a new method for analyzing control dependencies in systems implemented in IEC 61499, was developed.
2. Background
This chapter describes fundamental theoretical principles, in order for the reader to understand this scientific report. Section 2.1 outlines the International Electrotechnical Committee (IEC) 61131-3 standard, while the following section 2.2 describes its successor namely the IEC 61499 standard. Next, Function Blocks (FBs) with a focus on data ports are illustrated in section 2.3. Finally, section 2.4 presents Framework for Distributed Industrial Automation and Control (4DIAC) for which the additional plug-in was created.

2.1. The IEC 61131-3 standard
Nowadays, the process of designing a distributed automation system is quite complicated. The reason is that the distributed automation system is composed of a wide variety of control nodes with multiple interactions. Programmable logic controllers are mainly used in order to implement the control nodes.

PLCs contain programmable memory in which all the necessary instructions and data are stored to enable PLCs to implement their tasks [8]. The instructions are written in a specific programming language and can be seen as a set of orders which guide the PLCs how to execute their tasks. Moreover, all the required functions are stored in the memory of PLCs. These functions are needed in order for the PLCs to perform logic, timing or arithmetic operations etc. [8]. One of the major advantages of the PLC is its reusability. It is possible to reuse the same PLC in different distributed automation systems which makes a PLC cost effective [8].

The main problem that was needed to be addressed in the beginning of the PLC development was that there wasn’t a globally defined programming language that could be used in order to develop PLCs [8]. Different manufacturers were developing PLCs by using their own programming languages. As a result of this, the significant advantage of PLC, reusability, tended to be diminished. It was not possible to reuse a PLC in more than one distributed automation systems because the programming language in which it was developed was different. As a consequence of this, PLC development became costly and time consuming [5]. Therefore, it became clear that a set of standardized programming languages that could be used for the development of PLCs was needed. Thus the 61131-3 standard was established by IEC.

This standard is an attempt to globalize the way the manufacturers program using PLCs by defining, among others, five programming languages [5]: Function Block Diagram (FBD), Ladder Diagram (LD) and Sequential Function Chart (SFC), which are graphical languages, Structured Text (ST) and Instruction List (IL) which are textual. The standard is mainly used as guidelines for developing the programs for PLCs but also includes all the necessary principles for implementing PLCs projects. We can separate the 61131-3 standard into two parts. The first part is called Common Elements and describes all the elements that are common in the programming languages which are defined in the second part [9].

In the Common Elements, parts the data typing and the variables are defined. With data typing we define the type of the parameters that will be used in the program [9]. This is considered to be useful because we can detect possible errors early in the development
process. The most commonly used data types are *Integer* and *Boolean* but depending on the program and the tasks of the PLC we develop, plenty of other data types can be used as well, such as Real, Date String and others [9]. In addition, we can define our own data types which are called derived data types.

Variables are used only in order to express hardware addresses, namely inputs and outputs [9]. Their scope is in principal limited to the program that is currently used but we can also define variables with a global scope. In the second we define the variable as **CAR_GLOBAL** [9].

In addition, in this part of the standard, the program organization unit (POU) is defined [5]. POU is the fundamental software of IEC 61131-3 standard. It has three different types, namely Function (FUN), Function Block and Program (PROG). On the other hand, the main difference between an FUN and an FB is that an FB has memory in which an algorithm and data are stored [9]. An FUN implements a defined function from the standard, such as SUB, SQRT and others. A PROG can be considered as a network which is composed of FUN and FBs [9].

In the second part of the 61131-3 standard the syntax and the semantics of the five programming languages are defined [9]. Figure 2.1 illustrates a simple function which performs the logical operation **AND** between two inputs. The first input is a variable *A* and the second input is the negated variable *B*. In the Figure 2.1 all four programming languages are implementing the same function.

![Diagram](image)

**Figure 2.1 – PL defined in IEC 61131-3 implementing a simple function [9]**

Over the recent years, distributed automation systems were enriched with other sources of information, such as sensors and actuators, except PLCs. Therefore, the way in which these systems behave is rather complicated, unpredictable and vulnerable to changes that are related with each of the components of the system [2]. These changes could be caused because of various different reasons such as differences in operating systems, network protocols that are used etc. Consequently, it is quite challenging for an engineer
to capture and express the overall behavior of a distributed automation system by using one design framework [2]. Therefore, IEC established the 61499 standard, which will be presented in the following section.

### 2.2 The IEC 61499 standard

The IEC 61499 standard is an industrial standard that came as a successor of the IEC 61131-3 standard. Using 61131-3 languages is not really the purpose of the 61499 standard. It is just a way to support development in a way similar to 61131[2]. The purpose is more to add the things that the 61131-3 standard did not support.

The basic objectives of the IEC 61499 standard is to cope easily with three major characteristics, namely portability, configurability and interoperability [1] [4]. According to Strasser [4] portability is the ability of software tools to accept and interpret correctly library elements produced by other software tools. In addition, configurability is the ability of devices and their software components to be configured (selected, assigned locations, interconnected and parameterized) by multiple software tools. Finally, interoperability is the ability of devices from different vendors operating together to perform the functions specified by one or more distributed applications [4].

This standard describes how we can use Function Blocks in order to develop industrial distributed control systems [10]. This is achieved by defining a component based model in which Function Blocks are logically connected with each other in order to construct the distributed control system [10, 11]. The components of the standard are presented in the following section.

### 2.3 IEC 61499 elements

The whole architecture of IEC 61499 is based on FBs. FBs have an interface which encapsulates programming details that implement its functionality [1]. FBs are used in order to express various computational units, which are independent from each other, and their connection [6]. When we need to connect more than one FB in an application, we use event and data connection arcs in order to connect them. Definition of FBs and their behavior in IEC 61499 is more explicit than in IEC 61131-3. According to [3], a FBs consists of two parts, namely the controlling part and the data part.

The controlling part specifies the behavior of the FB. Based on the current state of the FB and the input events, it produces output events, which are transferred to other FBs [6]. Therefore, event inputs and outputs are used in order to express the execution flow [1].

On the other hand, the data ports are responsible for the FBs computations. These computations are based on the input data that are used by the FB through the mapped input variables and are produced according to the algorithm of the FB [6]. In principle, this algorithm is invisible outside the FB [3]. The result of these computations is sent as output data through the output variables. Consequently, data inputs and outputs are used in order to transfer data [1].

According to [1, 3], FBs are divided into three categories: Basic Function Blocks (BFBs), Composite Function Blocks (CFBs) and Service Interface Function Blocks (SIFBs).
2.3.1 Basic Function Block

BFB are used in order to create a Function Block Network (FBN) [6]. The IEC 61499 standard specifies that the declaration of the BFB can be done either graphically or textually. In order for a BFB to be executed an event must be raised. This event is defined as one of the event inputs of the FBD [1].

A BFB applies an execution control chart (ECC), which can be seen as an event-driven finite state machine [6], which consists of states and guarded transitions. This is used in order to manage the order of the execution of its algorithms [3]. Each state may include one or more algorithms. The algorithms of each state are executed when we reach this state [1]. Each algorithm will use the input variables that are required in order to calculate and provide the output variables [6]. The transition from each state to another is done when the guard of the transition becomes TRUE. As a guard can be used either a Boolean variable, an event or statements that include the usage of data inputs and outputs.

It is also possible to associate input events with input data ports, as shown in Figure 2.2. This is achieved by using the WITH qualifier [1]. There are two types of association. The first type defines an association between one input event port with one or more data input ports, while the second one defines an association between one output event port with one or more data output ports [1]. The first type of association is used in order to retrieve information regarding the set of input data ports that will be read together with the input event port that are associated with [1]. The second type of association defines the set of output data ports that will be updated together with the output event port.

Figure 2.2 illustrates a Function Block with one input event called \textit{REQ} and two input variables on the left side, \textit{IN1} and \textit{IN2}. In addition, it produces one output event and one output variable, namely \textit{CNF} and \textit{OUT} respectively, which are shown in the right side and are sent to the following Function Blocks that are connected with it. In the center of the BFB we state its name \textit{F_ADD}. Finally, we also define the version of the BFB, which is in our example \textit{1.0}. The version is used in order for the engineers to recognize how many updates have been applied to the specific BFB.

![Figure 2.2 – A Basic Function Block](image_url)
2.3.2 Service Interface Function Blocks
According to Lednicki [1], Service interface function blocks are used as interfaces of external services. The IEC 61499 standard does not fully documents SIFBs but they may conclude a sequence diagram in order to express the behavior of the SIFB [1]. SIFBs provide same FB interface as other FB types. The main difference is that the execution of an SIFB can start independently to arrival of input events.

2.3.3 Composite Function Block
A CFB, as the name implies, is built by composing BFBs, SIFBs and other CFBs. Similarly to the BFBs, a CFB has a set of input variables and a set of output events. In addition, it provides a set of event outputs and a set of event variables. Furthermore, the variables and the events may be associated with each other [6] similarly to BFBs.

The Function Blocks that are used internally in the CFB are connected with each other in order to provide its functionality. Therefore, by using CFBs it is possible to represent the complex software functionality.

2.3.4 Function Block Network
A function block network consists of several FBs, CFBs and SIFBs. The elements of a FBN are connected with each other with the ports of these elements [1]. In principle, FBN is used in order to express the functionality of a control system. It can be also used in order describe the structure of a CFB. Figure 2.3 illustrates a simple FBN.

![Figure 2.3 – A Function Block Network](image)
2.4. The 4DIAC initiative

Over the recent years only few tools are created in order for an engineer to develop an application based on IEC 61499. One of the main reasons for this is that the standard does not clearly define how the FBs will be executed [4]. Some of the most well-known and widely used tools are: 4DIAC [13], Function Block Development Kit [10], nxtStudio [11] and IsaGRAF [12].

This thesis report will focus on the 4DIAC-IDE tool. The main purpose of this tool is to give an opportunity to the developers to use an open-source framework that supports completely the IEC 61499 standard and provides a big extent automation and control environment. It is developed by PROFACTOR GmbH and provides an editor for Eclipse that can be used in order to create FBs, CFBs and SIFBs[13]. The standard is based on Graphical Editing Framework, which is used in order to develop graphical user interfaces.

There are two main projects which are developed by 4DIAC initiative [12]. The first one is the 4DIAC-RTE/FORTE which is a runtime environment, implemented in C++, complied with IEC 61499 standard and is used for developing simple control automated systems [12]. It supports development of the basic elements described in the standard, namely FBs, CFBs and FBNs. The second project is the 4DIAC-IDE which can be seen as an extension of 4DIAC-RTE/FORTE since it is an extensive framework that is used for developing large distributed automation applications [12].

2.5. State Of The Art

Over the recent years, there has been an extensive research on several perspectives of the IEC 61499 standard. In [14] Thramboulidis discuss the vagueness regarding execution semantics of an ECC. The problem stems from the fact that the IEC 61499 standard does not clarify the order of the execution of the FBs in cases where there is more than one FBs ready to be executed. As result of this, manufacturers of distributed automated systems follow different approaches regarding the execution of the application. In [15] Čengić et al. showed how the same application may have different behavior depending on the platform that is executed. Sünder et al. [16] investigated this matter even further in terms of discussing how the event ports that are contained in a FBN are scheduled.

In addition, several researchers investigate how we can have a plain transition from PLC’s to the 61499 standard. Wegner et al. [18] suggested a transformation of a PLC application to an application that will be complied with 61499 standard, while Gerber et al. [19] suggested a manual migration. Furthermore in [2] Zoitl presents how we can use the programming languages defined in IEC 61131-3 standard in IEC 61499 FBs in order to achieve a certain level of harmonization between the two standards.
3. Method
The research method that was used during this thesis is illustrated in Figure 3.1. In the beginning the research questions related to the thesis were stated (problem formulation). Then the necessary background research was conducted. During this activity, we gathered all the information from previous topics that are related to this thesis. The purpose of the thesis is to investigate how we can analyze the dependency between input and output data ports. In order to do that, we developed and formulated the theory behind the analysis method. In addition, a research for gathering state of the art and related work was conducted. Based on this information an analysis method was formulated. This analysis method answered the questions that were previously formulated.

Afterwards, we analyzed and evaluated our results by implementing a plug-in for the 4DIAC-IDE tool which validated our theory. Finally, all the results of this research were discussed and presented in the thesis report.

Figure 3.1 - Research Method
4. Formal definition of IEC 61499

In order to continue this report with the description of data dependency analysis, first we must address needed definitions of the used IEC 61499 elements. This section presents these definitions. Parts of the definitions presented in this report are based on the work of Čengić and Åkesson [7] and Lednicki [1]. Definitions that vary from the ones defined in the mentioned paper and Ph.D. dissertation are formed based on the IEC 61499 standard definition [3]. Elements that are not significant for this work, such as event ports and connections between them, are disregarded.

**Definition 1.** *A function block interface* is defined as:

Function block interface:

\[ I = \langle D_i, D_o \rangle, \]

where

\[ D_i \] is a set \( \{d_{i0}, ..., d_{i|D_i|} \} \) of data inputs;

\[ D_o \] is a set \( \{d_{o0}, ..., d_{o|D_o|} \} \) of data outputs.

**Definition 2.** *A basic function block* and its subcomponents are defined as:

Basic function block:

\[ bFB = \langle I, ECC, IntVars \rangle, \] where

\[ I \] is a function block interface;

\[ ECC \] is an execution control chart (ECC);

\[ IntVars \] is a set \( \{IntVar_0, ..., IntVar_{|IntVars|} \} \) of internal variables.

Execution control chart (ECC):

\[ ECC = \langle S, T, A \rangle, \] where

\[ S \] is a set \( \{s_0, ..., s_{|S|} \} \) of ECC states;

\[ T \] is a set \( \{t_0, ..., t_{|T|} \} \) of ECC transitions.

\[ A \] is a set of algorithm functions.

ECC state:

\[ s = \langle sa_1, ..., sa_{|S|} \rangle, \] where

\[ sa_i \] is an ECC state action.

ECC state action:

\[ sa = \{a_0, ..., a_{|sa|} \}, \] where

\[ a_i \] is the algorithm to be executed, \( a_i \in A \);
ECC transition:
\[ t = (s_s, s_d, g), \]
where
- \( s_s \) is the source state, \( s_s \in S; \)
- \( s_d \) is the destination state, \( s_d \in S; \)
- \( g \) the guard of the transition.

Guard of transition:
\[ g = (GIntVars, GD_i), \]
where
- \( GIntVars \) a set of internal variables used in the guard, where \( GIntVars \subseteq IntVars; \)
- \( GD_i \) a set of data inputs used in the guard, where \( GD_i \subseteq D_i. \)

**Definition 3.** A **composite function block** and its elements are defined as:

Composite function block:
\[ cFB = (I, FBN), \]
where
- \( I \) is a function block interface.
- \( FBN \) is the internal function block network.

Function block network:
\[ FBN = (F, C), \]
where
- \( F \) is a set of function blocks, each of which is either \( bFB, siFB \) or \( cFB; \)
- \( C \) a set of connections between data ports, where \( C = \{c_0, ..., c_{|c|}\} \)

Connection:
\[ c = (d_s, d_d), \]
where
- \( d_s \) is the data port used as the source of the connection;
- \( d_d \) is the data port used as the connection target.
5. Results
Now that we have described the needed IEC 61499 elements that we will use in the analysis, we can present the actual analysis method. This chapter will outline data dependency analysis method on three levels. First one is the algorithm level, while second and third level present basic and composite function blocks analysis respectively. In addition to the detailed algorithms explanations, we will present a 4DIAC plug-in which implements the previously mentioned analysis method. This implementation is used to evaluate the analysis method.

To answer RQ1, on which ways can output data be dependent on input data based on the IEC 61499 standard, we distinguish eight different types of data dependency:

1. An output data depends on input data on the algorithm level.
2. An output data depends on internal variable on the algorithm level.
3. An output data depends on data input on FB level.
4. An output data depends on internal variable on FB level.
5. An internal variable depends on data input on the algorithm level.
6. An internal variable depends on internal variable on the algorithm level.
7. An internal variable depends on data input on FB level.
8. An internal variable depends on internal variable on FB level.

Dependency types 1 to 4 are considered to be direct data dependency, while dependency types 5 to 8 are considered to be indirect data dependency. In the following sections all these types of dependency will be thoroughly analyzed with the use of examples.

5.1 Data dependency analysis in the algorithm level
The lowest level of the IEC 61499 model hierarchy consists of algorithms that are implemented by code and not defined by models. Determining the data dependency based on this code is outside the scope of this thesis report. We assume that the analysis of the code is previously done and that the data dependency information that is retrieved by the algorithms is already available.

Data dependency types 1, 2, 5 and 6 fall into this category.

5.1.1 Data Dependency Type 1
In this case the result of the calculation for each data output is related to one or more data inputs. This means that in order for the algorithm to produce the value of the data output, it is necessary to know the values of data inputs. Consequently, the data output depends on these data inputs. The algorithm \textit{REQ}, shown in Listing 1, is part of a Function Block named \textit{F_BAND_B}, that has two data input ports \textit{X} and \textit{B} and one data output port \textit{OUT}. Having a closer look at the statements of this algorithm, we notice that the input is used to calculate the output (line 6). In addition, we notice that the input is used in control flow statements of the program, for example “if... else” blocks. In these statements, updates to an output are influenced by the value of input ports (lines 1 and 2).
1 IF \( X < B \) THEN
2 \hspace{1em} \text{OUT} := 0;
3 ELSIF \( X > 255-B \) THEN
4 \hspace{1em} \text{OUT} := 255;
5 ELSE
6 \hspace{1em} \text{OUT} := X;
7 END IF;

Listing 1 – REQ Algorithm

5.1.2 Data Dependency Type 2
In this case the result of the calculation of the data output is related to one or more internal variables that are used in the algorithm. This is visible in listing 2, which presents algorithm \( \text{REQ} \) of a basic function block named \( \text{FB_INTEGRATE} \) version 0.0 that has three data input ports \( X, K \) and \( \text{reset} \) and one data output port, \( \text{Y_OUT} \). This BFB has additional four internal variables \( X_{\text{last}}, \text{last}, \text{tx} \) and \( \text{Y_intern} \). There are two states in the ECC for this BFB, named \( \text{INIT} \) and \( \text{REQ} \). For the purpose of this example we will investigate the algorithm of the state \( \text{REQ} \).

As we can see, in line 9 in order to calculate the data output \( \text{Y_OUT} \) we need to know the value of the internal variable \( \text{Y_intern} \). This is a typical example of a data dependency where an output data depends on an internal variable.

5.1.3 Data Dependency Type 5
In this type of data dependency an internal variable depends on a data input. Then the internal variable is used to calculate a data output. Therefore, the data output depends on the internal variable. In addition, the internal variable depends on the data input. As a result of this, we conclude that the data output depends on the data input. This is visible in the example of the previous section. Internal variable \( \text{Y_intern} \) depends on data input \( X \), since, in order to calculate it, we need to know the value of \( X \) (line 5). Furthermore, data output \( \text{Y_OUT} \) depends on internal variable \( \text{Y_intern} \) (line 9). As a result of these dependencies, we conclude that there is an indirect dependency between data output \( \text{Y_OUT} \) and data input \( X \).
5.1.4 Data Dependency Type 6
In this type of data dependency, an internal variable depends on another internal variable. Then the first internal variable is used in order to calculate a data output and because of the data dependency between the two internal variables, we conclude that the data output depends in the second internal variable as well. This is clearly visible in the example of algorithm we presented in section 5.1.2. In line 5 we notice that internal variable \( Y_{\text{intern}} \) depends on internal variables \( X_{\text{last}}, \text{last} \) and \( tx \). In addition, internal variable \( Y_{\text{intern}} \) is used in order to calculate the data output \( Y_{\text{OUT}} \). Thus, data output \( Y_{\text{OUT}} \) depends also on the internal variables \( X_{\text{last}}, \text{last} \) and \( tx \).

As we mentioned in the beginning of section 5.1, we will use as an input the data dependency analysis on the algorithm level in our analysis methods, in order to conclude for data dependencies between data inputs and data outputs in the BFB and CFB level.

5.2 Data dependency analysis in the FB level
In this section we will analyze data dependency types 3, 4, 7 and 8. These types of dependency occur in the FB level, which means we will have to analyze the ECC of the Function Block in order to conclude for data dependencies.

5.2.1 Data Dependency type 3
According to data dependency type 3, a data output depends on a data input on the FB level. This occurs when one or more data inputs are used in the guards of the transitions between the various states of the ECC. As a result of this, the data outputs that are used in the destination states of the transitions, where the data inputs are used, depend on these data inputs. This is visible in the following example.

![Figure 5.1 – ECC of FB_FT_TN16 FB](image)

Figure 5.1 represents the ECC of the \texttt{FB_FT_TN16} Function Block. This function block has two data inputs, namely \texttt{IN} and \texttt{T} and one data output \texttt{OUT}. In addition, it uses the internal variables \texttt{length, X, cnt, last} and \texttt{tx}. As we can see from the figure in the
transition from checkTime state to REQ state we use as a guard a comparison condition. If this condition holds, then we can go to the destination state (REQ). In this condition we use the data input $T$. As a result of this, all the data outputs that are used in the algorithm of the destination state will be dependent on the data input $T$.

5.2.2 Data Dependency type 4
Based on this type of data dependency, a data output depends on one or more internal variables. In order for this to happen, these internal variables should be used in the guards of the transitions. Then we conclude that all the data outputs that are used in the destinations state of the transition will be dependent on these internal variables. This is clearly visible in the example that we mentioned in the previous section. The guard of the transition from checkTime state to REQ state uses the internal variables $x$, last and length. Therefore, we conclude that all the data outputs that are used in the algorithm of the destination state (REQ) will be dependent on these internal variables.

5.2.3 Data Dependency type 7
This type of dependency is related to the internal variables that are used in algorithms when they depend on data inputs, which are used in the guards of the transitions between the states. In order for this to happen, one or more data inputs need to be used in the guard of a transition, and the algorithm(s) of the destination state need to use one or more internal variables. Then one or more data outputs depend on these internal variables. Therefore, we conclude that the data outputs of the destination state are dependent on the data inputs that were used in the guard. The following listing presents the code of the algorithm REQ of the previous example.

```plaintext
1   IF cnt = length - 1THEN
2       cnt := 0;
3   ELSE
4       cnt := cnt + 1;
5   END_IF
6   OUT := X[cnt];
7   X[cnt] := IN;
8   last := tx;
```

Listing 3 – REQ Algorithm

In this listing we notice that the data output OUT depends on the internal variable cnt (line 6). In addition the internal variable cnt depends on the data input $T$ that is used in the guard of the transition. Consequently, we say that the data output OUT depends on the internal variable cnt.

5.2.4 Data Dependency type 8
This last type of data dependency is related to internal variables that are used in an algorithm of a state and internal variables that are used in the guard of the transition, which reaches the state where the algorithm belongs to. In addition, we assume that a data output depends on one internal variable that is used in the algorithm. If this internal variable depends on the internal variables of the guards then we say that the data output depends on the internal variables of the guards as well. This is visible in the
example that we presented in the section 5.2.1 and 5.2.2. There, the data output \textit{OUT} depends on the internal variable \textit{cnt}. But the internal variable \textit{cnt} depends on the internal variables \textit{tx}, \textit{last} and \textit{length} that are used in the transition of the guard. Therefore we conclude that the data output \textit{OUT} depends on the internal variables \textit{tx}, \textit{last} and \textit{length}.

In order to answer RQ2 on which method can we use for calculating data dependency, two analysis methods were formulated. The first one investigates dependency on BFBs and is presented in the Section 5.2.1. Respectively, Section 5.2.2 presents CFBs dependency analysis method.

### 5.3 Basic Function Block Dependency Analysis Method

The Basic Function Block Dependency Analysis Method is used in order to determine the data dependencies between data inputs and data outputs. It consists of 4 algorithms, namely \texttt{do\_Analysis}, \texttt{UpdateMatrix}, \texttt{coverIndirectDependency\_1} and \texttt{coverIndirectDependency\_2}. We start the basic function block analysis by reading all data input and output ports from the interface. To explicitly show dependencies between input and output data ports for IEC 61499 function blocks, we introduced a dependency matrix. The matrix consists of rows representing data inputs and columns representing data outputs. A cell represents an ordered pair of data input and data output. If there is a dependency for that ordered pair, then their cell is set to a value 1, otherwise 0. Afterwards, we go through each algorithm of a state, where we read the existing information regarding which data output port is dependent on which data input port. This is achieved with the \texttt{UpdateMatrix} algorithm. Once the dependency among a pair of data output and data input is found, their cell value in dependency matrix is changed from 0 to 1. The \texttt{coverIndirectDependency\_1} and \texttt{coverIndirectDependency\_2} are two algorithms that are called inside the \texttt{UpdateMatrix} algorithm and are used in order to cover the several types of data dependencies that we defined by answering the RQ1. We will present an example that will be used in order to explain how the algorithms work.

In Figure 5.2 (a) we present the interface of the BFB. This FB has three data inputs, namely \textit{a}, \textit{b} and \textit{c}, and two data outputs, namely \textit{x} and \textit{z}. In addition the FB has three internal variables \textit{h}, \textit{i} and \textit{g}. In Figure 5.2 (b) we show the ECC of this FB. We notice that this ECC has two states \textit{S1} and \textit{S2} with one algorithm in each state \textit{a1} and \textit{a2} respectively. The code of each algorithm is presented in listing 4 and 5.
We start the analysis method by iterating over the transitions of the ECC. Therefore we start with the first transition from state \textit{Start} to state \textit{s1}. This transition has no guard, and thus we have no data inputs or internal variables to take into account for the destination state. Once we found the destination state we iterate over the algorithms of it. In this case we have only one algorithm called \textit{a1}. We retrieve the information regarding data dependencies between data inputs and outputs in the algorithm level. For algorithm \textit{a1} we notice that there is a dependency between data output \textit{x} and data input \textit{a}. Thus, in the dependency Matrix we update the respective cell from 0 to 1. In addition, we update the cells of the dependency Matrix that indicate dependency between internal variables.

It is important to mention at this point that the internal variables are included in the dependency matrix during the BFB analysis. Thus, we add next to the data inputs and the data outputs the internal variables of the algorithm. This is done in order to save also dependencies between data outputs and internal variables. The result of the BFB analysis though will not contain the internal variables, since it should be on the level of BFB interface and contain only data inputs and data outputs that are visible on the interface level. Thus we hide the complexities of the internal BFB structures.

In addition we check if there is a dependency between internal variables and data outputs. In algorithm \textit{a1} we notice that there is a dependency between data output \textit{x} and internal variable \textit{i}. Thus we will update the respective cell in the dependency matrix from 0 to 1. The next step in the analysis method is to transform data dependencies type 6 to 2. This means that we will check for each data output if it depends on internal variable(s) that depend(s) on other internal variable(s). We notice that internal variable \textit{i} depends on internal variable \textit{g}. Thus we conclude that data output \textit{x} depends on internal variables \textit{g} as well. At this point the dependency Matrix is as shown in step 1 in figure 5.3.

\begin{verbatim}
1 x := a + 1;
2 x := i;
3 c := 10;
4 h := 0;
5 i := b + 1;
6 i := g + 10;
\end{verbatim}

\textbf{Listing 4 – a1 Algorithm}

\begin{verbatim}
1 z := 3;
\end{verbatim}

\textbf{Listing 5 – a2 Algorithm}
The next step in the algorithm analysis is to transform data dependency type 5 to type 1. This means that we have data outputs that depend on internal variables, which depend on data inputs. In our algorithm we notice that data output \(x\) depends on internal variable \(i\) which depends on data input \(b\). Therefore there is a dependency between data output \(x\) and data input \(b\). This is shown in step 2 from figure 5.4. Once we completed the dependency analysis for algorithm \(a1\) we go to the next transition. This transition uses in its guard data input \(c\) and internal variable \(h\). Therefore the data outputs, which are used in the algorithms of the destination state, will be dependent on this data input and internal variable. In our case there is only one algorithm that will be executed, that is \(s2\). This algorithm contains only one data output, namely \(z\). Therefore this data output will be dependent on data input \(x\) and internal variable \(h\), as shown in step 3 of figure 5.4 below.

**Figure 5.3 – Stages of the Dependency Matrix**

<table>
<thead>
<tr>
<th></th>
<th>x</th>
<th>z</th>
<th>h</th>
<th>i</th>
<th>g</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>b</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>c</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>h</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>i</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>g</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Figure 5.4 – Stages of the Dependency Matrix**

<table>
<thead>
<tr>
<th></th>
<th>x</th>
<th>z</th>
<th>h</th>
<th>i</th>
<th>g</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>b</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>c</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>h</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>i</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>g</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**doAnalysis Algorithm**

Algorithm 1 is used in order to iterate over all the transitions of the ECC. In line 1 we define the basic function block as a set of data inputs and data outputs, that are retrieved by the interface of the function block, an ECC, a set of transitions of the ECC and a set of Algorithms in the ECC. In line 3 we iterate over the transitions of the ECC. For each transition we define a set of internal variables and a set of data inputs that are used in the guard of the transition (lines 4 and 5). Then, for each of the algorithms of each of the state actions of the destination state we call the algorithm 2 named updateMatrix.
**Algorithm 1** doAnalysis()

1: \( \langle D_i, D_o, (S, T, A), IntVars \rangle \leftarrow bFB \)
2: Let \( MbFB \) be the zero – filled dependency matrix of \( bFB \)
3: for each \( t \in T \) do
   4:   Let \( guard\_vars \) be a set of internal variables used in the guard of \( t \)
   5:   Let \( guard\_inputs \) be set of data inputs used in the guard of \( t \)
   6:   Let \( dest\_state \) be the destination state of \( t \)
   7:   for each \( sa \in dest\_state \) do
      8:     for each \( a \in sa \) do
         9:       updateMatrix\( (a, guard\_vars, guard\_inputs) \)
      10:   end for each
   11: end for each
12: end for each

**updateMatrix Algorithm**

This algorithm is used in order to update the Matrix that represents the dependency between data inputs and data outputs. In lines 1 and 2 of the algorithm we define a set of data outputs and internal variables that are used in the algorithm \( \alpha \). In lines 3 to 10 we iterate over the internal variables of the algorithm. In lines 4 and 5 we retrieve the information for dependency between the internal variable and the data inputs and other internal variables that are used in the algorithm. Based on this information, we update the matrix in case there is a dependency between the internal variable and the data input or other internal variable.

In lines 11 to 22 we iterate over the data outputs of the algorithm \( \alpha \). In this loop we will update the dependency Matrix when we detect a dependency between data input and data output, or internal variable and data output, or an internal variable or data input used in the guard and the data output. In lines 12 and 13 we retrieve the information for dependency between the data output and the data inputs and internal variables that are used in the algorithm. Based on this information in the two loops of lines 14-15 and 16-17 we update the dependency matrix. In lines 18 to 21 we update the matrix in order to save the dependency between the data output and the data inputs and internal variables that are used in the guard. Once we iterate over this procedure for every data output of the algorithm then we iterate over a loop by calling the algorithm coverIndirectDepedency_1. This is done in order to transform data dependencies types 6 to data dependency type 2. When this is done, we call the algorithm coverIndirectDepedency_2 in order to transform data dependencies types 2 and 5 to data dependency type 1.

**Algorithm 2** UpdateMatrix\( (a, guard\_vars, guard\_inputs) \)

1: Let \( data\_outputs \) be a set of data outputs of \( \alpha \)
2: Let \( internal\_variables \) be a set of internal variables of \( \alpha \)
3: for each \( internal\_variable \) in \( internal\_variables \) do
   4:   Let \( dep\_data\_inputs \) be a subset of data inputs of \( \alpha \) that \( internal\_variable \) depends on
   5:   Let \( dep\_internal\_variables \) be a set of internal variables of \( \alpha \) that \( internal\_variable \) depends on
   6:   for each \( int\_var \) in \( dep\_internal\_variables \) do
      7:      \( MbFB [internal\_variable, int\_var] = 1 \)
for each data_input in dep_data_inputs do
    $MbFB[\text{internal\_variable}, \text{data\_input}] = 1$
end for each

for each data_output in data_outputs do
    Let $\text{dep\_d\_inputs}$ be a subset of data inputs of $\alpha$ that $\text{data\_output}$ depends on
    Let $\text{dep\_int\_vars}$ be a set of internal variables of $\alpha$ that $\text{data\_output}$ depends on
    for each internal\_variable in dep\_int\_vars do
        $MbFB[\text{data\_output}, \text{internal\_variable}] = 1$
    end for each
    for each data_input in dep\_d\_inputs do
        $MbFB[\text{data\_output}, \text{data\_input}] = 1$
    end for each
    for each guard\_var in guard\_vars do
        $MbFB[\text{data\_output}, \text{guard\_var}] = 1$
    end for each
    for each guard\_input in guard\_inputs do
        $MbFB[\text{data\_output}, \text{guard\_input}] = 1$
    end for each
end for each

coverIndirectDependency\_1 Algorithm
As we mentioned before, Algorithm 3 is used in order to transform data dependency type 6 to data dependency type 2. We iterate the procedure of the algorithm for each of the data outputs of the Function Block (lines 2 to 14). For each of the data outputs we check if there is dependency between this data output and all the internal variables (line 4). Then for this internal variable, we check in line 6, whether there is a data dependency with another internal variable, which is actually the data dependency type 6. If this condition holds then we say that there is a data dependency between the data output and the internal variable. Therefore we transformed data dependency type 6 to data dependency type 2.

Algorithm 3 coverIndirectDependency\_1()
1: checker = 0;
2: for each data_output in the MbFB do
3:     for each internal\_variable in the MbFB do
4:         if $MbFB[\text{data\_output}, \text{internal\_variable}] == 1$
5:             for each IntVar in the MbFB do
6:                 if $MbFB[\text{internal\_variable}, \text{IntVar}] == 1$
7:                     if $MbFB[\text{data\_output}, \text{IntVar}] == 0$
8:                         $MbFB[\text{data\_output}, \text{IntVar}] = 1$
9:                         checker = 1;
10:                    end if
11:             end if
12:         end for each
13:     end if
14: end for each
15: return checker
The variable checker is used in order to iterate over the loop of line 5 as many times as it is needed so that we can cover the case where there is a dependency between more than two internal variables.

**CoverIndirectDependency_2 Algorithm**
The next step is to call algorithm 4. In terms of structure, this algorithm is similar to the previous one. The purpose of it is to transform data dependency type 5 to type 1. This is achieved by iterating over every data output in lines 2 to 12. For each of the data outputs and for each of the internal variables we check if there is data dependency between them (line 4). If yes, then we check if there is any data dependency between this internal variable and the data inputs. This is achieved in line 6. If there is data dependency, then we update the matrix so that we define a data dependency between the data input and the data output (line 7). Thus, we transformed data dependency type 5 to data dependency type 1.

**Algorithm 4** coverIndirectDependency_2()

2: for each data_output in the MbFB do
3:    for each internal_variable in the MbFB do
4:       if MbFB [data_output, internal_variable] == 1
5:       for each data_input in the MbFB do
6:          if MbFB [internal_variable, data_input] == 1
7:             MbFB [data_output, data_input] = 1
9:       end if
10:   end for each
11: end if
12: end for each
13: end for each

The analysis method that we defined in this section covers the data dependencies types 1 to 6. A limitation of the suggested solution, which would be interested to investigate even further, is to extend it in order to cover data dependencies types 7 and 8. That is, data dependencies that derive from the fact that an internal variable is dependent on a data input or on another internal variable that are used in the guards of the transitions of the ECC. Another aspect of the proposed analysis method that we should mention is that it does not cover the case where the ECC is cyclic.

In our solution we consider only the direct destination state to be dependent on data inputs and internal variables used in the guards of the transitions. But there is a case, where from a transition $T1$ with a data input in the guard, we go to state $S1$ and from state $S1$, we go to state $S2$ without using a guard in the transition. This means that the data outputs of state $S2$ are also dependent on the data input that is used in the transition $T1$. This case can be generalized when the ECC is cyclic. Therefore the suggested analysis method takes into account only direct dependencies from data inputs and internal variables used in the transitions to the data outputs used in the algorithms of the destination state.
5.4 Composite Function Block Dependency Analysis Method

For the dependency analysis method of a CFB we have created two algorithms. First algorithm starts the CFB analysis using the information defined on the CFB interface level, while the second one performs recursive network analysis. The reason that we use two algorithms is that we need to read only once the interface of the CFB in order to find data input ports. After reading the interface, the dependency matrix is initialized and all its values are set to 0. The function `doDependencyAnalysis` is called for each data input port. This algorithm goes through each and every connection of the data input port to find the destination data input port. If the destination data input port is one of the data output ports of the CFB then the dependency matrix is updated. If this is not the case, for each data output which depends on the destination data input port we call recursively `doDependencyAnalysis` algorithm.

![Diagram of CFB analysis example](image)

**Figure 5.5** – CFB analysis example

Figure 5.5 captures a composite function block that we will use in order to explain data dependency analysis on them. This CFB has two data inputs `a` and `b`, and two data outputs `x` and `z`. In addition, this CFB consists of two BFBs. First BFB, named `F_1`, has one data input `c` and one data output `f`. Second BFB, named `F_2`, contains two data inputs `d` and `e`, and two data outputs `g` and `h`.

In order to find dependencies in CFB we start by searching all the connections for each data input. For this example, we begin with the data input `a` and we find that there is one connection `c0` to BFB `F_1`, and its data input `c`. When visiting BFBs that are inside of a CFB we retrieve also their dependency matrix.

<table>
<thead>
<tr>
<th></th>
<th>c</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>d</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>g</th>
<th>h</th>
</tr>
</thead>
<tbody>
<tr>
<td>d</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>e</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

**Figure 5.6** – Dependency matrices of BFBs `F_1` and `F_2"
With this information we know that there is a data dependency between data output $f$ and data input $c$ of $F_1$. Next, we search for connections of a data port $f$. This is done by calling recursively doDependencyAnalysis function. Next connection, named $c2$ leads us to the BFB $F_2$ and its data input port $d$. Dependency matrix of $F_2$ tells us that data output port $h$ is dependent on data input port $d$. Now we follow the connection $c4$ from the data output port $d$ to data output port of a CFB $z$. Here we have found the first data dependency on the level of CFB. The cell in dependency matrix that corresponds to data output $z$ and data input $a$ is changed from 0 to 1. Since we have come to the end of connections for the data output $z$, we come back to the beginning and start looking into data input port $b$. We are following the same procedure which leads us to BFB $F_2$ and data input $e$, by going through connection $c1$. Dependency matrix of BFB $F_2$ tell us that within it there is a dependency between data output port $g$ and data input port $e$. Next we search for connections deriving from data output $g$. This connection, called $c3$, connects CFB data output port $x$ with BFB $F_2$ data output port $g$. Finally, we have found the second data dependency for this CFB. This dependency is between data output port $x$ and data input port $b$. Having in mind that we have visited all connections all that is left to do is to update the dependency matrix.

\[
\begin{array}{ccc}
\begin{array}{ccc}
x & z & 0 \\
 a & 0 & 0
\end{array}
\end{array}
\rightarrow
\begin{array}{ccc}
\begin{array}{ccc}
x & z & 0 \\
b & 0 & 0
\end{array}
\end{array}
\rightarrow
\begin{array}{ccc}
\begin{array}{ccc}
x & z & 0 \\
 a & 0 & 1
\end{array}
\end{array}
\rightarrow
\begin{array}{ccc}
\begin{array}{ccc}
x & z & 1 \\
b & 0 & 0
\end{array}
\end{array}
\rightarrow
\begin{array}{ccc}
\begin{array}{ccc}
x & z & 1 \\
b & 1 & 0
\end{array}
\end{array}
\]

**Figure 5.7 – Stages of updating dependency matrix of CFB**

Figure 5.7 presents all the stages that dependency matrix is going through. The dependency matrix is updated every time we reach a data output of the CFB.

**The CFB Dependency Analysis Algorithm**

Algorithm 5 defines the dependency analysis of a composite function block. The *for* loop starting on the line 3 is used to iterate over all data input ports. For each of the data input ports we will call the function doDependencyAnalysis. The doDependencyAnalysis function performs a recursive analysis starting from the port, and updates the dependency matrix. The updated dependency matrix of a composite function block is returned on the line 6.

**Algorithm 5** cFBDependencyAnalysis(cFB)

1: \(⟨(D_i, D_o), F, C⟩ \leftarrow cFB\)
2: Let $MbFB$ be the zero filled dependency matrix of cFB
3: **for** each $d_i \in D_i$ **do**
4: \( \text{doDependencyAnalysis} (cFB, McFB, d_i, d_i) \)
5: **end for**
6: return $McFB$
The doDependencyAnalysis Algorithm
doDependencyAnalysis function is presented in the Algorithm 6. The for loop starting on the line 2 is used to iterate over all connections starting from a specific data port. On the line 4, we test if the destination data input port is one of the data output ports of the CFB. In case it is, we update the dependency matrix of a CFB by setting 1, on both the row and column that represent data input and data output port respectively. If this is not the case, loop shown on line 9 is used to iterate over the data output ports, which are dependent on the data input port, by calling the function doDependencyAnalysis recursively.

Algorithm 6 doDependencyAnalysis (cFB, McFB, di, current_port)
1: Let Cons be the set of connections leading out of current_port
2: for each c ∈ Cons do
3: let d_dest be the destination data port of c
4: if (d_dest ∈ Do) then
5: McFB[di, d_dest] = 1
6: else
7: let MFB be the dependency matrix of FB that d_dest belongs to
8: let D_FB_o be the set of data outputs of the bFB
9: for each data_output_FB in D_FB_o
10: if M_FB_FB [d_dest, data_output_FB] == 1
11: current_port = data_output_FB
12: doDependencyAnalysis (cFB, McFB, di, current_port)
13: end if
14: end for
15: end if
16: end for

5.5 The 4DIAC-IDE plug-in
In conjunction with the formulation of the algorithms that can be followed in order to investigate the dependency between data output and data inputs of FBs we implemented a prototype tool. In this section we will give an overview of the tool and we will provide the necessary instructions in order to be installed and used.

The prototype tool that we developed implements all the algorithms that are defined in subchapter 5.1. The tool is built as a plug-in for 4DIAC-IDE.

The functionality of the plug-in is accessible through a context menu which appears when right-clicking a BFB or CFB. The plug-in is called DataDependencyAnalysis and by selecting Analysis we start its execution, as shown in Figure 5.8.
The results of the Analysis are shown in a popup window that is generated when we click on "Start Analysis". This popup window illustrates the dependency matrix that we defined in the algorithms of section 5.1. Each row is a data input and each column is a data output. Cells that are evaluated to 1 represent a dependency between data input and data output port. An example of the popup window is shown in Figure 5.9 (c).

In order to read the data dependency between data inputs and data outputs or/and internal variables, we parse the comment field of each algorithm. In this textfield we set a string which contains all the required information regarding dependencies. We defined this string in the form of: “data_output1=data_input1,int_var2 int_var1=int_var2,data_input2 – data_output1”. The part of the string to the right of “-“ contains all the data outputs that are used in the current algorithm. The part of the string to the left of the “-“, describes the types of dependencies that occur in this algorithm. These types of dependencies can be any of the types of dependencies that we described in section 5.1. In the above example we define that there is direct dependency between data output 1 and data input 1 and internal variable 2. In addition, we define that there is a dependency between internal variable 1 and internal variable 2 and data input 2. Finally we can also retrieve the information that in the current algorithm only data output 1 is used.
In order to install the plug-in, it has to be download as an Eclipse plug-in and imported as a new project to the existing framework of Eclipse. The preferred edition of Eclipse is the Eclipse Modeling Tools 4.4.2. Furthermore Plug-in Development for Eclipse should be installed. In addition, 4DIAC-IDE has to be downloaded in order to build and run it and then use the plug-in. Once we import the plug-in we are able to use it. In order to achieve this we need to launch an Eclipse application as illustrated in Figure 5.10.

**Figure 5.10 – Launching the 4DIAC-IDE**
6. Evaluation
We used the implemented plug-in in order to validate the algorithms that we defined in section 5.1. For the purpose of evaluation of our results we used existing systems that are provided by 4DIAC-IDE.

First we will introduce the results of the analysis of BFB as it was performed by the plug-in. The BFB that we selected is representative of the different cases of data dependency that may occur. The name of the BFB is **FB_FT_TN16** and it is a part of an existing system called **Boiler**. The interface of this BFB is presented in Figure 6.1. As we can notice this BFB contains two data inputs, namely **IN** and **T**. In addition, it contains one data output, namely **OUT**. The ECC of **PedCrossingCtl** is shown in Figure 6.2. In addition, the algorithms that executed during the travelling of the ECC of this BFB uses 5 internal variables, namely **length**, **X**, **cnt**, **last** and **tx**.

![Figure 6.1 – Interface of FB_FT_TN1 BFB](image)

![Figure 6.2 – ECC of FB_FT_TN1 BFB](image)
Furthermore, the states of the ECC contain in total three algorithms, namely \textit{INIT}, \textit{checkTime} and \textit{REQ}. By parsing the comment field of algorithm \textit{INIT} we retrieve the information that internal variable \textit{X} depends on data input \textit{IN}. When we parse the comment field of algorithm \textit{checkTime} we retrieve the information that there is no dependency within the algorithm. Finally, for algorithm \textit{REQ} we retrieve the following information regarding dependencies:

1. data output \textit{OUT} depends on internal variable \textit{X} and on internal variable \textit{cnt}.
2. internal variable \textit{cnt} depends on internal variable \textit{length}.
3. internal variable \textit{X} depends on data input \textit{IN}.
4. internal variable \textit{last} depends on internal variable \textit{tx}.

As we can notice the types of dependencies that occur in this BFB are types 2, 5, 6. Moreover, we see that there is a guard in the ECC which uses the internal variables \textit{X}, \textit{last} and \textit{length} and the data input \textit{T}. As a result of this, all the data outputs that are used in the destination algorithm(s) will be dependent on these internal variables and data input. In this case, the destination algorithm is \textit{REQ} and the data output that will be dependent is \textit{OUT}. Therefore, we notice that also data dependencies types 3 and 4 occur since:

1. data output \textit{OUT} depends on data input \textit{T}
2. data output \textit{OUT} depends on internal variable \textit{tx, last} and \textit{length}.

We performed data dependency analysis by using the plug-in. The results of the analysis are illustrated in Figure 6.3.

![Dependency matrix of the FB_FT_TN1 BFB](image)

**Figure 6.3 – Dependency matrix of the FB_FT_TN1 BFB**

As we mentioned in section 5 the final result of the dependency Matrix will present only dependencies between data inputs and data outputs. In figure 6.3 we notice that based on the result of the plug-in, data output \textit{OUT} depends on data inputs \textit{IN} and \textit{T}. The actual matrix, extended with internal variables after the data inputs and data outputs is shown in Figure 6.4.
As we can see from the data dependency matrix shown in Figure 6.4, the plug-in detected all the dependencies that occur in the BFB.

Furthermore, we evaluated our proposed analysis method for identifying data dependency in CFBs. In order to achieve this, we selected two representative existing systems that are provided by 4DIAC-IDE. For the purpose of this evaluation we simplified these examples in order to be understandable how our algorithm works.

The first system is called **Int_To_LReal**. Figure 6.5 represents its Interface. This system contains five data inputs, namely \( x, minInt, maxInt, minReal, maxReal \) and one data output namely \( Out \).
This CFB is composed of three BFBs. As illustrate in Figure 6.6 there are two instances of the BFB \textit{FB\_SUB\_INT} and one instance of the BFB \textit{FB\_ADD\_INT}.

As we can notice from the composite network of the CFB there is a dependency between data inputs $x$, \textit{minInt}, \textit{maxInt} and the data output \textit{Out}. We performed data dependency analysis on the above mentioned CFB with our plug-in. The results of the analysis are shown in Figure 6.7.

![Composite Network of Int\_To\_LREAL CFB](image)

**Figure 6.6 – Composite Network of Int\_To\_LREAL CFB**

As we can notice from the composite network of the CFB there is a dependency between data inputs $x$, \textit{minInt}, \textit{maxInt} and the data output \textit{Out}. We performed data dependency analysis on the above mentioned CFB with our plug-in. The results of the analysis are shown in Figure 6.7.

![Results of Data Dependency Analysis in CFB INT\_To\_RLeal](image)

**Figure 6.7 – Results of Data Dependency Analysis in CFB INT\_To\_RLeal**

We can notice that the plug-in identified all the dependencies between data inputs and outputs since the corresponding cells in the Data Dependency Matrix are evaluated to 1.

Afterwards we performed data dependency analysis on a second CFB, which is called \textit{LReal\_To\_Int}. This CFB contains five data inputs, namely $x$, \textit{minInt}, \textit{minReal}, \textit{maxReal} and \textit{maxInt} and one data output, namely \textit{Out}. The interface of the CFB is illustrated in Figure 6.8.
This CFB consists of two BFBs and one CFB, namely *FB_ADD_INT*, *FB_SUB_INT* and *Int_To_LReal* respectively. The way that the BFBs and the CFB are connected with each other is illustrated in the Composite Network of the system as shown in Figure 6.9.

As it is shown by the Composite Network data output *Out* depends on the data inputs *minInt* and *maxInt*. This is also certified by our plug-in. The results of the performed analysis are captured in Figure 6.10.
Based on the results that were produced by the prototype tool we developed we can conclude that the analysis methods defined in section 4 are valid. The data dependency are calculated correctly. Furthermore, the evaluation showed that we can apply our analysis method to real-world examples since the systems that were selected for evaluations were simplified but real.

Figure 6.10 – Results of the analysis performed by the plug-in
7. Related Research

Even though the research regarding the IEC 61499 standard is quite extensive, it is a fact that regarding the dependency between data inputs and outputs is limited. Other types of model level analysis of IEC 61499 have been proposed.

Lednicki in [1] proposed a technique that can be followed in order to calculate the worst case execution time (WCET) of applications that are complied with the IEC 61499 standard. In order to achieve that, he defined four different algorithms, that can be used to calculate the WCET of BFBs, CFBs and FBNs. The main logic behind these four algorithms is that we need to investigate the several execution paths a FB may have been based on the event inputs. The analysis is performed on a scale from BFBs to an actual application. In order to calculate the WECT of a CFB, we need to analyze and calculate the WCET of each of the BFBs that is composed of. Then, the result of the calculation for a CFB is stored and will be used whenever the CFB is used by any application.

By having a deeper look at Lednicki’s research we identify common points but also different approaches and goals. He mainly focused on the analysis of event inputs and outputs and investigated how we can calculate the different execution paths of a BFB. In the case of data dependency we focused on data inputs and outputs and we defined the algorithms that are needed in order to detect the dependency. On the other hand, both cases are related in terms of the similar logic that co-exists behind the algorithms. Our goal is to investigate how to detect data dependency on the lower level, which is BFB, store this information and when the BFB is used for the composition of a CFB it will be used in order to detect the data dependency between the CFB’s data inputs and outputs. Similarly, the data dependency that will be detected in a CFB will be stored and used when the CFB is used for the composition of a FBN.

Outside of the scope of IEC 61499 standard, an extensive research has been conducted for detecting data dependences between program statements. Identifying data dependencies between program statements plays a vital role in optimizing and parallelizing compilers. As Psarris and Kyriakopoulos explained in [23], the results of the data dependency analysis will be used by the compiler in order to enhance memory locality and perform efficient scheduling. In addition, it will provide crucial information for identifying loop iterations that can be executed in parallel on a multiprocessor system [23]. In general, two statements are data dependent in the case when they both access the same memory location and one or both of them writes on it [23]. Therefore, the compiler needs to analyze array reference patterns. The problem of data dependence in this concept can be explained even further with the following snippet of code of [24]:

```
1:   for i = 11 to 20 do
2:       a[i] = a[i-10] + 3
3:   end for
4:   for i = 11 to 20 do
5:       a[i] = a[i-1] + 3
6:   end for
```
In the first loop, the memory locations that are read do not overlap with the memory locations that are written since the values that are read are written before the loop begins. Thus we conclude that these references are independent. This is not the case in the second loop where the values that are read in each iteration are written in the previous one. Therefore these array references are dependent. Consequently, it is not possible to execute the iterations of each loop in parallel [24]. Maydan in [25] and Psarris in [23] state that the problem of detecting data dependency between array references is equivalent to integer linear programming and therefore it cannot be solved efficiently.

Despite this, there are several data dependence techniques that can solve specific cases of the problem in polynomial time [23].

In [26] U. Banerjee discusses the GCD Test. This technique is based on a theorem of the elementary number theory [23]. According to this theorem, an integer solution exists for a linear equation if and only if the greatest common divisor of the coefficients on the left-hand side of the equation (LHS) divides evenly the constant term on the right-hand side (RHS) [23, 26]. The GCD test checks if this condition holds. If not, then there is no integer solution to the system and therefore there is no dependency. If yes, then the test concludes that dependency does not necessarily exist [23]. The GCD test has been used as a basis for several tests even though it is an inexact test, since it is a necessary but not sufficient condition for the existence of data dependency [23].

In conjunction with the GCD Test U. Banerjee in [26] analyzes the Banerjee Test. This test is based on the Intermediate Value Theorem. It calculates the maximum and minimum values that an expression on the LHS can achieve, given as a bound each of the variable involved [23]. Once the maximum and minimum values are calculated, then the test detects if the constant on the RHS is in between these two values. If not, then we conclude that there is no integer solution and therefore there is no data dependency. If yes, then it concludes that there is an integer solution to the linear equation [23]. Because of the fact that the test cannot distinguish between integer and real solutions we cannot conclude that a dependency exists, but only that it may exist, which makes it an inexact test [23].

In [27] and [28] Kong, Psarris and Klappholz discuss the I-Test, which is based on the Banerjee Test. The improvement that is present on this test is that it can distinguish between integer and real solutions and therefore it can conclude for the existence of data dependences [23].

Pugh in [29] presented the Omega Test, which is another test that is used in order to detect dependency between two array references. It has worst case exponential time complexity, but for many cases Pugh shows that the Test has low order polynomial time complexity [29]. It is based on an extension of a linear programming method called Fourier-Motzkin variable elimination [23, 29].

These tests focus mostly on analyzing one dimensional arrays, while they are quit powerless when it comes to multi-dimensional arrays. Li, Yew and Zhu in [30] present an algorithm, the $\lambda$-test for analyzing multi-dimensional array references and conclude
whether or not there is data dependence between them. They show that the $\lambda$-test has higher efficiency and accuracy comparing to previous algorithms.

All the above mentioned techniques determine data dependencies in high level programming languages source code in order to optimize and parallelize compilers [31]. Amme in [31] presents an algorithm that can be used in order to detect data dependence on assembly language code. The purpose of detecting data dependencies on assembly code is to increase instruction level parallelism [31]. They categorize the data dependencies into accesses to registers and accesses to memory and in order to conclude whether data dependency exists, they use the information from reaching definitions and reaching that they obtain by a monotone data flow analysis [31].

Kuo et al. [17] proposed a technique for retrieving worst-case reaction time of a function block program. In order to achieve this, they analyze the states and take into consideration the dependency between data.
8. Conclusion

Proving a method to characterize and analyze data dependency among data output port and data input ports plausible, was quite an interesting journey. This thesis report presents an approach how to tackle data dependency analysis in basic and composite function blocks and their description on the level of FB interface. Having the information regarding data dependency on the level of FB interface, we can use it in order to define data dependency of an FBN and consequently of a distributed application. Formulation of data dependency analysis for basic function blocks is given by an algorithm which analyzes the BFBs internal ECC, and builds the BFBs dependencies based on dependency data defined for the executed algorithms. Formulation of data dependency analysis for composite function blocks is given by two algorithms. In the first algorithm we identify all the data outputs of the CFB and for each of them we execute the second algorithm. This algorithm will perform the actual analysis and will return the results. In order to represent the dependencies on the FB interface level, we have defined a dependency matrix, which describes which outputs are dependent on which inputs. As a part of the thesis we have developed an implementation in form of a plug-in for 4DIAC-IDE. The plug-in executes the proposed algorithms by identifying whether the FB is basic or composite one. Then, depending on the type of the FB the corresponding algorithm is performed in order to identify the data dependency. The plug-in was used to show that the proposed algorithms can be applied to real-world FBs.

8.1 Future work

This section presents some of the possibilities regarding future research based on the work described in this thesis report.

Data dependency analysis presented in this thesis report assumes that the dependency between output and input data ports is already known. This information can be retrieved from the code of each algorithm. First suggestion for the future work would be to extend this analysis method on the code itself. Looking into code would allow us to know which exact variable influenced the output. In order to achieve that, we need to use a grammar of ST language, since the algorithms are written in ST, parse the code of each algorithm and store the dependency between data inputs and data outputs in a matrix.

In section 5.2.5 we explained two limitations of the proposed analysis method. The first one was related to the fact that we do not take into consideration data dependencies types 7 and 8. It is a great challenge to try to extend the suggested solution so that it covers these two cases as well.

Furthermore, the proposed analysis method takes into consideration that only the data outputs, used in the algorithms of the direct destination state, are dependent on data inputs and internal variables that are used in the guards of the transition. Extending the suggested solution in order to cover the case where data outputs of later states are also dependent on the guard would complete the current solution.

Finally, the solution regarding the CFBs handles the case where all the internal FBs of it are BFBs. Thus the suggested solution can be extended so that it provides a complete data dependency analysis of CFBs.
References


**Abbreviations**

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IEC</td>
<td>International Electrotechnical Commission</td>
</tr>
<tr>
<td>FBs</td>
<td>Function Blocks</td>
</tr>
<tr>
<td>BFB</td>
<td>Basic Function Block</td>
</tr>
<tr>
<td>CFB</td>
<td>Composite Function Block</td>
</tr>
<tr>
<td>4DIAC</td>
<td>Framework for Distributed Industrial Automation and Control</td>
</tr>
<tr>
<td>ABB</td>
<td>Asea Brown Boveri</td>
</tr>
<tr>
<td>PLCs</td>
<td>Programmable Logic Controllers</td>
</tr>
<tr>
<td>FBD</td>
<td>Function Block Diagram</td>
</tr>
<tr>
<td>LD</td>
<td>Ladder Diagram</td>
</tr>
<tr>
<td>ST</td>
<td>Structure Text</td>
</tr>
<tr>
<td>IL</td>
<td>Instruction List</td>
</tr>
<tr>
<td>SFC</td>
<td>Sequential Function Chart</td>
</tr>
<tr>
<td>POU</td>
<td>Program Organization Unit</td>
</tr>
<tr>
<td>FUN</td>
<td>Function</td>
</tr>
<tr>
<td>PROG</td>
<td>Program</td>
</tr>
<tr>
<td>SIFB</td>
<td>Service Interface Function Block</td>
</tr>
<tr>
<td>FBN</td>
<td>Function Block Network</td>
</tr>
<tr>
<td>ECC</td>
<td>Execution Control Chart</td>
</tr>
<tr>
<td>LHS</td>
<td>Left-hand side</td>
</tr>
<tr>
<td>RHS</td>
<td>Right-hand side</td>
</tr>
</tbody>
</table>