Automated Test Generation for Structured Text Language using UPPAAL Model Checker

Mälardalens Högskola

Akademin för Innovation, Design och Teknik

Filip Marković

Master Thesis

08.02.2015.

Examiner: Björn Lisper

Supervisors: Adnan Čaušević and Eduard Paul Enoiu
Abstract

In this research we addressed the problem of test case generation for the Structured Text (ST) programming language from the IEC 61131-3 PLC standard. ST is a non-executable programming language with a standardization purpose for the runtime PLC programming languages. Therefore, it needs to be formalized before the test case generation. For that purpose we used UPPAAL model checker because of its analogy with certain elements in ST language and the possibility of deriving test cases for achieving maximum logic coverage. Although UPPAAL model checker is not specifically intended for the purpose of test case generation we overcome this constraint by conducting the transformation defined in this research. In order to achieve maximum logic coverage we use a defined annotation concept for targeted logic elements (clauses and predicates). We also showed how to implement a tool-supported approach suitable for industrial adoption. With that in mind we also conducted test generation results and performance analysis, comparing them with results provided in similar research for IEC 61131-3 FBD programming language. The results in this thesis show performance improvements in terms of generation time and memory consumption when using this novel transformation. With the approach defined in this thesis, test generation for certain type of FBD programs, which are translatable to ST, could be improved in terms of efficiency of generating these tests. Finally, we provided the answer to the question of what is the achievable formalization level from ST to UPPAL model checker.
Contents

Abstract ...................................................................................................................................... 2
1. Introduction ............................................................................................................................. 6
  1.1 Problem formulation .......................................................................................................... 7
  1.2 Approach and results ........................................................................................................ 8
2. Background ............................................................................................................................. 9
  2.1 Programming logic controller .......................................................................................... 9
  2.2 State of the art .................................................................................................................. 11
  2.3 Language specifications .................................................................................................... 12
    2.3.1 Structured Text ............................................................................................................ 12
      Elements inherited from the standard .............................................................................. 12
      Variables ........................................................................................................................... 17
      Variable sections .............................................................................................................. 17
      Standard functions .......................................................................................................... 18
      Standard function blocks ............................................................................................... 18
      Specifically ST elements ................................................................................................. 19
    2.3.2 UPPAAL ...................................................................................................................... 22
      Timed automata ................................................................................................................ 22
      UPPAAL extension for Timed Automata ........................................................................ 24
      Transition defining elements .......................................................................................... 25
      C language subset .......................................................................................................... 26
      Operators ........................................................................................................................ 27
  2.4 Testing and logic coverage ............................................................................................... 30
3. Method .................................................................................................................................... 32
4. Technical design ..................................................................................................................... 33
  4.1 Overview of ST-UPPAAL transformation and test case generation ............................... 33
    4.1.1 ST code example ........................................................................................................ 34
  4.2 Global picture of the transformed model ......................................................................... 35
  4.3 Control template ............................................................................................................... 36
  4.4 Read inputs templates ...................................................................................................... 40
  4.5 Function/FB templates ..................................................................................................... 42
  4.6 Global declarations .......................................................................................................... 44
    4.6.1 Variable declarations ............................................................................................... 48
      Limitations ...................................................................................................................... 48
  4.7 Function C subset implementation .................................................................................. 49
    4.7.1 Local variables implementation ............................................................................... 51
    4.7.2 Behavior transformation ......................................................................................... 52
      Keywords, operators and statements transformation ...................................................... 52
  4.8. Logic coverage analysis ................................................................................................. 59
    4.8.1 Predicate annotation and monitoring ...................................................................... 59
    4.8.2 Clause annotation and monitoring ........................................................................... 62
    4.8.3 Reachability property ............................................................................................... 63
  4.9 Test case generation ......................................................................................................... 65
5. Results and evaluation .......................................................................................................... 67
  5.1 Experimental comparison ............................................................................................... 70
  5.2 Related work discussion ................................................................................................. 75
6. Conclusion ............................................................................................................................. 77
  6.1 Future work ....................................................................................................................... 77
List of figures

1.1: FBD diagram with ST expansion................................................................. 7
1.2: PLC structure........................................................................................... 9
1.2: PLC hardware based structure................................................................. 9
1.3: Possible invocations between function (F), function block (FB) and Program ......... 13
1.4: Timed automata example......................................................................... 23
1.5: Formal definition of C subset declarations .............................................. 26
1.6: Formal definition of C subset statements................................................ 27
1.7: Research process...................................................................................... 32
1.8: Transformation diagram ......................................................................... 33
1.9: ST code example 1.................................................................................. 34
1.10: Overview of the UPPAL model ............................................................... 35
1.11: Control template ................................................................................... 37
1.12: Input assignment process of the control template................................. 38
1.13: Execution of functions and FBs behavior .............................................. 38
1.14: UpdateOutputs → KeepInputs transition .............................................. 39
1.15: Automata stop and continuation ............................................................. 40
1.16: Read boolean-type template ................................................................. 40
1.17: Read inputs templates from ST Example 1 ........................................... 41
1.18: Boolean-type read template, definition ................................................. 41
1.19: Integer-type read template, definition .................................................. 42
1.20: Functions from the ST example 1........................................................... 43
1.21: Behavior function definition ................................................................. 43
1.22: Logic coverage function definition ....................................................... 44
1.23: Global declarations of ST example 1..................................................... 46
1.24: Initialization of the ST example 1........................................................... 49
1.25: Initialization definition ......................................................................... 49
1.26: Update outputs function of ST example 1............................................. 50
1.27: Update label of the function1 (from ST example 1) ................................ 50
1.28: Local variables declaration section (in Function Blocks) ....................... 51
1.29: Local variables declaration section (in Functions)................................. 51
1.30: Behavior part of the ST example 1.......................................................... 52
1.31: Nested if statements (red – unwritten closed)....................................... 53
1.32: Transformation of case statements....................................................... 55
1.33: ST FOR statement (x, y and z are integer values) ................................ 55
1.34: ST FOR transformation ....................................................................... 56
1.35: ST FOR transformation (increment/decrement construct) ...................... 56
1.36: Executable for statement ..................................................................... 57
1.37: ST WHILE statement ........................................................................... 57
1.38: ST WHILE transformation .................................................................... 57
1.39: ST REPEAT statement ......................................................................... 58
1.40: ST REPEAT transformation ................................................................. 58
1.41: Example for boolean assignment monitoring ...................................... 60
1.42: Definition of boolean assignment predicate annotation ........................ 60
1.43: Definition of IF statement’s predicate annotation ................................... 61
1.44: Definition of while statement predicate annotation ............................... 62
1.45: Definition of clause annotation ............................................................. 63
1.46: ST example of clause annotation (code injection)................................. 63
1.47: Test case iteration in control template (ST example) ............................. 65
1.48: UPPAAL diagnostic trace .................................................................... 66
5.1: Test suite 1 for LATCH1_I......................................................................... 67
5.2: Test suite 2 for LATCH1_I................................................................. 68
5.3: Test suite 3 for LATCH1_I................................................................. 68
5.4: LATCH1_I logic elements annotation ........................................... 69
5.5: Fan_Control FBD program ............................................................ 70
5.6: Fan_Control ST program ................................................................. 71
5.7: Fan_Control FBD test suite 1........................................................... 72
5.8: Fan_Control ST transformation test suite 1 ................................. 72
5.9: Fan_Control FBD test suite 2........................................................... 73
5.10: Fan_Control ST test suite 2 ......................................................... 73
5.11: Fan_Control FBD test suite 3 ......................................................... 74
5.12: Fan_Control FBD test suite 3 ......................................................... 74

List of tables

2.1: Elementary ST data types ............................................................ 14
2.2: Elementary ST data types with specifications ............................... 15
2.3: N bits notes explanations .............................................................. 16
2.4: Variable sections ........................................................................ 17
2.5: ST operators and their precedence ................................................. 19
2.6: ST formal definition ....................................................................... 21
2.7: C subset data types ...................................................................... 26
2.8: C subset operators ........................................................................ 28
2.9: Precedence order of C subset operators ........................................ 29
4.1: Transformation of operators ........................................................ 54
5.1: Logic coverage analysis of Test suite 1 ........................................ 69
5.2 Performance analysis for breadth first SO with shortest trace ........... 73
5.3 Performance analysis for depth first SO with trace set to some .......... 74
5.4 Performance analysis for random depth first SO with trace set to some 75
5.5 State analysis ................................................................................. 76
1. Introduction

IEC 61131-3 is an international standard that summarizes requirements of programming logic controllers (PLC). It is created by the International Electrotechnical Commission (IEC) [12]. The main goal of the standard is to provide consistency of integration, norms and reusability of PLCs. Before the standard was established different PLC manufacturers had difficulties in meeting these criteria. As soon as the IEC 61131-3 standard enabled the reusability of PLCs, the need for their verification and validation increased. It is commonly important to reach adequate level of risk in order to integrate PLC in certain fields.

All of the programming languages within the standard are not executable. In this research we are addressing Structured Text (ST). ST is a textual language which has the biggest level of expression among the languages from IEC 61131-3 standard [3]. It contains the logics and concepts which can be implemented with run-time PLC’s programming languages [2]. Because of the issues stated before, it is recommendable to follow the implementation made in some of the standard’s language. If it is done properly, the integration with other PLCs and its usage in different systems would be facilitated. Many programmers and companies implement their PLCs according to the abstract languages from the IEC 61131-3 standard. Therefore it is necessary to validate and verify them from the beginning of development. This is however not an easy task as they cannot be executed on the real hardware.

That is the reason why many researchers nowadays strive towards the formalization of those languages [3,4,5,6]. Transforming the standard’s languages to a formal representation can benefit verification and validation of the program. The formalization process of the standard-implemented PLC program is also known as the reinterpretation/translation of the PLC program (from one source to another) [5].

In this survey paper [5] it is also mentioned that there are two main aims for formalization:

- Verification and Validation, and
- Reverse Engineering

The interest for verification of safety, lifecycle and timing properties of PLC programs is increasing. Different methods are used in order to provide adequate verification. Among them are: static analysis, abstract interpretation, invariant generation and slicing. According to [5], the two most promising directions are: model checking and theorem proving. According to [24], model checking is a method for formal verification of finite-state systems. Therefore, many of the formalization researches strive to transform one of the standard’s programming languages into the input code of some of the existing model checkers [7] (Uppaal, SVM, NuSMV and others). After that, a model checker can automatically generate test cases for the model that is analogical to the implementation. In this thesis we address test generations for logic coverage. Safety standards mandate logic coverage analysis when testing safety-critical software written in ST language.
1.1 Problem formulation

ST serves as the language abstraction which gives specific concepts and guidelines to the executable PLC programming languages. Therefore, we need to formalize it before being able to automatically generate test cases which will provide us with maximum logic coverage for specific ST codes.

Many research papers, that are proposing the formalization of IEC 1131-3 programming languages, mainly covered graphical languages like FBD and LD. The structure of the PLC program in many cases can be described with FBD. But as FBD has limits in its syntax (caused by its graphical nature), ST is sometimes needed in order to expand its expression [2] as shown in Figure 1.1.

![Figure 1.1: FBD diagram with ST expansion](image)

As shown in Figure 1.1, most of the code is written in FBD programming language. However, functional block called Norm is implemented in ST. A lot of research was done in order to enable formalization and maximum logic coverage in FBD, but if implemented code contains only one instance of ST then we need to find the way to formalize this language as well. Therefore, in this research we address the problem from the preceding case, shown in Figure 1.3, and similar ones. With that in mind we created rules for the formalization and automatic test generation of ST programming language. By pointing the maximum and not 100% logic coverage of tested programs, we mainly refer to the fact that 100% logic coverage is implementation dependent. We can easily create a program which will reach some absorbing logical state and will not be able to alter it by any given input. The logic that can’t be reached shouldn’t be taken into account because of this issue.

As stated before, automatic test generation and maximum logical coverage could be provided by model checkers [7]. In this case it means that we need to transform ST into the input code of a particular model checker (UPPAAL in this research). Other options would be to transform ST directly to some formal programming language (C [23]) and then create an algorithm for test case generation which can produce maximum logic coverage.

One of the main problems encountered during this process is the level of possible formalization, which depends on the targeted language. By conducting the transformation into the input model, used by model checkers, we encounter the problem...
with ranges of different data types. Some model checkers cannot produce test cases for ranges of float numbers or even cannot produce models for complex mathematical algorithms because of the time consumption and slow performance in executing such a task. Some researchers also point the problems within ST programming language itself [8]. This could also affect the reachable level of the formalization.

According to [5], there are three levels of formalization:
- Formalization of parts of the control program (algorithms)
- Formalization of the complete programs
- Formalization of the whole control configurations

Based on this discussion, we have devised the following research questions which we will answer with this thesis:

- RQ1: What are the transformation rules needed for translating ST programs into UPPAAL input model?
  - RQ1a: Can this transformed ST model be used using UPPAAL model checker to generate tests achieving logic coverage of the ST program?
- RQ2: What is the achievable level of formalization of ST language if we use UPPAAL model checker for test case generation?

1.2 Approach and results

In this research we use UPPAAL [13] model checker as the targeted tool for modeling and test case generation. UPPAAL is successfully used for the formalization and test case generation of FBD language from the IEC 61131-3 standard [15]. The defined transformation of ST language to UPPAAL model checker generates test cases which achieve maximum logic coverage of the tested ST codes. It also maintains execution order and rules described in the standard [13] and the runtime execution order of the derived transformation model in UPPAAL. Because of that we are able to track covered logic elements of the underlined ST code in each execution cycle. Syntax analogy between ST language and UPPAAL C language subset allows a straightforward annotation of the code with logical elements. By injecting side-effect free snippets of logic monitoring code, we are able to terminate model checker execution by using temporal properties. After this step we analyze the trace provided by UPPAAL and detect the instances of test case cycles. At the end we can derive test suite that satisfies maximum logic coverage of the underlined ST code. We also compared the performance of test suite generation proposed in this research with the one proposed in [15]. Although the research conducted in [15] addresses FBD language, ST and FBD share analogy. Moreover, any FBD diagram can be translated to the ST code [3]. Therefore, we transformed the same program written in both languages with the proposed transformation rules. This comparison primarily showed significant performance improvements in shortest test suite generation which achieves maximum logic coverage. Needed time for the test suite generation of ST code, with breadth first search order, is 0.052s, compared to 0.319s needed for the transformation of the FBD diagram. States stored in those two transformations are constant for each generation, and they are: 6000 stored states (ST test suite generation) compared to 41120 stored states (FBD test suite generation). This difference causes memory consumption improvements in the generation of the underlined ST code for certain search orders (breadth first and depth first).
2. Background

2.1 Programming logic controller

Programming logic controller (PLC) is a type of the microprocessor-based controller. It uses programmable memory in order to store instructions, data and implement various functions such as: logic, timing, counting, arithmetic and others [1]. With these features PLC controls the machines and the processes.

One of the most important properties of PLCs is a reusability. Certain basic controller can be used among different control systems. Consequences are visible in nowadays flexible and cost-effective control systems [1], even though they vary in purpose, complexity and some other properties.

PLCs are mainly optimized for control tasks and the industrial environment. In many other terms, they are quite similar to the computers [1]. Next to the industrial prerequisites such as resistance to: vibrations, temperature, humidity and other risk factors, PLCs share one more important characteristic. They all have an interfacing for inputs and outputs inside the controller (Figure 2.1).

![Figure 2.1: PLC structure](image1)

There are two main perspectives for viewing and manipulating programmable logic controllers. First one is hardware-oriented, and the second one is software oriented. From hardware aspect, according to [1], PLC system is generally consisted of:

- central processing unit (CPU),
- power supply unit,
- programming device,
- memory unit,
- input and output sections and
- communication interfaces.

![Figure 2.2: PLC hardware based structure](image2)
PLC’s software and hardware are highly analogical. The purpose of programming device is to provide specified program into the program and data memory unit which communicates with the processor (shown in Figure 2.2). Input sections are the places where CPU is provided with information from external devices. On the other side, output sections are the places where CPU provides the information to external devices [1]. Inputs could be derived from switches, different type of sensors (photoelectric cells, temperature sensors...). Outputs could be provided to motor starter coils, solenoid valves etc.

One of the main principles that must be met in PLC software is an easiness of programming and understanding of the programming language [1]. In the first years of the PLC history, different manufacturers used different languages, mainly manufacturer-dependent [2]. Because of that, reusability of PLCs in the industry was limited. This also caused the issue of development time and cost [2]. By the time it was clear that it is important to standardize PLC programming languages in order to enable better integration between PLCs of different manufacturers. Providing ready-made standardized software components also resolved issues of reusability, development time and the development cost. It is important to mention one more organization which revises and updates the standard - PLCopen. PLCopen is a manufacturer independent international organization for PLC programming harmonization. It is an organization which enabled the second revision of the IEC 61131-3 standard and it has several activities with the goal of striving towards the standard improvement [2]. It defined three different levels for programming systems certification. The second one considers reusability [2]. Therefore, the IEC 61131-3 standard was constituted. It primarily describes concepts and guidelines for creating PLC projects. It also could be seen as a guideline for PLC programming [2]. Each particular programming system can keep to the standard in a certain amount. This amount is evaluated through 3 possible levels: Base Level (BL), Reusability Level (RL) and Conformity Level (CL) [2]. Each of them certifies certain programming system and declares how it can be treated, reused or handled.

IEC 61131-3 standard consists of 5 programming languages: Structured Text (ST), Function Block Diagram (FBD), Ladder diagram (LD), Instruction List (IL) and Sequential function chart (SFC) [2]. Among them, ST and IL are the only textual programming languages, whilst the others are graphical ones.

The main building unit of IEC 61131-3 is a POU (Program Organization Unit). It could be seen as the smallest independent unit of the PLC software program [2]. There are three types of POU units: Function (FN), Program (PROG) and Function block (FB). Program is often referred as the main program. It is an initial place for the various variable initializations and contextual declaration [2]. All of them can be expressed with ST language (which is not the case for some others standard’s programming languages). There are even assertions that equivalent code for ST can be derived from the code from any other IEC 61131-3 programming languages, but not vice versa [3]. However it is important to point that this is possible only in a certain abstract level when it comes to IL, as IL can directly manipulate physical memory unlike the ST. According to [2], ST’s main purpose, among the other programming languages of the standard, is the representation of complex algorithms, mathematic calculations and control tasks.

Because of above mentioned facts and the fact that ST can express FBD or even enlarge its level of expression, ST is the main interest point of this research.
2.2 State of the art

There are several different techniques proposed in the recent researches which are used for the formalization/transformation of the IEC 61131-3 programming languages. We can classify them in the following categories:

- XML-based transformations [4,9,10],
- compiler-based formalizations [3],
- formal language transformations [6,8,11,23] and
- model-checking transformations [7, 15, 20]

PLCopen has a branch called TC6 which establishes an XML standard for all of the IEC 61131-3 programming languages. One of the main goals of these XML representations is re-engineering and visualization but also the formalization of the PLC programs. Those standard representations are mainly used for the formalization to the vendor-independent languages as stated in [10].

Compiler-based formalization, consists of creating a ST language compiler which produces universal code that afterwards could be executed on a different virtual and real machines. In the tool-approached research proposed by Rzonca et al. [3], they use a CPDev tool in order to generate code (from ST program) which afterwards can be executed on Java-like virtual machines. This approach uses compiler-based units such as: parser, scanner and code generator. However, this research does not cover the verification part of the ST programs.

Formal language transformation consists of direct transformation from ST programs to the programs written in some of the formal programming languages [8,23]). In the paper proposed by Kabra et al. [8], they create a translator from ST language to MISRA-C language. MISRA-C is a subset of C used in safety-critical applications. Although they do not address the automated test generation or some other verification possibilities, they addressed some of the standard’s issues [11]. In papers [6,23] formalization is made from ST to ANSI-C programming language. This language is usually used for the prototype systems [6]. In paper proposed by Sadolewski et al. [6] the formalization rules are proposed and later on adapted for the verification purposes in [23]. However in [23] the research does not cover test generation nor the concept of the logic coverage of the ST programs. It uses tool-supported approach for the verification of the compliance between specification and the code, which is performed by Coq tool [23].

Model checking transformation consists of defining set of rules which can simulate the execution of the ST program in one of the existing model checkers. In paper proposed by Gourcuff et al. [7] the main point was addressed to the analysis of the model-checking scalability for the NuSMV model-checker. They also propose the method for the translation of the ST language to NuSMV model. This method does not cover logic coverage analysis and test generation. The two most important related works with this thesis are papers of Ammann et al. [20] and Enoiu et al. [15]. Amman’s paper explore the role of model checkers in software testing and proposes general approaches for using model checkers in test generation evaluation. Based on this work, in the paper proposed by Enoiu et al. [15] they create an approach for the automated test case generation of programs written in FBD language using UPPAAL model checker. These two papers provided the main guidelines for the approach defined in this thesis.
2.3 Language specifications

The required transformation\(^1\)/translation could be seen as a conceptualization of the mathematical function. In this transformation - ST is a domain and UPPAAL elements and language subsets represent the codomain of the function. Both of these entities represent the finite sets of building elements with certain levels of syntax and expression. Before doing the transformation it is necessary to analyze them both. By observing their building units, data types and constraints we can do the first selection of the possible transformations among those units.

2.3.1 Structured Text

*Elements inherited from the standard*

As stated before, the main building unit of the 61131-3 standardized PLC program is POU (Program Organization Unit) which could be presented in three following forms:
- as a function,
- a function block\(^2\), or
- a program

**Function** is a stateless POU type \([12]\). It does not store nor persist any state. Important properties of the functions are:
- provides a result which could be an one-data element or a multi-valued element - array/structure
- can provide output variable (one or more) and they can be multi-valued elements as well

**Function block (FB)** is a type of POU which unlike functions has its own state. Its main purpose is to modularize and structure a straightforwardly defined portion of the program. It is analogical to the class-object manifestation in the OO programming. FB is existent in two forms - as a type and as an instance.

As a type, function block consists of:
- the definition, which is structured of: input, output, internal variables, and
- a set of operations which should be performed when the instance of the function block is called

From the instance perspective:
- it is a multiple specifically named usage of a function block type, and
- each instance should have an identifier associated with it, also it should have a data structure containing the static input, output and internal variables

**Program** is defined as a "logical assembly of all the programming language elements and constructs necessary for the intended signal processing required for the control of a machine or process by a PLC-system", by IEC 61131-3 standard \([12]\).

\(^1\) In the following pages of the thesis, the formalization process from ST to UPPAAL model will be referred as a transformation, because UPPAAL is not a formal language but the model checker which contains a C language subset among many other concepts.

\(^2\) Function block will be abbreviated as FB in the following pages.
As shown in Figure 2.3 we show what are the possible invocations between different POU types. Program can invoke functions and function blocks. Functions can only invoke functions, function blocks can invoke functions and function blocks.

In order to understand a declaration of these organization units we need to define and present common language elements of the ST programming language. PLC programs are made of many different basic language elements. These elements together create declarations and statements. Basic division of ST's language elements according to [1] addresses the following elements:

- Delimiters
- Keywords
- Literals
- Identifiers

**Delimiters** are special characters with different meanings which make strict borders between different language elements. Delimiter examples in ST are the following characters: colon (':'), comma (','), parentheses - ('(' and ')'), asterisk ('*'), equal ('=), plus ('+), minus ('-') and semicolon (';').

**Keywords** represent the standard identifiers which belong to the word set of the programming language itself (ST in this case). Each keyword has the intended purpose and syntax which is clearly defined by IEC 61131-3 standard. User cannot override keywords for its own purpose. Standard is not case-sensitive when it comes to the keywords so "retain" and "RETAIN" is treated the same. Reserved keyword could represent:

- name of elementary data type
- name of standard function
- name of standard function block
- names of input and output parameters of standard function blocks
- name of input parameters of standard function

Some of the keywords in ST are the following elements: (RETAIN), (VAR_INPUT), (END_VAR) and (FUNCTION).

**Literals** are the value representations for different data types. Format of the literal depends on the data type of the variable. Data type further defines the possible value ranges. According to [1] there are three basic types of literals:

- Numeric literal (numeric values for bit string numbers, integers and floating-point numbers)
• Character string literals (character string values in single-byte or double byte representation)
• Time literals (values for date, time and duration)
The examples of literals in ST are: \(64, 4.943 \times 10^{-12}, 16\#0B\) for a bit string numbers, ('this is a text') for a character strings, and \(\text{tod}\#12:16:28.44\) for a time literal.

**Identifiers** are alphanumeric character strings for variable names, labels, POUs etc. specified by the PLC programmer. They must start with a letter or a single underline character followed by some amount of letters, digits and/or underline characters. Standard is not case sensitive and therefore an identifier "CIRCLE_ST" is regarded the same as "circle_ST". Identifiers can be assigned to different language elements, such as:
• Jump and network labels
• Enumeration constraints
• Configurations, resources, tasks/run-time programs
• Programs, functions, function blocks
• Access paths
• Variables (general, symbolic and directly represented variables)
• Derived data types, components of a structure
• Transitions, steps, action blocks

Identifier examples are: (Var_2, Inp3), (EmergOff), (Real_Out) and (RealAdd).

**Pragmas** are the language structures used for automatic pre-processing and post-processing of the programs. The syntax and semantics of pragmas is implementation-dependent and therefore is not defined by standard. An example of pragmas is:
\{Author M, Version 3\}\{m := 3\}

Next to the above mentioned language elements it is important to address a set of possible data types in ST. **Data type** is a classification of each variable and literal [1]. It defines their possible values, operations that can be performed on them and the way the values are stored. According to standard [12], a set of pre-defined or elementary data types is specified. The main characteristic of elementary data types is their data width and their possible value range. Elementary data types are shown in Table 2.1.

<table>
<thead>
<tr>
<th>Boolean/string/bit</th>
<th>Signed integer</th>
<th>Unsigned integer</th>
<th>Floating point (Real)</th>
<th>Time, duration, date and character string</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOOL</td>
<td>INT</td>
<td>UINT</td>
<td>REAL</td>
<td>TIME</td>
</tr>
<tr>
<td>BYTE</td>
<td>SINT</td>
<td>USINT</td>
<td>LREAL</td>
<td>DATE</td>
</tr>
<tr>
<td>WORD</td>
<td>DINT</td>
<td>UDINT</td>
<td></td>
<td>TIME_OF_DAY</td>
</tr>
<tr>
<td>DWORD</td>
<td>LINT</td>
<td>ULINT</td>
<td></td>
<td>DATE_AND_TIME</td>
</tr>
<tr>
<td>LWORD</td>
<td></td>
<td></td>
<td></td>
<td>STRING</td>
</tr>
</tbody>
</table>

In Table 2.1, D stands for double, L for long, S for short and U for unsigned. The characteristics of elementary data types are not presented in this table. They are presented in Table 2.2. The most important characteristic in our case is a default value. The default value will be taken if the one is not provided within the initialization of the variable. Another provided information in this table is a number of bits required per data type. This information is derived from the standard itself [12].
# Table 2.2: Elementary ST data types with specifications

<table>
<thead>
<tr>
<th>Description</th>
<th>Keyword</th>
<th>Default init value</th>
<th>N (bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean</td>
<td>BOOL</td>
<td>0,FALSE</td>
<td>1</td>
</tr>
<tr>
<td>Short integer</td>
<td>SINT</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>Integer</td>
<td>INT</td>
<td>0</td>
<td>16</td>
</tr>
<tr>
<td>Double Integer</td>
<td>DINT</td>
<td>0</td>
<td>32</td>
</tr>
<tr>
<td>Long Integer</td>
<td>LINT</td>
<td>0</td>
<td>64</td>
</tr>
<tr>
<td>Unsigned short integer</td>
<td>USINT</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>Unsigned integer</td>
<td>UINT</td>
<td>0</td>
<td>16</td>
</tr>
<tr>
<td>Unsigned double integer</td>
<td>UDINT</td>
<td>0</td>
<td>32</td>
</tr>
<tr>
<td>Unsigned long integer</td>
<td>ULINT</td>
<td>0</td>
<td>64</td>
</tr>
<tr>
<td>Real number</td>
<td>REAL</td>
<td>0.0</td>
<td>32</td>
</tr>
<tr>
<td>Long reals</td>
<td>LREAL</td>
<td>0.0</td>
<td>64</td>
</tr>
<tr>
<td>Duration</td>
<td>TIME</td>
<td>T#0s</td>
<td></td>
</tr>
<tr>
<td>Duration</td>
<td>LTIME</td>
<td>LTIME#0s</td>
<td></td>
</tr>
<tr>
<td>Date</td>
<td>DATE</td>
<td>Implementer specific</td>
<td></td>
</tr>
<tr>
<td>Long Date</td>
<td>LDATE</td>
<td>LDATE#1970-01-01</td>
<td></td>
</tr>
<tr>
<td>Time of day</td>
<td>TIME_OF_DAY/TOD</td>
<td>TOD#00:00:00</td>
<td></td>
</tr>
<tr>
<td>Time of day</td>
<td>TIME_OF_DAY/LTOD</td>
<td>LTOD#00:00:00</td>
<td></td>
</tr>
<tr>
<td>Date and time of Day</td>
<td>DATE_AND_TIME or DT</td>
<td>Implementer specific</td>
<td></td>
</tr>
<tr>
<td>Date and time of Day</td>
<td>LDATE_AND_TIME or LDT</td>
<td>LDT#1970-01-01-00:00:00</td>
<td></td>
</tr>
<tr>
<td>Variable-length single-byte character string</td>
<td>STRING</td>
<td>'' (empty)</td>
<td>8</td>
</tr>
<tr>
<td>Variable-length double-byte character string</td>
<td>WSTRING</td>
<td>&quot;&quot; (empty)</td>
<td>16</td>
</tr>
<tr>
<td>Single-byte character</td>
<td>CHAR</td>
<td>'$00'</td>
<td>8</td>
</tr>
<tr>
<td>Double-byte character</td>
<td>WCHAR</td>
<td>&quot;$0000&quot;</td>
<td>16</td>
</tr>
<tr>
<td>Bit string of length 8</td>
<td>BYTE</td>
<td>16#00</td>
<td>8</td>
</tr>
<tr>
<td>Bit string of length 16</td>
<td>WORD</td>
<td>16#0000</td>
<td>16</td>
</tr>
<tr>
<td>Bit string of length 32</td>
<td>DWORD</td>
<td>16#0000_0000</td>
<td>32</td>
</tr>
<tr>
<td>Bit string of length 64</td>
<td>LWORD</td>
<td>16#0000_0000_0000_0000</td>
<td>64</td>
</tr>
</tbody>
</table>

Table 2.3 shows the notes from the N (bits) column entries in the Table 2.2. It is also derived from the standard [12].
Table 2.3: N bits notes explanations

<table>
<thead>
<tr>
<th>Note</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>The range of values and precision of representation in these data types is Implementer specific.</td>
</tr>
<tr>
<td>c</td>
<td>The range of values for variables of this data type is from -(2N-1) to (2N-1)-1.</td>
</tr>
<tr>
<td>d</td>
<td>The range of values for variables of this data type is from 0 to (2N)-1.</td>
</tr>
<tr>
<td>e</td>
<td>The range of values for variables of this data type shall be as defined in IEC 60559 for the basic single width floating-point format. Results of arithmetic instructions with denormalized values, infinity, or not-a-number values are Implementer specific.</td>
</tr>
<tr>
<td>f</td>
<td>The range of values for variables of this data type shall be as defined in IEC 60559 for the basic double width floating-point format. Results of arithmetic instructions with denormalized values, infinity, or not-a-number values are Implementer specific.</td>
</tr>
<tr>
<td>g</td>
<td>A numeric range of values does not apply to this data type.</td>
</tr>
<tr>
<td>h</td>
<td>The possible values of variables of this data type shall be 0 and 1, corresponding to the keywords FALSE and TRUE, respectively.</td>
</tr>
<tr>
<td>i</td>
<td>The value of N indicates the number of bits/character for this data type.</td>
</tr>
<tr>
<td>j</td>
<td>The value of N indicates the number of bits in the bit string for this data type.</td>
</tr>
<tr>
<td>k</td>
<td>The maximum allowed length of STRING and WSTRING variables is Implementer specific.</td>
</tr>
<tr>
<td>l</td>
<td>The character encoding used for CHAR, STRING, WCHAR, and WSTRING is ISO/IEC 10646.</td>
</tr>
<tr>
<td>m</td>
<td>The data type LTIME is a signed 64-bit integer with unit of nanoseconds.</td>
</tr>
<tr>
<td>n</td>
<td>The data type LDATE is a signed 64-bit integer with unit of nanoseconds with starting date 1970-01-01.</td>
</tr>
<tr>
<td>o</td>
<td>The data type LDT is a signed 64-bit integer with unit of nanoseconds with starting date 1970-01-01-00:00:00.</td>
</tr>
<tr>
<td>p</td>
<td>The data type LTOD is a signed 64-bit integer with unit of nanoseconds with starting time midnight with TOD#00:00:00.</td>
</tr>
<tr>
<td>q</td>
<td>The update accuracy of the values of this time format is Implementer specific, i.e. the value is given in nanoseconds, but it may be updated every microsecond or millisecond.</td>
</tr>
</tbody>
</table>

Next to the elementary data types, standard covers arrays and data structures as well. **Arrays** represent consecutive data elements of the same type in memory, while **data structures** represent programmable hierarchical structures derived from elementary or derived data types. With defined data types we can continue by defining variables.
Variables identify data objects whose content might change. Unlike the literals, value of the variable may change over time. They are declared inside specifically designated variable sections. According to the standard [12], variable can be declared using:

- an elementary data type or
- a previously-defined type or
- a reference type or
- an instantly user-defined type

A variable can be a single-element variable, a multi-element variable (ARRAY or a STRUCT) and a reference (a variable which refers to some other variable or function block instance).

Variable declaration includes:

- name or a list of variable names which are declared
- a colon (":" ) and
- a data type which can be followed with the variable initialization

Example:

```
VAR myVar : INT; END_VAR
```

Variable sections are dedicated variable declaration segments which may be used in function blocks, functions or programs. Commonly, each POU consists of several variable sections. Variable sections not only distinguish and declare different types of variables, they also affect the access rights of each declared variable. According to the standard [12], Table 2.4 shows defined variable sections and their access rights:

### Table 2.4: Variable sections

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Usage</th>
<th>Access rights</th>
</tr>
</thead>
<tbody>
<tr>
<td>VAR</td>
<td>Internal to entity (function, function block, program)</td>
<td>NA RW</td>
</tr>
<tr>
<td>VAR_INPUT</td>
<td>Externally supplied, not modifiable within entity</td>
<td>W R</td>
</tr>
<tr>
<td>VAR_OUTPUT</td>
<td>Supplied by entity to external entities</td>
<td>R RW</td>
</tr>
<tr>
<td>VAR_IN_OUT</td>
<td>Supplied by external entities, can be modified within entity and supplied to external entity</td>
<td>RW RW</td>
</tr>
<tr>
<td>VAR_EXTERNAL</td>
<td>Supplied by configuration via VAR_GLOBAL</td>
<td>RW RW</td>
</tr>
<tr>
<td>VAR_GLOBAL</td>
<td>Global variable declaration</td>
<td>RW RW</td>
</tr>
<tr>
<td>VAR_ACCESS</td>
<td>Access path declaration</td>
<td>RW RW</td>
</tr>
<tr>
<td>VAR_TEMP</td>
<td>Temporary storage for variables in function blocks, methods and programs</td>
<td>NA RW</td>
</tr>
<tr>
<td>VAR_CONFIG</td>
<td>Instance-specific initialization and location assignment</td>
<td>- -</td>
</tr>
<tr>
<td>END_VAR</td>
<td>Terminates various VAR sections above</td>
<td>- -</td>
</tr>
</tbody>
</table>

In Table 2.4, R stands for read rights, W for write, RW for read and write rights and NA for not accessible. The external access rights are indicated for the calling (external) POU,
while the internal rights are indicated within the internal POU, where the declaration is made. The variable section keywords can be followed by the following qualifiers:

- **RETAIN** (Retentive variables),
- **NON_RETAIN** (Non-retentive variables),
- **PROTECTED** (Accessible from inside the own entity and its derivations),
- **PUBLIC** (Accessible from all entities),
- **PRIVATE** (Accessible from own entity),
- **INTERNAL** (Accessible from the same namespace) and
- **CONSTANT** (variables cannot be modified).

By this, we covered the most important ST language elements and properties which will be mentioned in the research. However, IEC standardized some of the basic functionalities (POUs) as well. Standardized PLC functionality could be observed through standard functions and standard function blocks.

**Standard functions** represent basic logical operators (bit-shifting, addition, comparison etc.). As previously mentioned in POU section, they are stateless. The IEC 61131-3 standard differentiates eight groups of standard functions:

- Data type conversion functions,
- Numerical functions,
- Arithmetic functions,
- Bit-string functions,
- Selection and comparison functions,
- Character string functions,
- Functions for time data types and
- Functions for enumerated data types

**Standard function blocks** represent PLC functions with status or a state information. Typical representatives of the standard function blocks are timers, counters etc. The IEC 61131-3 standard differentiates five groups of standard function blocks:

- Bistable elements or flip flops
- Edge detection
- Counters
- Timers
- Communication function blocks

The aim of this chapter is to compare the structure of ST and UPPAAL. It also tends to point basic ST elements and building blocks in PLC systems.
Specifically ST elements

**ST expressions** are constructs which return a value of some data type after the evaluation. Expressions consist of operators and operands. Operand could be a variable, a literal, enumerated value, function call with a result, FB call instance with a result or another expression. According to the standard [12], there are several rules for the evaluation of expressions:

1. Operands are applied by the operators in a predefined sequence known as operator precedence (shown in the Table 2.5).
2. Equal precedence operators are applied from left to right as written in the expression.
3. If an operator has two operands then the leftmost one should be evaluated first.
4. Boolean expressions may be evaluated to the extent necessary to deliver the final value of the expression. Example: \((A<B) \& (C>D)\) could be evaluated only until \((A<B)\) evaluates to \textit{FALSE}\)
5. Functions and methods may be called within the expression as its elements. In this case they are called in a form of the function name followed by the parenthesis with parameters.
6. If the operator in an expression could be represented as one of the overloaded functions, conversion of operands and results shall follow the rules and examples given above.

**ST operators** are constructs which behave like functions but generally have a different syntax or even semantics than the general functions. Table 2.5 shows all of the ST operators.

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
<th>Precedence</th>
</tr>
</thead>
<tbody>
<tr>
<td>( )</td>
<td>Parenthesis alter the evaluation order</td>
<td>11(Highest)</td>
</tr>
<tr>
<td>Identifier (parameter list)</td>
<td>Evaluation of result of function and method -if a result is declared</td>
<td>10</td>
</tr>
<tr>
<td>^</td>
<td>Dereference</td>
<td>9</td>
</tr>
<tr>
<td>-</td>
<td>Negation</td>
<td>8</td>
</tr>
<tr>
<td>+</td>
<td>Unary plus</td>
<td>8</td>
</tr>
<tr>
<td>NOT</td>
<td>Complement</td>
<td>8</td>
</tr>
<tr>
<td>**</td>
<td>Exponentiation</td>
<td>7</td>
</tr>
<tr>
<td>*</td>
<td>Multiply</td>
<td>6</td>
</tr>
<tr>
<td>/</td>
<td>Divide</td>
<td>6</td>
</tr>
<tr>
<td>MOD</td>
<td>Modulo</td>
<td>6</td>
</tr>
<tr>
<td>+</td>
<td>Add</td>
<td>5</td>
</tr>
<tr>
<td>-</td>
<td>Subtract</td>
<td>5</td>
</tr>
<tr>
<td>&lt;,&gt;,&lt;=&gt;</td>
<td>Comparison</td>
<td>4</td>
</tr>
<tr>
<td>=</td>
<td>Equality</td>
<td>4</td>
</tr>
<tr>
<td>&lt;&gt;</td>
<td>Inequality</td>
<td>4</td>
</tr>
<tr>
<td>&amp;</td>
<td>Boolean AND</td>
<td>3</td>
</tr>
<tr>
<td>AND</td>
<td>Boolean AND</td>
<td>3</td>
</tr>
<tr>
<td>XOR</td>
<td>Boolean Exclusive OR</td>
<td>2</td>
</tr>
<tr>
<td>OR</td>
<td>Boolean OR</td>
<td>1(Lowest)</td>
</tr>
</tbody>
</table>
**ST statements** are instructions that should perform a specified action by some of the executable programming languages. In the most used programming languages, it can be one of three different types: assignment, selection and iteration statement.

**Assignment** statement is used for the replacement of the current value of the single or multi-element variable. The new value can be a result of the expression evaluation. 
*Example: A := B*

**Comparison** statement returns a Boolean value. They are mainly consisting of a variable reference on the left side, followed by the comparison operator and a variable reference on the right side. This statement is of high importance in this research as its structure is very similar to the conditions and can affect condition coverage at the end.

**Selection statements** select one or a group of its component statements for execution according to the specified condition. Selection statements can be *IF* and *CASE*.

**Iteration statements** specify the repetitive execution of the associated statements. Iteration statements can be *WHILE*, *REPEAT*, *EXIT*, *CONTINUE* and *FOR*.

In Table 2.6 we show a grammar of the ST expressions and statements.
### Table 2.6: ST formal definition

<table>
<thead>
<tr>
<th>Element</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expression</td>
<td>Xor_Expr ('OR' Xor_Expr );</td>
</tr>
<tr>
<td>Constant_Expr</td>
<td>Expression; // a constant expression must evaluate to a constant value at compile time</td>
</tr>
<tr>
<td>Xor_Expr</td>
<td>And_Expr ('XOR' And_Expr );</td>
</tr>
<tr>
<td>And_Expr</td>
<td>Compare_Expr ( ('&amp;'</td>
</tr>
<tr>
<td>Compare_Expr</td>
<td>( Equ_Expr ( ('='</td>
</tr>
<tr>
<td>Equ_Expr</td>
<td>Add_Expr ( ('&lt;'</td>
</tr>
<tr>
<td>Add_Expr</td>
<td>Term ( ('+'</td>
</tr>
<tr>
<td>Term</td>
<td>Power_Expr ('''</td>
</tr>
<tr>
<td>Power_Expr</td>
<td>Unary_Expr ('**' Unary_Expr )*;</td>
</tr>
<tr>
<td>Unary_Expr</td>
<td>Constant</td>
</tr>
<tr>
<td>Primary_Expr</td>
<td>Variable Multibit_Part_Access ?;</td>
</tr>
<tr>
<td>Multibit_Part_Access</td>
<td>'.' ( Unsigned_Int</td>
</tr>
<tr>
<td>Func_Call</td>
<td>Func_Access (' ( Param_Assign (;', Param_Assign )* )? )';</td>
</tr>
<tr>
<td>Stmt_List</td>
<td>( Stmt ? ';' )*;</td>
</tr>
<tr>
<td>Stmt</td>
<td>AssignStmt</td>
</tr>
<tr>
<td>AssignStmt</td>
<td>( Variable ':=' Expression )</td>
</tr>
<tr>
<td>Assignment_Attempt</td>
<td>( Ref_Name</td>
</tr>
<tr>
<td>Invocation</td>
<td>( FB_Instance_Name</td>
</tr>
<tr>
<td>Subprog_Ctrl_Stmt</td>
<td>Func_Call</td>
</tr>
<tr>
<td>Param_Assign</td>
<td>( ( Variable_Name ':=' )? Expression )</td>
</tr>
<tr>
<td>SelectionStmt</td>
<td>IFStmt</td>
</tr>
<tr>
<td>IFStmt</td>
<td>'IF' Expression 'THEN' StmtList ('ELSIF' Expression 'THEN' StmtList )* ('ELSE' StmtList)? END_IF;</td>
</tr>
<tr>
<td>CaseStmt</td>
<td>'CASE' Expression 'OF' Case_Selection + ('ELSE' StmtList)? END_CASE;</td>
</tr>
<tr>
<td>Case_Selection</td>
<td>CaseList ':=' StmtList;</td>
</tr>
<tr>
<td>CaseList</td>
<td>CaseList_Atom (', CaseList_Atom)*;</td>
</tr>
<tr>
<td>CaseList_Atom</td>
<td>Subrange</td>
</tr>
<tr>
<td>IterationStmt</td>
<td>ForStmt</td>
</tr>
<tr>
<td>ForStmt</td>
<td>'FOR' Control_Variable ':=' ForList 'DO' StmtList 'END_FOR';</td>
</tr>
<tr>
<td>Control_Variable</td>
<td>Identifier;</td>
</tr>
<tr>
<td>ForList</td>
<td>Expression 'TO' Expression ('BY' Expression )?;</td>
</tr>
<tr>
<td>WhileStmt</td>
<td>'WHILE' Expression 'DO' StmtList 'END_WHILE';</td>
</tr>
<tr>
<td>RepeatStmt</td>
<td>'REPEAT' StmtList 'UNTIL' Expression 'END_REPEAT';</td>
</tr>
</tbody>
</table>
2.3.2 UPPAAL

UPPAAL is a model checking, verification toolbox for the real-time systems [13]. It is developed by Uppsala University and Aalborg University. Since then it was successfully applied in various case studies. The main concept used in UPPAAL is a verification of systems that can be modeled as a network of timed automata. According to [5,24], model checking is a method for formal verification of finite-state systems. It consists of two main concepts [20]:

- A model, which is a state machine described with variables, initial values and conditions under which variables may change values.
- Temporal logic constraints, which is a stopping criteria for the verification. Model checker visits all reachable states in order to verify that temporal logic is satisfied.

Timed automata

Timed automata (TA) is a theoretical paradigm for modeling and verification of real time systems. It is important to mention that timed automata, according to [14] is a general term that describes a finite state Büchi automation which is extended with a set of real-valued variables and modeling clocks. More specific and simplified version called Timed Safety Automata is used in UPPAAL model checker and this one will be briefly described in this section.

A timed finite automata consists of finite set of nodes (locations) and finite set of labeled edges. In general type of timed automata, automata is extended with real-valued variables but in UPPAAL it is extended with integer variables. This kind of automata represents an abstract model of a timed system. Variables represent the logical clocks in the system. When the system is started variables are initialized with zero values and afterwards they may increase synchronously with the same amount. The behavior of the automata is constrained and controlled by guards i.e. clock constraints which could be defined for edges in the automata. Some edge could be taken only if the clock values satisfy the guard which is labeled to that edge (transition).

Formal syntax of timed automata according to [13,14,18] consists of:

- $C$: a set of clocks
- $B(C)$ - a set of conjunctions over simple conditions of the form $x \bowtie c$ or $x - y \bowtie c$, where $x, c \in C, c \in N$ and $\bowtie \in \{<, \leq, =, \geq, >\}$.
- $L$ - a set of locations
- $l_0$ - initial location, where $l_0 \in L$
- $A$ - a set of actions, co-actions and the internal $\tau$ - action
- $E \subseteq L \times A \times B(C) \times 2^C \times L$ - a set of edges between locations with an action, a guard and a set of clocks to be reset
- $I: L \rightarrow B(C)$ - assigns invariants to the locations

Therefore the timed automata $A$ is a tuple $(L, l_0, C, A, E, I)$ where:
- $L$ is a finite set of locations (nodes),
- $l_0 \in L$ is the initial location
- $E \subseteq L \times A \times B(C) \times 2^C \times L$ is a set of edges and
- $I: L \rightarrow B(C)$ assigns invariants to locations
Semantics of TA is defined as follows. Let \((L, l_0, C, A, E, I)\) be a timed automata, semantics is defined as a labeled transition system \(\langle S, s_0, \rightarrow \rangle\), where:

- \(S \subseteq L \times R^C\) is the set of states,
- \(s_0 = (l_0, u_0)\) is the initial state and
- \(\rightarrow \subseteq S \times (R_{\geq 0} \cup A) \times S\) is the transition relation

\[\begin{align*}
&12 \leq y \leq 24 \\
&y = 0
\end{align*}\]

\[\begin{align*}
&15 \leq y \leq 25 \\
&x = 0, y = 0
\end{align*}\]

\[\begin{align*}
&45 \leq y \leq 51 \\
&y = 0
\end{align*}\]

\[\begin{align*}
&x = 1 \\
&x = 0
\end{align*}\]

\[\begin{align*}
&12 \leq y \leq 24 \\
&y = 0
\end{align*}\]

\[\begin{align*}
&15 \leq y \leq 25 \\
&x = 0, y = 0
\end{align*}\]

\[\begin{align*}
&45 \leq y \leq 51 \\
&y = 0
\end{align*}\]

\[\begin{align*}
&x = 1 \\
&x = 0
\end{align*}\]

\[\begin{align*}
&12 \leq y \leq 24 \\
&y = 0
\end{align*}\]

\[\begin{align*}
&15 \leq y \leq 25 \\
&x = 0, y = 0
\end{align*}\]

\[\begin{align*}
&45 \leq y \leq 51 \\
&y = 0
\end{align*}\]

\[\begin{align*}
&x = 1 \\
&x = 0
\end{align*}\]

In Figure 2.4, we show a more concrete example of timed automata. A set of clocks in this automata consists of two clocks \(x\) and \(y\). A set of locations consists of start, loop and end location. Initial location is start and it is indicated by two concentric circles. A set of conjunctions over clocks in this example includes the following:

- \(12 \leq y \leq 24\)
- \(15 \leq y \leq 25\)
- \(x = 1\) and
- \(45 \leq y \leq 51\)

A set of actions consists of work, leave and enter action. In the example, shown in Figure 2.4, there are no invariants. Figure 2.4 shows one specific timed automata, however UPPAAL mostly uses models which represent network of timed automata [19]. As a continuation of the previous definition of timed automata (A) we will define the network according to [13] which states following:

Let \(A_i = (L, l_i^0, C, A, E_i, I_i)\) be a network of timed automata\(^3\) and let \(l_0 = (l_1^0, l_2^0, \ldots, l_n^0)\) be an initial location vector. The semantics of TA network is defined as a transition system \(\langle S, s_0, \rightarrow \rangle\), where:

- \(S \subseteq (L_1 \times L_2 \times \ldots \times L_n) \times R^C\) is the set of states,
- \(s_0 = (l_0, u_0)\) is the initial state and
- \(\rightarrow \subseteq S \times (R_{\geq 0} \cup A) \times S\) is the transition relation.

Transition relation in the network of timed automata is different than the relation in TA. It is expanded with synchronization functions (i.e., a! is correlative with a?) [15]. These functions are responsible for communication between different timed automata. For more information about TA refer to [13,14,15,18].

\(^3\) In the following sections, Timed Automata will be abbreviated with TA.
UPPAAL extension for Timed Automata

Next to the previously defined elements and terms, UPPAAL modeling language extends Timed Automata with the following ones [13].

**Templates** are automata segments defined with locations and edges. Template can also have a set of parameters that can be of any supported data type (defined below). Templates are instantiated by a process assignment which is defined in the system definition.

**Constants** are non-modifiable integer values in this case. They are declared with *const* keyword. Example:

\[
\text{const variable\_name value.}
\]

**Bounded integer variables** defines an integer range variable in UPPAAL. It can be declared as *int[min,max] variable\_name*, where *min* represents lower and *max* represents an upper bound. Bounded integer variables can be used in guards, invariants and assignments (special UPPAAL expressions which will be describe later on).

**Binary synchronization channels** are used for the synchronization between different edges. They can be declared with *chan c* notation. Edge labeled with *c!* synchronizes with another edge labeled *c?*.

**Broadcast channels** similarly as in previous case are used for edge synchronizations. In this case however a sender *c!* can synchronize with arbitrary number of receivers *c?*. Declaration for this type is made with the following notion- *broadcast chan variable\_name*.

**Urgent synchronization** defines the situations when delays must not occur if a synchronization on an urgent channel is enabled. Edges that uses urgent synchronization cannot have time constraints (clock guards).

**Urgent location** is the equivalent to the location which has an extra clock *x* which is reset on all of the incoming edges and has an invariant *x<=0*. This means that time is not allowed to pass in an urgent location.

**Committed locations** are acting even more restrictively than the urgent location. A state is committed if any of the locations in that state is committed. This committed state cannot delay and demands the next transition on an outgoing edge of any of the committed locations.

**Arrays** are same as in ST, and they are allowed for clocks, channels, constants and integer variables. Example of the array declaration is as follows: *clock a[2]*;

**Initialisers** are used for the initialization of the integer variables or arrays of integers. Example of the initialiser in both cases respectively is:

- *int i = 2;*
- *int i[3] = {1,2,3}.*
Record types are similar to the data structures in ST when used in custom type creation and they are declared with the same keyword - \textit{struct}. In the bellow written example the record \( s \) consists of two fields \( a \) and \( b \):

\[
\text{struct \{int } a; \text{ int } b; \} s;
\]

Custom types are defined with the \textit{typedef} construct and can be defined with any other elementary type such as record.

User functions can be defined globally or locally (assigned to the template). Template parameters are accessible from the local functions. In UPPAAL, they are a part of the C subset language which will be described in the following sections.

Next to these elements, it is of importance to mention five more, which are crucial for the creation of TA with UPAAL. Each transition (branch) in UPPAAL can be defined and labeled with one of the expression types defined in the following Section.

\textit{Transition defining elements}

Select label contains list of \textit{name : type} expressions where \textit{name} is a variable and \textit{type} is a defined data type. These variables are accessible on the edges for which they are associated and they will take a non-deterministic value in the range of their defined types.

Guard is a specific type of an expression that must evaluate to a boolean value and must be side-effect free. A guard also can call a side-effect function that evaluates to the boolean value.

Synchronization label can be one of three possible forms: \textit{Expression!}, \textit{Expression?} or empty label. Expression must be free of any side-effect. It must evaluate to a channel and can only refer to integers, constants and channels.

Update label is a list of expressions which have side-effects. It can only refer to the integer variables, constants, and only assign integer values to a clocks. It also can call a function.

Invariant is an expression that satisfies the side-effect free condition. It also satisfies the conditions of possible restrictive usage of clocks, integer variables and constants only. It is a conjunction of the conditions which follow the form: \( x < e \) or \( x \leq e \) where \( x \) is a clock reference and \( e \) evaluates to an integer. Invariant can call a side-effect free function which evaluates to the boolean value, but a clock constraint is not supported in such a function.

UPPAAL model checker has another feature which is mainly addressing functions mentioned in some of the preceding elements (user functions). Those functions, as well as the global and local declarations are written in a C language subset \(^4\) provided by UPPAAL. The following content will define its elements and features.

\(^4\) UPPAAL C subset is defined in the help section of the UPPAAL toolkit and on the official UPPAAL web page: \url{http://www.uppaal.com/index.php?sida=217&rubrik=101}
Declarations in a C subset can be global or local. Declarable data types are: clocks, bounded integers, channels, arrays, records and types. Grammar definition of the declaration is shown in Figure 2.5.

```
Declarations ::= (VariableDecl | TypeDecl | Function | ChanPriority)*
VariableDecl ::= Type VariableID (',' VariableID)* ';'
VariableID ::= ID ArrayDecl* [ '=' Initialiser ]
Initialiser ::= Expression
   | '{' Initialiser (',' Initialiser)* '}'
TypeDecls ::= 'typedef' Type ID ArrayDecl* (',' ID ArrayDecl*)* ';'
```

**Figure 2.5: Formal definition of C subset declarations**

Data Types can be int (integer values), bool (boolean), chan (channel) and clock. Array and record types can be defined using these types and other user defined types.

<table>
<thead>
<tr>
<th>Description</th>
<th>Keyword</th>
<th>Value range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean</td>
<td>BOOL</td>
<td>False or true</td>
</tr>
<tr>
<td>Integer</td>
<td>INT</td>
<td>[-32768, 32767]</td>
</tr>
<tr>
<td>Channel</td>
<td>CHAN</td>
<td>Urgent, broadcast or binary</td>
</tr>
<tr>
<td>Clock</td>
<td>CLOCK</td>
<td>No info⁶</td>
</tr>
</tbody>
</table>

Boolean type can have one of two possible values - true or false. It also can be presented in the numerical form and in that case false is 0 and true is any non-zero value.

Integer data type can take a value from the range in the table. Any value out of the predefined range will cause a verification to abort.

Channels can be declared as urgent or broadcast. If those two words are omitted than it is a binary channel.

Clocks can take only integer values.

System definition provides a definition of the system model. The model consists of: one or more concurrent processes (templates), local and global variables, functions and channels. Templates without parameters are translated into exactly one process. If the template is parameterized, then the process is created for each set of the parameter combinations.

---


⁶ This information is not provided, however clocks are referring to integer values, so the range should be the same.
**Scalars** are integer-like elements with a limited number of operations. It means that scalars are unordered (model cannot distinguish between different orders). This feature at the end results in a faster verification and less memory being used.

**Functions** can be global or locally assigned to the template. Their grammar definition is:

*Function ::= Type ID '(' Parameters ')' Block*

As a continuation of the grammar for the function, other statements are defined in Figure 2.6.

```
Block ::= '{' Declarations Statement* '}'
Statement ::= Block
                | ';'
                | Expression ';'
                | ForLoop
                | Iteration
                | WhileLoop
                | DoWhileLoop
                | IfStatement
                | ReturnStatement
ForLoop ::= 'for' '(' Expression ';' Expression ';' Expression ') Statement
Iteration ::= 'for' '(' ID ':' Type ')' Statement
WhileLoop ::= 'while' '(' Expression ')' Statement
DoWhile ::= 'do' Statement 'while' '(' Expression ')' ';'
IfStatement ::= 'if' '(' Expression ')' Statement [ 'else' Statement ]
ReturnStatement ::= 'return' [ Expression ] ';
```

*Figure 2.6: Formal definition of C subset statements*

**Iterators**, in C subset, are used with keyword for. However, for could be used in two different ways - like a C for loop and like a Java iterator.

- C like for loop:
  *ForLoop ::= 'for' '(' Variable Init ';' Condition ';' Variable Update ')' Statement*

- Java like iterator
  *Iteration ::= 'for' '(' ID ':' Type ')' Statement*

Iterator will execute the *Statement* once for each value in *ID* of the type *Type* (can be array or a scalar).

**Operators** are constructs which behave like functions but generally have a different syntax or even semantics than the general functions. Table 2.8 contains all of the C subset operators.
### Table 2.8: C subset operators

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>()</td>
<td>Parenthesis alter the evaluation order</td>
</tr>
<tr>
<td>[]</td>
<td>Array lookup</td>
</tr>
<tr>
<td>.</td>
<td>Infix lookup operator to access process scope</td>
</tr>
<tr>
<td>!</td>
<td>Logical negation</td>
</tr>
<tr>
<td>++</td>
<td>Increment (can be used as both prefix and postfix operator)</td>
</tr>
<tr>
<td>--</td>
<td>Decrement (can be used as both prefix and --&gt; --postfix operator)</td>
</tr>
<tr>
<td>-</td>
<td>Integer subtraction (can also be used as unary negation)</td>
</tr>
<tr>
<td>+</td>
<td>Integer addition</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Less than or equal to</td>
</tr>
<tr>
<td>==</td>
<td>Equality operator</td>
</tr>
<tr>
<td>!=</td>
<td>Inequality operator</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Greater than or equal to</td>
</tr>
<tr>
<td>&gt;</td>
<td>Greater than</td>
</tr>
<tr>
<td>&amp;</td>
<td>Bitwise and</td>
</tr>
<tr>
<td>^</td>
<td>Bitwise xor</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>*</td>
<td>Integer multiplication</td>
</tr>
<tr>
<td>/</td>
<td>Integer division</td>
</tr>
<tr>
<td>%</td>
<td>Modulo</td>
</tr>
<tr>
<td>&lt;&lt;</td>
<td>Left bit shift</td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td>Right bit shift</td>
</tr>
<tr>
<td>&lt;?</td>
<td>Minimum</td>
</tr>
<tr>
<td>&gt;?</td>
<td>Maximum</td>
</tr>
<tr>
<td>&lt;</td>
<td>Less than</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>Logical and</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>?:</td>
<td>If–then–else operator</td>
</tr>
<tr>
<td>not</td>
<td>Logical negation</td>
</tr>
<tr>
<td>and</td>
<td>Logical and</td>
</tr>
<tr>
<td>or</td>
<td>Logical or</td>
</tr>
<tr>
<td>imply</td>
<td>Logical implication</td>
</tr>
<tr>
<td>forall</td>
<td>Forall quantifier</td>
</tr>
<tr>
<td>exists</td>
<td>Exists quantifier</td>
</tr>
</tbody>
</table>

Precedence rules for these operators are shown in Table 2.9.
### Table 2.9: Precedence order of C subset operators

<table>
<thead>
<tr>
<th>Associativity</th>
<th>Operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>left</td>
<td>() [] .</td>
</tr>
<tr>
<td>right</td>
<td>! ++ - - unary -</td>
</tr>
<tr>
<td>left</td>
<td>* / %</td>
</tr>
<tr>
<td>left</td>
<td>- +</td>
</tr>
<tr>
<td>left</td>
<td>&lt;&lt; &gt;&gt;</td>
</tr>
<tr>
<td>left</td>
<td>&lt;? ?&gt;</td>
</tr>
<tr>
<td>left</td>
<td>&lt;= &gt;= &gt;</td>
</tr>
<tr>
<td>left</td>
<td>== !=</td>
</tr>
<tr>
<td>left</td>
<td>&amp;</td>
</tr>
<tr>
<td>left</td>
<td>^</td>
</tr>
<tr>
<td>left</td>
<td></td>
</tr>
<tr>
<td>left</td>
<td>&amp;&amp;</td>
</tr>
<tr>
<td>left</td>
<td></td>
</tr>
<tr>
<td>right</td>
<td>?:</td>
</tr>
<tr>
<td>right</td>
<td>= := += -= *= /= %= &amp;=</td>
</tr>
<tr>
<td>right</td>
<td>not</td>
</tr>
<tr>
<td>left</td>
<td>and</td>
</tr>
<tr>
<td>left</td>
<td>or imply</td>
</tr>
<tr>
<td>left</td>
<td>forall exists</td>
</tr>
</tbody>
</table>

**Reserved keywords** as in ST are already assigned for the functionalities provided by language and cannot be used as identifiers. UPPAAL has the following keywords: `chan, clock, bool, int, commit, const, urgent, broadcast, init, process, state, guard, sync, assign, system, trans, deadlock, and, or, not, imply, true, false, for, forall, exists, while, do, if, else, return, typedef, struct, rate, before_update, after_update, meta, priority, progress, scalar, select, void, default`. The following words are reserved for the future updates: `switch, case, continue, and break.`

---

7 Operators are presented in the descending precedence order. Borders and brackets have the highest priority while `forall` and `exists` have the lowest.
2.4 Testing and logic coverage

Software testing, in one of its many definitions, is defined as: "a process, or a series of processes which are designed to make sure that a computer code does what it was designed to do and, conversely, that it does not do anything unintended"[17]. In order to achieve the proper accuracy of such claims there are several different approaches and test requirements. Different perspectives in the software artifacts are used as the criteria for the testing. However, in all of them the most fundamental testing unit is a test case. Test case is composed of several elements:

- test case values
- expected results
- prefix values and
- postfix values

Mostly all of them are necessary for a complete execution and evaluation of the tested software[16]. Test case values are input values which are necessary for a complete execution of the tested software. Expected result is a result produced by the test execution only in the case the program satisfies its behavior. Prefix value represents any input value that is needed in order to set the software into the appropriate state so it can receive test case values [16]. Postfix values represent input values that needs to be sent to the software after the test case values are sent [16].

According to [16], "Test requirement is a specific element of a software artifact that a test case must satisfy or cover". In order to systematically generate test requirements we also define coverage criterion. It represents a rule or set of rules that binds a test requirement with a test case.

If we declare a set of test requirements as TR for a coverage criterion C, then we state that test set T satisfies C if and only if for every test requirement tr in TR, at least one test t in T exists such that t satisfies tr [16]. If we want to measure coverage C and present it as a value we define it as a ratio between test requirements satisfied by the test case (T) and the size of test requirements TR. This value is called coverage level. In order to make the theory more specified we will explain logic coverage.

Logic coverage is a branch of software testing which tests criteria based on logical expressions in the code. Logical coverage is defined with its two main units - predicate and clause. A predicate is an expression which evaluates to a boolean value. It is important to mention that predicate is a topmost structure. If we occur the following expression: $((a < b) \lor C) \land b(x, y))$ we can derive several expressions that may evaluate to a boolean value. However, the whole expression represents a predicate. Predicate may contain: function calls (that evaluate to a boolean value), boolean variables and values and finally non-boolean variables or values which are compared with comparator operators. Comparator operators are $\{<,>,\leq,\geq,\neq,=\}$\(^8\). Internal structure of the predicate also consists of logical operators $\{\text{conjunction} \lor, \text{disjunction} \land, \}$

\(^8\) Comparator operators between ST and UPPAAL language subset are almost the same. The difference is observable in inequality. Unlike the mathematical operators $\{\leq,\geq\}$ they both use $\{\leq,\geq\}$ instead. For the inequality operator ST uses ‘<>’ while C subset uses ‘!’=’. For equality operator ST uses ‘=’ and C subset uses ‘==’.
negation ¬, implication →, exclusive implication ⊕ and equivalence ↔\). Each of them is described in the tables of operators for the given languages. These logical operators also define the basic difference between predicate and the clause.

A **clause** is an expression which evaluates to the boolean value and does not contain logical operators. In the previously given example: (((a < b) ∨ C) ∧ b(x, y)), clauses would be:

- (a < b)
- C
- b(x, y)

These two testing units can define many coverage criteria. We will define two or them: predicate coverage (PC) and the clause coverage (CC). If we define the following terms such as P represents a set of all predicates in the code and p represents each predicate where p ∈ P, the definition of the predicate coverage is:

"For each p ∈ P, TR contains two requirements\(^9\): p evaluates to true, and p evaluates to false" [16].

In order to achieve maximum predicate coverage of the example(((a < b) ∨ C) ∧ b(x, y)), we need at least 2 test cases:

- a = 5, b = 7, C = true, b(x, y) = true
- a = 5, b = 7, C = true, b(x, y) = false

By these two test cases we are not testing each clause. Therefore we declare C as a set of clauses in the predicates in P. For a unique predicate p ∈ P we define \(C_p = \{c|c \in p\}\). Then C also represents an union of the clauses in each predicate in P, that is:

\[ C = \bigcup_{p \in P} C_p \]

The definition of the clause coverage according to [16] is:

"For each c ∈ C, TR contains two requirements: c evaluates to true, and c evaluates to false."

In order to achieve maximum clause coverage of the example (((a < b) ∨ C) ∧ b(x, y)), we need at least 2 test cases:

- a = 5, b = 7, C = true, b(x, y) = true
- a = 7, b = 5, C = false, b(x, y) = false

In the following sections we will refer to the predicate coverage as the decision coverage and the clause coverage as the condition coverage. Maximum logic coverage will refer to the maximum amount of covered predicates and clauses during the testing.

---

\(^9\) The two requirements for each predicate and clause will be referred as obligations in the following sections of the thesis (defined in Section 4.8, Logic coverage analysis)
3. Method

Method is a term that represents a procedure or process for attaining an object as a systematic procedure, technique, or mode of inquiry (scientific in this case). Throughout this research we used two main high-level procedures (shown in Figure 3.1).

First one is a constant survey which is an important part of this thesis. Many of the techniques needed here were already achieved in some amount in [6,8,9,11]. It is necessary to address their functionality and usability for the final goal of this research.

More essential part of this research is an experimentation. It involves hypothesis formulation which will straightforwardly assert certain level of research goal. For example hypothesis (more of an assertion in this case) could be – for a specific ST semantic structure it is possible to achieve a second level of formalization [2] and maximum logic coverage test generation. Each experiment will test hypothesis and derive assertions about its feasibility, plausibility and other important scientific evaluation terms. It is important to point that ST lacks of the formal structure and therefore it will be important to assert the level and scope of each assertion. For that part, each assertion will be validated and tested again with a newly constituted experiment if necessary. Thus, this research could be categorized as a quantitative one.

Background survey is constantly affecting research goals, hypothesis formulation and therefore guides experimentation. This is highly important as some flaws of the ST language are constantly found [8,11] and we need to be sure that we avoid experimentation of those uncertain and not standardized structures from IEC 61131-3 [12].
4. Technical design

4.1 Overview of ST-UPPAAL transformation and test case generation

The defined transformation is a concept that enables cyclic execution analogy between ST program and its UPPAAL representation. It is mainly defined by observing rules defined in the standard and the behavior analysis of the transformed elements in the executable UPPAAL form. The following sections will give the explanations on how to transform certain group of ST programs into the UPPAAL model checker. These sections also explain the logic coverage analysis and the automated test case generation. Given that ST is a textual programming language, the transformation is in some parts defined as the textual processing set of rules. First, we will provide the overall diagram of the automated test case generation process (shown in Figure 4.1) and the ST implementation (shown in Figure 4.2), which will serve as the running example throughout the chapter.

![Transformation diagram](image)

**Figure 4.1: Transformation diagram**
4.1.1 ST code example

The following ST code example (shown in Figure 4.2) is provided by Bombardier Transportation AB in Västerås.

```
FUNCTION_BLOCK LATCH1_I
VAR_INPUT  (1)
   LOAD : BOOL;
   TLOAD : BOOL;
   RESET : BOOL;
   IN_1 : BOOL;
END_VAR

VAR_OUTPUT (2)
   OUT_1 : BOOL;
END_VAR

VAR (3)
   PREV_LOAD : BOOL;
   IS_LEAD_EDGE : BOOL;
END_VAR

(4)
   IS_LEAD_EDGE := (NOT PREV_LOAD) AND TLOAD;
   IF RESET THEN
      OUT_1 := 0;
   ELSIF LOAD OR (IS_LEAD_EDGE) THEN
      OUT_1 := IN_1;
   END_IF;
   PREV_LOAD := TLOAD;
END_FUNCTION_BLOCK (5)
```

Figure 4.2: ST code example 1

Function block, shown in Figure 4.1, serves for storing signal values according to the predefined occasions. We can differentiate four major code sections. From (1) to (2) we detect variable section for the input parameters. From (2) to (3) there is the variable section for the output parameters, and from (3) to (4) we detect variable section for local variables. At the end, from (4) to (5) this example contains the behavior of the function block it describes. The implementation consists of four inputs variables (LOAD, TLOAD, RESET and IN_1), one output variable (OUT_1) and two local variables (PREV_LOAD and IS_LEAD_EDGE). All of them are boolean variables. The behavior of the function block is very simple and it will be described in details during the transformation. The first goal is to separate specified sections, statements, declarations etc. Afterwards, we tend towards the straightforward implementation of the UPPAAL model which will include all of the ST elements with the highest possible analogy and
traceability. We need to detect what can be transformed in a template or a C subset data type, how to use UPPAAL non-deterministic features etc.

4.2 Global picture of the transformed model

The final product of the transformation is an UPPAAL model which has several templates and other UPPAAL elements. They are briefly described in this section. This model represents a general transformation entity. It means that each transformation consists of the same automata types and other UPPAAL units. Between different code transformations we can only differentiate particular variable declarations and code implementations, but the automata base is constant (Figure 4.3). Model consists of three main units:

- Control template
- Read inputs unit
- Function/FB unit

**Control template** is a supervising automata which handles various actions. Its main functionality is to correctly execute one program execution in the non-deterministic UPPAAL environment. Therefore, it is a trigger event for two main channels in the model (read and execute). When those two channels are triggered, they activate *read inputs unit* or *function/FB unit* respectively.

**Read inputs unit** contains the exact number of input templates as the number of input variables for all of the functions/function blocks in the transformed ST program. Their main purpose is to non-deterministically assign values to each input variable.

**Function/FB unit** consists of templates that implement the behavior of the transformed functions and function blocks respectively. However, the number of templates will always be *the number of transformed function/FBs + 1*. The added function represents the template which executes the logic coverage analysis by each execution cycle. It is always executed after the execution of all the behavior function/FB templates.
Regardless the specific ST implementation, we developed the control UPPAAL template which is initially same for all of the transformations. It is a base on which we build every ST transformation. Similar template was already used for the transformation of FBD programming language to UPPAAL model [15]. That is one of the benefits in this case, because we can transform several connected function blocks and execute them with one
control template even though some functions are written in FBD and others in ST language.

**Figure 4.4: Control template**

The template shown in Figure 4.4 is a main control unit of the model. It has 6 possible states: **Waiting, ReadInputs, ExecuteProgram, UpdateOutputs, KeepInputs** and **FailureState**. Template consists of two main activities which are implemented in form of channels - **read** and **execute**. Channel **read** is responsible for the redundant assignment of new values to the input variables of the transformed program. Channel **execute** is responsible for the execution of the function/FB templates (which implement the behavior of the specified functions and function block). One more important activity, which is not implemented as a channel, is an **update of output variables**. Now we will proceed with the explanation of the control template’s states.

**Waiting** state represents the initial location in this automata. It has one transition which initializes global system variables (**N** and **INc**).

- **N** is a variable that holds the information about function or FB which is being executed during the specific iteration (this iteration is executed in the transition **ExecuteProgram → UpdateOutputs**).
- **INc** is a variable which represents the input counter. Input counter is necessary because each input variable assignment is presented as a separate template in this model. UPPAAL non-deterministically assigns values to each input variable and this is controlled by **INc** variable (this iteration is executed in the transition **ReadInputs → ExecuteProgram**).

**ReadInputs** represents a state in which the control template supervises non-deterministic assignment of input variable values (shown in Figure 4.5). Every input in automata is created as the separate template and each of them can be started by synchronizing **channel read**. Depending on the boolean variable **keepInputs**, template can take one of two possible transitions which will restore the previously used inputs or
generate the new values. System variable \textit{keepInputs}\textsuperscript{10}\textsuperscript{10} is created for the purpose of achieving maximum logic coverage of the underlined program when we need to keep inputs of some function/FB. This is mainly used when we have several connected function blocks and some of them are timers [15].

\begin{figure}[h]
\centering
\includegraphics[width=0.8\textwidth]{image1.png}
\caption{Input assignment process of the control template}
\end{figure}

After the assignment of input parameters we may execute the behavior of the ST program. For that purpose model contains a state \textit{ExecuteProgram}. This state has only one outgoing transition which triggers the \textit{execute!} synchronization channel. This channel will trigger the execution of a function/FB’s behavior. If there are several functions/FBs in the execution queue, variable \textit{N} will control the iterations for each execution (Figure 4.6). It is possible because \textit{FunctionIndex} variable represents the number of functions and function blocks of the underlined ST program.

\begin{figure}[h]
\centering
\includegraphics[width=0.8\textwidth]{image2.png}
\caption{Execution of functions and FBs behavior}
\end{figure}

\textsuperscript{10} This approach is inherited from [15], however in the programs provided in this thesis it is always set to false.
Output variables are implemented as a part of the function’s/FB’s behavior. Each function/FB has its own set of output variables but they are all declared in the system’s global declaration field. This approach is justified\(^{11}\) by the cases when those variables are used in other functions or FBs. It is a common situation in ST programs to use output of one FB as the input of another FB. Therefore, after each transition with execute/synchronization, global outputs will be updated and the automata will be positioned in the UpdateOutputs state. It is important to mention that during the execution we also analyze the current logic coverage of the model. For each function/FB we have the information about current coverage of the program’s logic (predicate and clause coverage). This part will be thoroughly explained in the following sections. Based on the logic coverage information, automata can continue with one of two possible transitions (Figure 4.7). In the transition (UpdateOutputs $\rightarrow$ KeepInputs) where guard has value: $\text{keepInputs} == 0$, automata resets input variables and during the read inputs part of the automata they will be again non-deterministically assigned by the UPPAAL. The second transition is used if we need to keep current inputs in order to achieve maximum logic coverage.

![Diagram](image.png)

**Figure 4.7: UpdateOutputs $\rightarrow$ KeepInputs transition**

At the final part of the control template, automata takes the decision whether to continue executing another iteration of the template or to reach the final state - FailureState. This decision is based on the number of new predicate/clause obligations\(^{12}\) that are triggered during the previous iteration. The automata will be stopped if the keepInputs variable is set to false and the previous iteration did not trigger new logic coverage obligation. The automata will be continued if there are new triggered obligations from the previous iteration or the keepInputs variable is set to true (shown in Figure 4.8). By using reachability property, UPPAAL will continue executing the model from Waiting state to FailureState until the property is satisfied.

\(^{11}\) This approach is also justified by the test case generation process, which is derived from the UPPAAL’s state trace.

\(^{12}\) Obligation represents true or false evaluation of the triggered predicate/clause. This terminology is explained in Section 4.8, Logic coverage analysis.
4.4 Read inputs templates

The main purpose of this unit is to non-deterministically assign new values to the input variables of the functions and FBs being transformed. As previously stated, these assignments are triggered by the control template, more accurately they are triggered as the part of the channel synchronization called read. One example of the triggered template is shown in Figure 4.9.

In the scenario from the ST example 1, we created a template which has one initial state and two possible transitions. This is a consequence of the variable's data type. TLOAD is a boolean variable and therefore UPPAAL can assign only two possible values. In one transition, in the update label (colored blue in the Figure 4.9) we assign TLOAD a value true, and in the other one a value false. The synchronization label is defined as read? read channel and it is a connection to the triggering transition from the control template (read!). The guard label (INc == 2) controls proper cyclical value assignment of the input variables.
variables (initially controlled by control template). In *ST example 1*, there are three more input variables of the same boolean type. Each boolean input variable is transformed with the same concept (shown in Figure 4.10).

Based on these observations and the fact that boolean data type has only two possible values, the general definition of the input variable template for a boolean data type is:

![Figure 4.11: Boolean-type read template, definition](image)

In the Figure 4.11, \( x \) stands for any integer value greater than 0 and \( \text{bool	extunderscore variable} \) represents specified variable identifier of the boolean data type. \( \text{INC} \) variable represents a counter which tracks the order of the input assignments. As seen from the control template, it will iterate between \textit{ReadInputs} and \textit{ExecuteProgram} states as many times as there are input variables in the program. Counter \( \text{INC} \) is responsible for the straightforward execution of these iterations.

However, for the integer input variables we need a different approach compared to the boolean. For integer variables we need to define a range of values from which UPPAAL non-deterministically selects one. That part is implemented in the select label which is described before (Section 2.3.1).
In Figure 4.12, $z$ and $y$ stand for any integer value with a constraint $z \leq y$. Guard and synchronization labels are the same as in boolean template definition. The difference is observable in the select label which is defined as an integer variable with a range of possible values (from $z$ to $y$). In the update label we assign the value from the range $[z, y]$ to the global input variable (integer_variable). The difference in non-determinism between boolean and integer-type variables could be observed in the implementation. Reading of the boolean is made by using two transitions with different update labels, but it can be implemented with the same concept used for integer variables. In that case we would define the $[z, y]$ range as $[0, 1]$. This comes from the C UPPAL subset’s implementation of boolean ($true = 1$, $false = 0$).

If the specified ST program contains the input variable of TIME data type then we treat it same as integer variables. However it is important to address the integer range in those cases as the standard Timer FBs have strictly specified constant time values for inputs. The specific case when we generally do not create read inputs templates is when the variable is qualified with a qualifier - constant in the ST program.

### 4.5 Function/FB templates

The main purpose of the function/FB templates is to execute the behavior of the functions and FBs. This template also handles the update of output values and the logic coverage analysis. As it is previously mentioned, number of functions will always be for one greater than the number of functions and function blocks in the program. It is the case because we always execute one more function that analyses logic coverage after the behavior executions. In the ST example 1, only one function block is transformed, but there are two produced function templates (shown in Figure 4.13). First function (function_1) is a behavior implementation of the function block LATCH1_I. The second function (function_2) is a function that analyzes the logic coverage after the execution of the function_1. The order of the execution is defined by the variable $N$, similarly like in read inputs templates with INc variable. From the Figure 4.13 we can notice several local functions written in the update label of the transition (initialization, behavior, updateInputs, calculatePredicatesCount, calculateClausesCount). All of them are locally implemented in UPPAAL’s C subset.
Each function/FB is straightforwardly transformed into the predefined template type. The labels of these templates are structurally the same and can vary only in certain values of local function’s input parameters and the values of the counter variables in the guard \(N\). Therefore, we will define the general behavior template as:

![Behavior function definition](image)

**Figure 4.14: Behavior function definition**

In Figure 4.14, \(x\) stands for any integer value greater than 0. *Initialization* function sets the input *cross-reference* variables which are provided to the behavior function. We use *cross-reference* variables for the input and output variables because the PLC programs can contain global variables which are provided to several functions or function blocks as input variables. Standard [12] clearly defines that the changes of input variables inside the function/FB are forbidden. Also, the changes of the variables, made during the execution of the function blocks, can be globally manifested only after the execution of the whole function block.
The global definition of the function that calculates logic coverage is shown in Figure 4.15.

![Figure 4.15: Logic coverage function definition](image)

Similarly as in behavior functions, \( x \) stands for any integer value greater than 1 in this case. This is the rule because we need at least one executable function/FB in order to calculate the logic coverage. Every other UPPAAL label is constant among different ST program transformations. This template is triggered by the `execute!` channel. In the update label, this template executes the local functions `calculatePredicatesCount()` and `calculateClausesCount()`. Those functions calculate the number of triggered predicate and clause obligations. In the update label we increment the global function/FB counter \( N \) in order to continue the flow of the control template.

### 4.6 Global declarations

UPPAAL consists of global declaration space whose variables could be tracked during the execution and upon which we create test cases at the end. In the global declaration space of the transformed model we differentiate several declaration subspaces. All of them have their specified functionality and purpose. Therefore we can differentiate six of them:

- Automata variables (flow control variables),
- Input variables,
- Output variables,
- Cross-reference function variables,
- Cross-reference output variables,
- Code dependent initialized variables.

Automata variables are the variables that affect the flow of the automata. Without these variables UPPAAL would not be able to take transitions between states in the control and other specified templates. In this declaration subspace there are following variable declarations:

- Channels (`read` and `execute`)
- Logic coverage Integers: `newPredicatesCount`, `currentPredicatesCount`, `newClausesCount`, and `currentClausesCount`
- Control flow Integers (`N` and \( INc \)
Input variables are the variables specified within the `VAR_INPUT` section of the ST code. Analogically, output variables are declared in the variable section `VAR_OUTPUT`. All of these variables must be tracked during the execution, and based on these variables we derive test cases. That is the reason of their global declarative position in the transformed UPPAAL implementation. Two more sections which represent a cross-reference copy of the input and output variables are also implemented in the global declaration. This way we can easily track what are the values of these variables during the execution of the program and still not violate the standard’s rules described before.

Code dependent variables are variables that are derived from the program’s code analysis (textual analysis in this case). Those variables also serve for controlling the flow of automata. They are initialized with values we derive from the ST program. Those variables are mainly constant. They cannot be changed within the automata except in the case of array variables which reflect the logic coverage obligations. Code dependent variables are:

- `const int FunctionIndex` – number of functions/FBs incremented by two. This variable is always initialized this way because we have extra logic coverage function and because in this automata $N$ is initialized and reset with a value one (not a zero).
- `const int inputVariables` – number of input variables in the code incremented by one. Increment is made for the same reason as in `FunctionIndex`, as the counter $INc$ is initialized and reset with a value one.
- `const int numberOfPredicates` – number of predicate obligations\(^\text{13}\) in the code.
- `const int numberOfClauses` – number of clause obligations in the code.
- `bool predicates [numberOfPredicates]` – array whose elements represent each predicate obligation from the code. Initially they are set to 0, meaning – not triggered.
- `bool clauses [numberOfClauses]` – array whose elements represent each clause obligation from the code. Initially they are set to 0, meaning – not triggered.

In Figure 4.16, we show the global declaration implementation for the ST example 1.

\(^\text{13}\) Predicate and code obligations are defined in Section 4.8, Logic coverage analysis
Figure 4.16: Global declarations of ST example 1

The code shown in Figure 4.16 is implemented in UPPAAL C subset. As pointed previously, ST example 1 contains 4 input variables (LOAD, TLOAD, RESET and IN_1). Therefore, we write the same declarations in the UPPAAL with a difference initialization. All of them are initialized with the standard specified default value (false). Variable inputVariables is initialized with 5 (4 inputs + 1) because we already pointed that the counter initializations and resets (for N and INc) are set to 1. It is important to point out the usage of the META keyword. UPPAAL is treating variables as a part of the explorative state space unless we declare it as a META type. In that case it will not explore the state
space for these variables and therefore the execution will be shorter and will use less memory.

In order to derive Code dependent variables, we need to define a textual analysis\textsuperscript{14} of the ST program. First of all we will assume that the whole ST program could be converted to a String variable. After that we can do any kind of textual processing and modification such as: modification to lower case, concatenation, array conversions etc.

In order to get the FunctionIndex it is necessary to count the number of “END_FUNCTION” substring occurrences in the program string. This counting would only address two types of elements - functions and function blocks.

For the input variables it is necessary to make a substring between VAR_INPUT occurrence and the first occurrence of next END_VAR keyword. After that we need to count number of data type substring occurrences in the var_input-end_var substring. From the STexample1 we would count 4 keywords (BOOL data type substrings). If we have more functions or function blocks, we need to repeat the same process for all input variable sections.

Number of predicates is derived by detecting the number of boolean expressions multiplied by 2 (for each true and false obligation). Boolean expressions in ST are followed by IF, FOR, WHILE and UNTIL keywords. Another type we need to detect is an assignment of the boolean variable. Each statement of the form: \texttt{boolean_variable = boolean_expression}, contains a predicate (\texttt{boolean_expression}). The justification of this observation is shown with the following example. Each assignment can be analogically expressed in a form:

\begin{verbatim}
if (boolean_expression) {boolean_variable = true} else {boolean_variable = false}
\end{verbatim} 

Number of clauses is derived from the analysis of the logical operators (excluding negation) in the predicates of the code. The number of conditions is:

\begin{equation}
\text{number of predicates} + 2 \times (\text{number of logical operators in predicates})
\end{equation}

\textsuperscript{14}Textual analysis is here provided as the list of general guidelines for specified variables. If some of the future works address the software which will produce an UPPAAL xml file from the ST code, the methods will be language dependent but can be guided by the propositions made here.
4.6.1 Variable declarations

ST language differentiates several types of variable sections. They are defined by specified keywords – \texttt{VAR, VAR_INPUT, VAR_OUTPUT} etc. As shown in Table 2.4 they have different access rights. In order to meet those rights we define different transformation rules for their declaration. Each variable from \texttt{VAR_INPUT} section must be globally declared. Also they require a creation of cross-reference variables in the UPPAAL's global declaration section\textsuperscript{15}. \texttt{VAR_OUTPUT} variables follow the same logic. Variables from the \texttt{VAR_IN_OUT, VAR_EXTERNAL, VAR_GLOBAL} and \texttt{VAR_ACCESS} sections do not need to follow cross-reference concept because they have \textit{read} and \textit{write} access rights for both criteria (external and internal).

For each variable sections, transformation (ST $\rightarrow$ UPPAAL'C subset) of variable declarations, follows the next transformation rule:

\begin{align*}
\text{variable\_name : DATA\_TYPE; } & \rightarrow \text{data\_type variable\_name;}
\end{align*}

The order of declaration elements (data type and variable name) is inverted in a C subset. The colon character is removed and the data type is modified to the lower case characters. UPPAAL can recognize the following ST data types: \texttt{BOOLEAN} $\rightarrow$ \texttt{boolean, INT} $\rightarrow$ \texttt{int} and \texttt{TIME} $\rightarrow$ (\texttt{int or clock}). Regarding the variable qualifiers, UPPAAL contains a \textit{constant} qualifier analogical to the same qualifier in ST language. Each ST variable declared with this qualifier retains it in the UPPAAL form. However, other ST qualifiers must be implemented with a prevention accessibility analysis. This analysis should address the possible relations between entities and the qualified ST variables because UPPAAL cannot address this prevention.

\textbf{Limitations}

UPPAAL can cover the following integer type variations (\texttt{SINT, DINT, LINT, USINT, UINT, UDINT, ULINT}) but with a difference in the memory usage for their representation (Table 2.2). Depending of the integer range used in the program, execution of those transformations could lead to the out of bound errors and similar ones. We already stated that UPPAAL is designed as the timed automata of integer variables, therefore real numbers cannot be expressed in UPPAAL. Date types do not have the proper analogy as well. Although UPPAAL cannot cover specified data types, the majority of industrial programs and standard function blocks such as: \textit{Timer on delay}, \textit{Bistable blocks}, \textit{Counters} and \textit{Edge detectors} are transformable by the approach defined in this research.

\textsuperscript{15} With cross-reference variables we also prevent the possibility of the internal write violation shown in the Table 2.4.
4.7 Function C subset implementation

In order to get the executable form of function's/FB's behavior in UPPAL's C subset, we need to textually transform specified ST program. First of all we need to extract the part which represents a function or the function block whose behavior we want to transform. As the declaration of the input and output variables is implemented in the global declaration section, it is left to define the behavior transformation. This part will be explained through the ST example 1.

The function1 template from the automata (Figure 4.13) executes three local functions in the update label (initialization, behavior and updateOutputs). Initialization is a function that will assign the values to the cross-reference variables. These cross-reference variables will be given as the input parameters to the behavior function. At the end, the global output variables will be updated with the local output variables (modified during the behavior execution). The C subset implementation of the function/FB templates will always contain the following units in the exactly specified order:

1. Initialization function
2. UpdateOutputs function
3. Local variables declaration
4. Local outputs reference variables
5. Behavior function

The usage of the strict order is justified with the ST example 1. Initialization for the function1 is shown in Figure 4.17.

```c
void initialization () {
    cross_LOAD = LOAD;
    cross_TLOAD = TLOAD;
    cross_RESET = RESET;
    cross_IN_1 = IN_1;
}
```

**Figure 4.17: Initialization of the ST example 1**

Therefore we can form a definition of the initialization function shown in Figure 4.18

```c
void initialization () {
    cross_input_variable1 = input_variable1;
    cross_input_variable2 = input_variable2;
    ...
    cross_input_variableN = input_variableN;
}
```

**Figure 4.18: Initialization definition**
Output update for the ST example 1 is shown in Figure 4.19.

```c
void updateOutputs()
{
    OUT_1 = cross_OUT_1;
}
```

Figure 4.19: Update outputs function of ST example 1

It is important to distinguish two OUT_1 variables\(^\text{16}\) that we use in the transformed code. As this is the first occurrence of the OUT_1 variable in the C subset code, it will be treated as the global output variable, defined in the global declaration section. In Figure 4.20 we can see another OUT_1 variable which is a local, function1's variable. It is evaluated during the behavior execution.

![Figure 4.20: Update label of the function1 (from ST example 1)](image)

After the execution of the behavior function we assign the value of the local output variable (OUT_1) to the cross-reference output variable. This is implemented in the update label of the template so we could avoid the side effects into the behavior function. Eventually the definition of the update output function is shown in Figure 4.21.

```c
void updateOutputs()
{
    output1 = cross_output1;
    output2 = cross_output2;
    ...
    outputN = cross_outputN;
}
```

Figure 4.21: Update label of the function1 (from ST example 1)

---

\(^{16}\)Because of the potential invocation of local output variables we need to use the strictly defined order.
4.7.1 Local variables implementation

Local variables are declared as a part of the template’s declaration field in case they are a part of the function block (Figure 4.22). There is a very important reason why it is handled in this manner. It will be explained with the reference to the ST example 1. In that program we detect two local variables (PREV_LOAD and IS_LEAD_EDGE). In this example PREV_LOAD is a variable whose purpose is to memorize the value of the input variable TLOAD which is provided in the previous call of the function block. It means that PREV_LOAD must be implemented so its value is persistent upon two function block calls. If we declare PREV_LOAD in a behavior function, then it is reset to the false value upon each call. This occurs because false is C subset’s default value for the boolean data type declaration. Even in the standard [12] it is written: “The variables declared in the VAR ... END_VAR section persist from one call of the program or function block instance to another”.

```c
void behavior (input variables) {
    ...
}
```

Figure 4.22: Local variables declaration section (in Function Blocks)

However, there is a possibility that VAR declaration section is a part of the function, not the function block. In that case we declare those variables in the behavior function itself. This is the case because the standard [12] points: “Within functions the variables declared in this section do not persist from one call of the function to another”. The local variable treatment in functions is shown in Figure 4.23.

```c
void behavior (input variables) {
    data_type local_variable1;
    data_type local_variable2;
    ...
    data_type local_variableN;
}
```

Figure 4.23: Local variables declaration section (in Functions)
4.7.2 Behavior transformation

Transformation of the function's/FB's behavior can be treated as the textual transformation of the ST code into the C subset code. From the ST example 1, shown in Figure 4.2, we need to transform the part that is referring to the FB's behavior (shown in Figure 4.24) and implement it as the local function behavior() of the function1 template.

```
... IS_LEAD_EDGE := (NOT PREV_LOAD) AND TLOAD;

IF RESET THEN
  OUT_1 := 0;
ELSIF LOAD OR (IS_LEAD_EDGE) THEN
  OUT_1 := IN_1;
END_IF;

PREV_LOAD := TLOAD;
...
```

Figure 4.24: Behavior part of the ST example 1

Keywords, operators and statements transformation

The first step in this process is a lower case transformation of keywords from ST program. UPPAAL's C subset cannot recognize the uppercase keywords because it is case sensitive language. After this step, some keywords written in ST need to be changed as they have a different analogy in the C subset. In the example from Figure 4.24, and in all other cases of if statements, we need to make the following transformations:

- transform ELSIF keyword into the else if.
- transform THEN keyword into the left brace character '{' and
- transform END_IF keyword into the right brace character '}'.

In this moment, there is a possibility that we did not close opened braces. This occurs because ST has specific implementation of ELSIF statements.

By the standard's definition of IF statement (shown in the Table 2.6), the closing keyword is defined (END_IF), but we can also encounter several ELSIF and ELSE statements between IF and END_IF keywords. Those two expressions do not have a closing keyword. Taking into the consideration the nested cases as well, we will create an example which will serve us for the creation of general set of transformation rules for IF statement. It is shown in Figure 4.25 where we observe two codes. The first one is ST code example with nested IF, ELSIF and ELSE statements. The second one represents the code we get with a current set of transformation rules. The red braces represent the missing ones that we need to add in order to have an executable UPPAAL C code.

By observing the missing braces, we can derive a few very simple rules:

- Add right brace '}' before each else if keyword set
• Add right brace ‘}’ before each else keyword and add left brace ‘{’ after each else keyword

```
// ST CODE______________________________________________________________

IF (...) THEN
    Statements/Other expressions...
    IF (...) THEN
        Statements/Other expressions...
    ELSE
        Statements/Other expressions...
        IF (...) THEN
            Statements/Other expressions...
        ELSIF (...) THEN
            Statements/Other expressions...
        END_IF;
        END_IF
    ELSIF (...) THEN
        Statements/Other expressions...
    ELSE
        Statements/Other expressions...
    END_IF;

// TRANSFORMED BY THE RULES DERIVED SO FAR___________________________

if (...) {
    Statements/Other expressions...
    if (...) {
        Statements/Other expressions...
    }else {
        Statements/Other expressions...
        if (...) {
            Statements/Other expressions...
        } else if (...) {
            Statements/Other expressions...
        }
    }
}else if (...) {
    Statements/Other expressions...
}else {
    Statements/Other expressions...
}
```

Figure 4.25: Nested if statements (red – unwritten closed)

Finally, we get the executable if statement syntax.
The transformation of **assignment statements** requires modification of the ST assignment operator ':=:' to the character '==', which is an official assignment operator in C UPPAAL subset\(^\text{17}\).

Transformation of **comparison statements** and some assignment variations requires changing the ST operators to the analogical ones in C subset. In the Table 4.1 we defined the **operator transformation**.

<table>
<thead>
<tr>
<th>ST operators</th>
<th>Transformed to C subset operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>()</td>
<td>()</td>
</tr>
<tr>
<td>-</td>
<td>!</td>
</tr>
<tr>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>NOT</td>
<td>not</td>
</tr>
<tr>
<td>**</td>
<td>variable * variable</td>
</tr>
<tr>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>/</td>
<td>/</td>
</tr>
<tr>
<td>MOD</td>
<td>%</td>
</tr>
<tr>
<td>-</td>
<td>Subtract</td>
</tr>
<tr>
<td>&lt;,&gt;,&lt;&gt;=</td>
<td>&lt;,&gt;,&lt;&gt;=</td>
</tr>
<tr>
<td>=</td>
<td>==</td>
</tr>
<tr>
<td>&lt;&gt;</td>
<td>!=</td>
</tr>
<tr>
<td>&amp;</td>
<td>&amp;&amp;</td>
</tr>
<tr>
<td>AND</td>
<td>and</td>
</tr>
<tr>
<td>XOR</td>
<td>^</td>
</tr>
<tr>
<td>OR</td>
<td>or</td>
</tr>
</tbody>
</table>

Although the **case**\(^\text{18}\) statement will probably be included in some of the upcoming versions of UPPAAL, currently there is no proper analogy for that translation. One of the possible ways to translate the following structure\(^\text{19}\) is presented in the Figure 4.26.

---

\(^\text{17}\) Usage of characters ':=:' as the assignment also works in UPPAAL, but as it is not addressed in its language reference we decided to use the official assignment operator.

\(^\text{18}\) Besides case statement, the following keywords are also expected to be included in the next UPPAAL versions: switch, continue and break.

\(^\text{19}\) The case statement's execution flow is not thoroughly explained in the standard. Many languages use break after each case statement in order to exit the following condition evaluations. Here however we implemented the structure so that it is exited after the first condition is met.
Case statement in ST language can have an array of values as the criteria for the single execution (x and y in the Figure 4.26). In that case we create if statement with a condition which consists of disjunction array of equality comparisons with each value from the defined range (x = int_variable or y = int_variable). If the criteria in ST has only one value then we create only one equality comparison.

For statement in ST language has a significantly different form than the one we need to get in C subset. According the standard [12], its form is shown in Figure 4.27.

```
FOR int_variable := x TO y BY z DO
...
END_FOR
```

Figure 4.27: ST FOR statement (x,y and z are integer values)
The transformation of the braces and parenthesis is straightforward as in previous transformations. After keyword \textit{FOR} we need to add left parenthesis '(' before \textit{DO} we add the right parenthesis ')'. Instead of \textit{DO} we write the left brace '{' and instead of \textit{END\_FOR} we write the right brace '}'. With the predefined lower case and operator transformation we finally get the code shown in Figure 4.28.

\begin{figure}[h]
\centering
\begin{verbatim}
 for( int\_variable = x to y by z ){
 ...
 }
\end{verbatim}
\caption{ST FOR transformation}
\end{figure}

This is still not the finite executable \textit{for} statement. From three constructs that define \textit{for} statement, only the first one (assignment construct) will be transformed properly. In order to correctly transform the third construct (increment/decrement) we need to change keyword \textit{BY} with ';' and to write the construct\footnote{If the by keyword is omitted we use the default increment \textit{(int\_variable++)}. This is specified by the standard itself [12].} in the form shown in Figure 4.29.

\begin{figure}[h]
\centering
\begin{verbatim}
 for( int\_variable = x to y; int\_variable = int\_variable + (z) ){
 ...
 }
\end{verbatim}
\caption{ST FOR transformation (increment/decrement construct)}
\end{figure}

In order to be executable, the second construct (the guard) must be formed as a condition. This step needs the analysis of the values from first two constructs \{x, y\}. Standard [12] describes the following rule:

\textit{“The iteration is terminated when the value of the control variable is outside the range specified by the \textit{TO} construct.”}

In that case we need to define the range:
\begin{itemize}
\item \textit{int\_variable} \leq y, if \textit{x} \leq y
\item \textit{int\_variable} \geq y, if \textit{x} > y
\end{itemize}

Also, we change the \textit{TO} keyword with a semicolon character (;).
Finally, we get the executable for statement shown in Figure 4.30.

```c
// if x < y
for( int_variable = x ; int_variable <= y ; int_variable = int_variable + (z) ){
    ...
}

// if x > y
for( int_variable = x ; int_variable >= y ; int_variable = int_variable + (z) ){
    ...
}
```

Figure 4.30: Executable for statement

**While statement** has the following form defined by standard [12] which is shown in Figure 4.31.

```c
WHILE boolean_expression DO
...
END_WHILE
```

Figure 4.31: ST WHILE statement

The textual transformation requires:
- adding left parenthesis '(' after **WHILE** keyword,
- adding right parenthesis ') before **DO** keyword,
- changing **DO** keyword into the left brace "{",
- changing **END WHILE** keyword into the right brace "}"

With the previously defined set of transformations (lower case, operators etc.) the C subset statement is then executable and is shown in Figure 4.32

```c
while( boolean_expression ){
    ...
}
```

Figure 4.32: ST WHILE transformation
Repeat statement has the following form defined by standard [12], shown in Figure 4.33.

```
REPEAT ...
UNTIL boolean_expression
END_REPEAT
```

Figure 4.33: ST REPEAT statement

The C subset does not contain repeat keyword. However, the difference between this statement and the while statement is in the mandatory first execution (according to standard [12]). Because of that we can transform repeat statement with the following while statement shown in Figure 4.34.

```
bool isExecutedOnce = false;
while (!isExecutedOnce or boolean_expression) {
    ...
    isExecutedOnce = true;
}
```

Figure 4.34: ST REPEAT transformation

The textual transformation on the ST code (shown in Figure 4.33) requires:
- changing REPEAT keyword into the while keyword
- adding left parenthesis ‘(’ after while keyword,
- adding boolean_expression after left parenthesis
- adding right parenthesis ‘)’ after boolean_expression
- adding left brace ‘{’ after right parenthesis ‘}’
- changing END_REPEAT keyword into the right brace ”)"
- removing UNTIL keyword

The execution flow transformation requires the addition of the boolean variable (isExecutedOnce for example). This variable must detect whether the while statement is executed once and also must force the first execution of the inside statements. After the first execution, this variable is changed so it cannot affect the further executions and lets the predefined boolean_expression to control the execution flow.

As for the function and function block call statements, we need to point out an important rule. If function/FB is called inside the FB that we need to transform, the called function/FB must be declared in the global declaration space. The function/FB template remains the same except it lacks local declaration/implementation which is now positioned in the global declaration field. Selection, synchronization, guard and update remain the same (In case the function/FB is also called outside the FB).
4.8. Logic coverage analysis

Logic coverage analysis is integrated in the process of transformation. More accurately, the transformation includes several steps whose main purpose is to make logic coverage analysis possible. The created model is executed upon the temporal logic property which enables UPPAAL to create an execution path for the testing requirements. Several papers propose different approaches towards using model checking as testing mechanism. In [20], Amman identifies constraints of using temporal logic property as the logic coverage criteria, mainly because of the multiple necessary executions of the same code in order to reach some logic criteria (reflects to the several needed test cases for the achievement of maximum logic coverage). His model however does not use the concept proposed in [15] which overcomes this problem. In [15] this problem was solved by the implicit control loop (control template). Control loop has a reset transition which restores the program to its initial state and therefore does not require any changes on the predefined automata in order to achieve multiple runs and detect several needed test cases. This approach is also used in this thesis. In order to create straightforward logic analysis, Enoiu et al. [15] also proposed the following three steps:

1. annotate the conditions and decisions,
2. formulate a reachability property for logic coverage,
3. find a path from the initial state to the end of the program.

Those propositions are made for the FBD language, but with slight modifications they are also addressable for the ST language. The second and third step remain the same, but we need to change the annotation approach as FBD does not contain possibility to express predicates and clauses in the same manner as ST.

4.8.1 Predicate annotation and monitoring

Predicate annotation process consists of two steps:

1. detecting predicates in the code
2. creating side-effect free snippets of code which monitor whether the predicates are evaluated to true and false during the execution.

In ST, predicate can be any condition found in IF, ELSIF, WHILE and UNTIL statement. Also we explained that any boolean assignment of a type:

\[
boolean\_variable = boolean\_expression
\]

can be seen as:

\[
IF \ boolean\_expression \ THEN \ boolean\_variable := \text{TRUE};
ELSE \ boolean\_variable := \text{FALSE}; \ ENDIF;
\]

Therefore, we treat boolean expressions in those assignments as predicates. After detecting all the predicates in the code we need to create monitoring code statements which will not influence on the execution of the program behavior. However, they will monitor whether a certain predicate is evaluated to true and false (predicate coverage criteria). For this purpose we use a globally declared variable \(predicates[]\). This array contains an element for each predicate obligation in the transformed code. For ST example 1 it has ten elements. This variable is always initialized with all zero elements. If
a certain predicate evaluates to true or false during the execution, the corresponding element in the array will be modified to one.

```c
IS_LEAD_EDGE = (! PREV_LOAD) & TLOAD;
// auto generated code...
if((! PREV_LOAD) & TLOAD){
    predicates[0] = 1;
} else {
    predicates[1] = 1;
} // autogenEND
```

**Figure 4.35: Example for boolean assignment monitoring**

In the Figure 4.35 we show the generated monitoring code for the **boolean variable assignment**. Predicate \( p_1 \) in this example has the form: \( p_1 = (\neg \text{PREV\_LOAD} \& \& \text{TLOAD}) \). In order to achieve predicate coverage we declare two obligations for each predicate in the code:

\[
\begin{align*}
o_{1,i} &= p_i \\
o_{2,i} &= \neg p_i
\end{align*}
\]

Because of that, we create *if-else* statement which will check both obligations for the specified predicate \( p_1 \). If the first obligation is satisfied, the corresponding element (\( \text{predicates \[0\]} \)) in the \( \text{predicates} \) array will be set to one. If the second obligation is satisfied then the corresponding element (\( \text{predicates \[1\]} \)) will be set to one. Each boolean assignment is treated this way. The definition of the boolean assignment annotation is shown in Figure 4.36.

```c
boolean_variable = boolean_expression;
// auto generated code...
    if(boolean_expression){
        predicates[i] = 1; // o_{1,i} = p_i
    } else { 
        predicates[i+1] = 1; // o_{2,i} = \neg p_i 
    }
// autogenEND
```

**Figure 4.36: Definition of boolean assignment predicate annotation**

In Figure 4.36, \( i \) stands for any integer value which satisfies the condition: \( i+1 < \text{number \[0\] of Predicates} \).

For **if statements** we use the same logic but with a difference in the monitoring place for the second obligation.
In Figure 4.37 we show how to monitor and annotate predicates in if conditions. First obligation ($o_{1,1} = p_1$) is monitored and annotated immediately as we enter if statement's body (before any other statement). The second obligation ($o_{2,1} = \neg p_1$) is monitored and annotated in else/else if blocks. If there is no else block in the code after if statement, we add one for this purpose. In case that if statement has several following else if blocks, the second obligation must be monitored in each of the following else if blocks and final else block.

Considering **while statements**, we use the similar principle as for if statements. Immediately upon entering into the while loop we monitor and annotate first obligation of the predicate in while condition. Analogically, immediately after the while loop we annotate second obligation. This principle is shown in the Figure 4.38.
The same principle is also used for the **repeat statements** as we translated it using *while* statement. However, when we analyze the predicate in this case, it is important not to include boolean variable that is added for the mandatory first execution (*isExecutedOnce*).

**Case statements** follow the logics used for *if-then-else* statements as UPPAAL does not have executable *case* keyword and it is currently transformed in *if-then-else* form.

### 4.8.2 Clause annotation and monitoring

Clause annotation process consists of the same two steps as predicate annotation:

1. Detecting clauses in the code
2. Creating side-effect free snippets of code which monitor whether certain clauses are evaluated to true and false during the execution.

As we already defined the process for predicate detection, with this pre-knowledge we will continue the clause detection. As stated in the Background Section, clause is an expression which evaluates to the boolean value and does not contain logical operators. In order to derive clauses from the predicates it is necessary to split predicate in place of logic operators (and, or, &, |, ^). After that we need to monitor if each clause is evaluated to true and false (clause coverage criteria). Array variable which annotates this information is $clauses[i]$. Same as in predicates, it is initialized with all zero values and each clause obligation is assigned to the specified element in the array. In order to achieve clause coverage we declare two obligations for each clause in the code: $o_{1,i} = c_i$ and $o_{2,i} = \neg c_i$. General rules for clause annotation are the same as for the predicate annotation. It is necessary to inject the following code definition (Figure 2.39) after or instead of each predicate annotation ($p$) with a condition: $c|c \in p$. This is performed for each clause in the specified predicate.

```c
while(predicate){
    // auto generated code...
    predicates[i] = 1; // $o_{1,i} = p_i$
    // autogenEND
    Statements...
}
    // auto generated code...
    predicates[i+1] = 1; // $o_{2,i} = \neg p_i$
    // autogenEND
    Statements...
```

**Figure 4.38: Definition of while statement predicate annotation**
In that case the ST example from the Figure 4.35 would have the structure shown in Figure 4.40.

```plaintext
if (c){
    clauses[i] = 1; // o_{1,i} = c_i
} else{
    clauses[i+1] = 1; // o_{2,i} = \neg c_i
}
```

**Figure 4.39: Definition of clause annotation**

In Figure 4.40 we injected the annotation code for each clause from the predicate \( p \):
\[
p = (\neg \text{PREV\_LOAD}) \&\& \text{TLOAD}
\]
Clauses of this predicate are: \( c_1 = \neg \text{PREV\_LOAD} \) and \( c_2 = \text{TLOAD} \). They are annotated immediately after the predicate annotation. The same principle is used for other statements (if, while, case, repeat and for).

4.8.3 Reachability property

Reachability property is a type of the temporal logic property that questions whether a given formula, \( \varphi \), can be satisfied [24]. It can be formulated as: Is there any path in the timed automata from the initial state such that \( \varphi \) is satisfied along the path.

At this level of transformation we have enough information in order to get the exact logic coverage levels. Those values are presented as a ratio between test requirements satisfied by the test suite (T) and the size of test requirements TR. In case of predicate coverage, the number of satisfied requirements is a number of array elements (in `predicate[]`) which are set to one (meaning reached predicate obligation).
This value is evaluated and assigned to the variable \textit{currentPredicatesCount} by each execution of the logic coverage analysis template (shown in Figure 4.15). The size of test requirements is a number of predicate obligations in the code (\textit{numberOfPredicates}). Finally the predicate coverage level (\textit{pcl}) is:

\[ pcl = \frac{\text{currentPredicatesCount}}{\text{numberOfPredicates}} \]

The same analogy stands for the clause coverage level (\textit{ccl}):

\[ ccl = \frac{\text{currentClausesCount}}{\text{numberOfClauses}} \]

Therefore, reachability property can be written in several forms. With newest UPPAAL extensions [21] we can use \textit{sup} query which will return the highest possible value for a specified variable, clock etc. In this case we can create following two queries:

- \textit{sup: CurrentPredicatesCount}
- \textit{sup: CurrentClausesCount}

The result of this query will be presented in a following form:

\[ \text{currentPredicatesCount} \leq x \text{ and } \text{currentClausesCount} \leq y \]

In the preceding expression, \textit{x} and \textit{y} could be any integer value. We achieved 100\% predicate and clause coverage if the following statement holds:

\[ x = \text{numberOfPredicates} \text{ and } y = \text{numberOfClauses} \]

For the better performances of the model checker we use the following queries:

- \( E<>\text{plc.UpdateOutputs and } (\text{predicates}[0] + \text{predicates}[1]+...+\text{predicates}[p-1]) = p \)
- \( E<>\text{plc.UpdateOutputs and } (\text{clauses}[0] + \text{clauses}[1]+...+\text{clauses}[c-1]) = c \)

Where \textit{p} stands for number of predicate obligations in the code and \textit{c} stands for number of clause obligations in the code. The reachability property of the type \( E <> \phi \) verifies if there exists a maximal path in the timed automata for which \( \phi \) is always true.
4.9 Test case generation

Deriving the test cases from the UPPAAL trace is the final step of the automated generation test case generation. The transformation defined so far has several aims and one of them is to provide execution analogy between ST program and the transformed UPPAAL model. Also, the important aim is to ensure traceable testing. This means that although the transformed code can be only executed in the model checker, the results of the executions can be presented in the form of test cases. UPPAAL model checker does not contain feature for exact test case generation. However, this is solved in paper proposed by Enoiu et al. [15]. They used implicit cycle execution in control template of the transformed model. When the execution is treated this way, the trace can be presented in a straightforward test case form. This is possible because each cycle run in control template reflects to the exactly one test case, as shown in Figure 4.41.

![Figure 4.41: Test case iteration in control template (ST example)](image)

In Figure 4.41 we show four specified moments in the test case derivation. First moment shows inputs that are not set. Note that they always contain some value (default one from the standard or from the previous iteration) however we describe them as not set because they need to be non-deterministically assigned for the following test case. Second moment comes after the assignment of each input. In this moment each of four inputs, from the LATCH 1_I function block, has false value. Third moment comes after the execution of the LATCH1_I behavior and now we get the output (OUT_1) which corresponds to the execution of the assigned inputs. After this moment we can generate a test case. Afterwards, in moment 4, we reset the inputs (at least for the test case purpose) and return to the new non-deterministic assignment of the inputs in order to generate a new test case.

Although we neglected synchronization and some other transitions in the Figure 4.41, we tended to express the basic set of states needed for the test case generation.
In general case UPPAAL gives a diagnostic trace which has the following form shown in Figure 4.42.

More formally, the diagnostic trace can be written in a form [15]:

\[(sys_0) \xrightarrow{a_1} (sys_1) \xrightarrow{a_2} ... \xrightarrow{a_n} (sys_n)\]

In this form, \((sys_k)\) stands for any single state in the transformed model and \(a_k\) stands for any transition, internal delay or a synchronization etc. Test case iterations are separated by each reset transition (when \(a_k = (KeepInputs \rightarrow ReadInputs)\)). Test case can be generated by reading the global input and output variables before the execution of the \(UpdateOutputs \rightarrow KeepInputs\) transition (it means that the test case iteration reached \(UpdateOutputs\) after the execution of each function/FB template).

In order to show test cases in a straightforward form (only inputs and outputs) we use the toolbox presented in [15] which analyses UPPAAL diagnostic trace (Figure 2.42) and derives different types of test suites\(^{21}\). This toolbox can derive test suites by selecting certain coverage criteria (predicate and clause), trace (shortest, fastest, some) and search order (breadth-first, depth-first and random-depth first). It is important to mention that this toolbox cannot affect the specific transformation nor the UPPAAL code, it only analyses the trace and visualizes the results.

The following section shows the test suites derived with a toolbox on a LATCH1_I transformed model and other achieved results.

---

\(^{21}\) Test suite is a set of test cases.
5. Results and evaluation

Results from the LATCH1_I test case generation are presented in the following figures. The generation of test cases is conducted using CompleteTest\textsuperscript{22} tool [15]. Each result is obtained with a specific set of properties defined for the generation. The properties are divided in three categories: logic coverage criteria, search order and trace type. Search order (SO) defines the state exploration order and contains the following choices:

- Breadth first
- Depth first
- Random depth first

Breadth first method searches the state space in breadth first search order. If it is necessary to search the complete state space, this method is the most efficient. Also if we want to generate shortest and fastest traces (test cases in this case), this is the best option according to the official UPPAAL document\textsuperscript{23}.

Depth first method searches the state space in depth first search order. It is efficient if a counter example or a witnessing trace are expected to exist. In a general case it is more efficient than the breadth first but not for generating of shortest and fastest traces.

Random depth first searches the state space in randomized depth first search order. It is the best method if a counter example or witnessing trace are expected to exist. Randomization affects traces to vary from run to run. However, for a shortest and fastest traces it is not a very efficient method.

Diagnostic trace controls whether a counter-example or witnessing trace (if there is one) should be generated during verification and the following choices are available:

- Some - generates a diagnostic trace
- Shortest - generates a trace with smallest number of transitions
- Fastest - generates a trace with the shortest accumulated time delay

As shown in Figure 5.1, using our approach, we generated a test suite that achieves 100% predicate and 100% clause coverage for the LATCH1_I example. Trace property used for deriving this test suite is shortest and search order property is breadth first.

\textsuperscript{22} Complete Test tool is available for download at www.completetest.org web page.

\textsuperscript{23} This is the official recommendation provided by UPPAAL model checker in its help file and also in the following web page http://www.uppaal.com/
As shown in Figure 5.2 we generated a test suite which achieves 100% predicate and 100% clause coverage for the LATCH1_I example. Trace property used for deriving this test suite is *fastest* and search order property is *depth first*.

As shown in Figure 5.3 we generated a test suite which achieves 100% predicate and 100% clause coverage for the LATCH1_I example. Trace property used for deriving this test suite is *some* and search order property is *random depth first*.

In Figure 5.4 we annotate the LATCH1_I example with predicate and clause identifiers in order to analyze and validate the execution, logic and clause coverage from the Test suite 1. Predicate annotations are colored red and clause annotations are colored blue. In Table 5.1 we show each variable from the LATCH1_I function block (input, local and output). Local variables PREV_LOAD and IS_LEAD_EDGE have default false values before the execution of first test case inputs. This value is standard defined [12] for the boolean data type. For each test case we show triggered predicate and clause obligations. Note that predicate annotations are presented in the form (Pi) where i presents the ordinal number of predicate in the code. The same analogy stands for clauses (Ci). If the predicate/clause is triggered with true value, the annotation in the predicate/clause obligation column will be - Pi or Ci respectively. If we triggered the negative obligation then we annotate it with ¬Pi and ¬Ci respectively.
IS_LEAD_EDGE := (NOT PREV_LOAD) c1 AND TLOAD c2 P1;

IF RESET c3 p2 THEN
    OUT_1 := 0;
ELSIF LOAD c4 OR (IS_LEAD_EDGE) c5 p3 THEN
    OUT_1 := IN_1 c6 p4;
END_IF;

PREV_LOAD := TLOAD c7 p5;

Figure 5.4: LATCH1_I logic elements annotation

Executing the transformed code with test cases from Test Suite 1 will lead to the logic coverage steps shown in Table 5.1.

Table 5.1 Logic coverage analysis of Test suite 1

<table>
<thead>
<tr>
<th>LOAD</th>
<th>TLOAD</th>
<th>RESET</th>
<th>PREV_LOAD</th>
<th>IS_LEAD_EDGE</th>
<th>Predicate obligations</th>
<th>Clause obligations</th>
<th>OUT_1</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>¬P1, ¬P2, ¬P3, ¬P5</td>
<td>C1, ¬C2, ¬C3, ¬C4, ¬C5, ¬C7</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>¬P1, P2, ¬P5</td>
<td>C1, ¬C2, C3, ¬C7</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>P1, ¬P2, P3, ¬P4, P5</td>
<td>C1, C2, ¬C3, ¬C4, C5, ¬C6, C7</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>¬P1, ¬P2, P3, P4, ¬P5</td>
<td>¬C1, ¬C2, ¬C3, C4, ¬C5, C6, ¬C7</td>
<td>T</td>
</tr>
</tbody>
</table>

Reached coverage (%): 100% 100%
5.1 Experimental comparison

The most relevant related work to this research is conducted in [15] and it addressed FBD automatic test generation with UPPAAL model checker. In order to compare results we get with that transformation and the one which is conducted in this research we found a specific FBD function block which can be wrote in ST as well. As pointed in the standard [12], ST and FBD share analogy but ST has certain extensions that cannot be covered by FBD. We will conduct transformations of these two programs (Figure 5.5 and 5.6) with the defined transformation rules and compare results. In our evaluation we used programs provided by Bombardier Transportation AB in Västerås. The Fan Control program contains of one integer input (DBC_PV_X_CoStep), ten constant integer variables (P_Fan1Lo_1=3, P_Fan1Lo_2=7, P_Fan1Hi=8, P_Fan2Lo_1=4, P_Fan2Lo_2=6, P_Fan2Hi_1=7, P_Fan2Hi_2=8, P_Fan3Lo=5, P_Fan3Hi_1=6, P_Fan3Hi_2=8), and six boolean output variables (DBC_PV_C_Fan1Lo, DBC_PV_C_Fan1Hi, DBC_PV_C_Fan2Lo, DBC_PV_C_Fan2Hi, DBC_PV_C_Fan3Lo, DBC_PV_C_Fan3Hi).

Figure 5.5: Fan_Control FBD program
FUNCTION_BLOCK FAN_CONTROL
VAR_INPUT
  DBC_PV_CoStep : INT;
END_VAR
VAR_OUTPUT
  DBC_PV_C_Fan1Lo : BOOL;
  DBC_PV_C_Fan1Hi : BOOL;
  DBC_PV_C_Fan2Lo : BOOL;
  DBC_PV_C_Fan2Hi : BOOL;
  DBC_PV_C_Fan3Lo : BOOL;
  DBC_PV_C_Fan3Hi : BOOL;
END_VAR
VAR CONSTANT
  P_Fan1Lo_1 : INT := 3;
  P_Fan1Lo_2 : INT := 7;
  P_Fan1Hi : INT := 8;
  P_Fan2Lo_1 : INT := 4;
  P_Fan2Lo_2 : INT := 6;
  P_Fan2Hi_1 : INT := 7;
  P_Fan2Hi_2 : INT := 8;
  P_Fan3Lo : INT := 5;
  P_Fan3Hi_1 : INT := 6;
  P_Fan3Hi_2 : INT := 8;
END_VAR
IF (DBC_PV_CoStep >= P_Fan1Lo_1) & (DBC_PV_CoStep <= P_Fan1Lo_2) THEN
  DBC_PV_C_Fan1Lo := TRUE;
ELSE DBC_PV_C_Fan1Lo := FALSE; END_IF;
IF (DBC_PV_CoStep = P_Fan1Hi) THEN
  DBC_PV_C_Fan1Hi := TRUE;
ELSE DBC_PV_C_Fan1Hi := FALSE; END_IF;
IF (DBC_PV_CoStep >= P_Fan2Lo_1) & (DBC_PV_CoStep <= P_Fan2Lo_2) THEN
  DBC_PV_C_Fan2Lo := TRUE;
ELSE DBC_PV_C_Fan2Lo := FALSE; END_IF;
IF (DBC_PV_CoStep >= P_Fan2Hi_1) & (DBC_PV_CoStep <= P_Fan2Hi_2) THEN
  DBC_PV_C_Fan2Hi := TRUE;
ELSE DBC_PV_C_Fan2Hi := FALSE; END_IF;
IF (DBC_PV_CoStep = P_Fan3Lo) THEN
  DBC_PV_C_Fan3Lo := TRUE;
ELSE DBC_PV_C_Fan3Lo := FALSE; END_IF;
IF (DBC_PV_CoStep >= P_Fan3Hi_1) & (DBC_PV_CoStep <= P_Fan3Hi_2) THEN
  DBC_PV_C_Fan3Hi := TRUE;
ELSE DBC_PV_C_Fan3Hi := FALSE; END_IF;
END_FUNCTION_BLOCK

Figure 5.6: Fan_Control ST program
Using the transformation shown by Enoiu et al. [15], we generated a test case shown in Figure 5.7. It reaches 100% decision and condition coverage in this case (analogically to predicates and clauses). The search order used in this generation is breadth first and the test suite is same for all of the trace options (shortest, some, fastest). During the performance testing we used same performance environment for both models.

![Figure 5.7: Fan_Control FBD test suite 1](image)

Test suite we get from the ST transformation (proposed in this research) of Fan_Control ST program is shown in Figure 5.8. It reaches 100% predicate and clause coverage in this case. The search order used in this generation is breadth first and the test suite is same for all of the trace options (shortest, some, fastest).

![Figure 5.8: Fan_Control ST transformation test suite 1](image)

Finally, both test suites have the same test cases. The order of them in this example is of no relevance (because although it is FBD, it does not contain its own mutable state). Regarding the performance of executed tests, we conducted 31 test generations per each and derived the results shown in the Table 5.2.

---

24 This is the case because all the local variables in Fan_Control FB are local.
Table 5.2 Performance analysis for breadth first SO with shortest trace

<table>
<thead>
<tr>
<th>ST transformation</th>
<th>Measured property:</th>
<th>Time used (s)</th>
<th>Memory used (KB)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Performance type:</td>
<td>Verification</td>
<td>Kernel</td>
</tr>
<tr>
<td></td>
<td>Mean:</td>
<td>0.052</td>
<td>0.002</td>
</tr>
<tr>
<td></td>
<td>Standard deviation:</td>
<td>0.009</td>
<td>0.005</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>FBD transformation</th>
<th>Measured property:</th>
<th>Time used (s)</th>
<th>Memory used (KB)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Performance type:</td>
<td>Verification</td>
<td>Kernel</td>
</tr>
<tr>
<td></td>
<td>Mean:</td>
<td>0.319</td>
<td>0.025</td>
</tr>
<tr>
<td></td>
<td>Standard deviation:</td>
<td>0.025</td>
<td>0.020</td>
</tr>
</tbody>
</table>

For the depth first search order and trace set to some, both transformation achieve 100% of logic criteria. The generated test cases are shown in Figure 5.9.

Figure 5.9: Fan_Control FBD test suite 2

Figure 5.10: Fan_Control ST test suite 2
Performance analysis in this case (search order: depth first, trace: some) is shown in Table 5.3.

Table 5.3 Performance analysis for depth first SO with trace set to some

<table>
<thead>
<tr>
<th>ST transformation</th>
<th>Measured property:</th>
<th>Time used (s)</th>
<th>Memory used (KB)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Performance type:</td>
<td>Verification</td>
<td>Kernel</td>
</tr>
<tr>
<td>Performance type:</td>
<td>Mean:</td>
<td>0,005</td>
<td>&lt;0.001</td>
</tr>
<tr>
<td>Standard deviation:</td>
<td>0,007</td>
<td>&lt;0.001</td>
<td>0,005</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>FBD transformation</th>
<th>Measured property:</th>
<th>Time used (s)</th>
<th>Memory used (KB)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Performance type:</td>
<td>Verification</td>
<td>Kernel</td>
</tr>
<tr>
<td>Performance type:</td>
<td>Mean:</td>
<td>0,011</td>
<td>&lt;0.001</td>
</tr>
<tr>
<td>Standard deviation:</td>
<td>0,007</td>
<td>&lt;0.001</td>
<td>0,003</td>
</tr>
</tbody>
</table>

For the random first search order and trace set to some, both transformation cover 100% of logic criteria. Generated test cases differ by each test case generation. However, we showed one instance for each transformation (Figure 5.11 and Figure 5.12).

Figure 5.11: Fan_Control FBD test suite 3

Figure 5.12: Fan_Control FBD test suite 3
Performance analysis in this case (search order: *random depth first*, trace: *some*) is shown in Table 5.4

<table>
<thead>
<tr>
<th>ST transformation</th>
<th>Measured property:</th>
<th>Time used (s)</th>
<th>Memory used (KB)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Performance type:</td>
<td>Verification</td>
<td>Kernel</td>
</tr>
<tr>
<td>Mean</td>
<td>0.005</td>
<td>&lt;0.001</td>
<td>0.003</td>
</tr>
<tr>
<td>Standard deviation</td>
<td>0.007</td>
<td>&lt;0.001</td>
<td>0.005</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>FBD transformation</th>
<th>Measured property:</th>
<th>Time used (s)</th>
<th>Memory used (KB)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Performance type:</td>
<td>Verification</td>
<td>Kernel</td>
</tr>
<tr>
<td>Mean</td>
<td>0.003</td>
<td>&lt;0.001</td>
<td>0.004</td>
</tr>
<tr>
<td>Standard deviation</td>
<td>0.010</td>
<td>&lt;0.001</td>
<td>0.009</td>
</tr>
</tbody>
</table>

5.2 Related work discussion

The transformation defined in this research is built on the model-checking approach for software testing (presented in [20]). We used the implicit control loop from [15] which overcomes issues for model checking verification stated in [20]. By using implicit control loop, we are able to use reachability property of model checker as the stopping logic coverage argument. The model proposed in [15] is adapted to the specific purpose of the ST transformation and test case generation. Because of the different abstraction levels of ST and FBD (which is used in [15]), we redefined some of the model checking properties such as: output update mechanism, annotation of logic criteria and logic criteria elements. In the previously conducted analysis of the test generation performance we can see that for the shortest trace generation we get better results in terms of model checking performance by using the transformation defined in this thesis. Verification time is significantly improved for the breadth first search order (0.052s compared to 0.319s for the FBD). Resident memory usage is also improved (8975 KB compared to 17684 KB for the FBD). Virtual memory analysis provided following values (29427 KB compared to 41384 KB for the FBD). However analysis for this category has a large standard deviation and it must be derived from the larger number of executions in order to be representative (Table 5.2). The same holds for the *depth first* performance analysis (0.005s compared to 0.011s). The model needs less time in order to retrieve test cases but the memory analysis is not representative (Table 5.3). When it comes to the *random depth first* search order results are almost the same (Table 5.4).

In order to see one of the reasons of improvement we will present number of stored states per each transformation. This is done only for the test suite generations which always result with the constant number of stored and explored states for the specified Fan_Control example. The results are shown in Table 5.5.
This significant improvement comes primarily from the difference in the logic coverage annotation approach, which is a consequence of abstraction differences between ST and FBD. Transformation proposed in [15] uses more transitions and timed automata in the UPPAAL network which executes the specified FBD diagram. The transformation proposed in [15] requires 14 different function templates in network of timed automata for the Fan_Control example (Figure 5.5). For the same example, the transformation proposed in this research requires only one function block for translation and therefore there are less timed automata for the function/FB section of the transformation. The test generation results are the same, but the performance is much improved for breadth and depth first search order. It is especially improved for the generation of the shortest traces which are highly demanded in software testing nowadays. In order to achieve straightforward transformation we used the concept of different element treatment proposed in [22, 23]. Some of the formalization techniques proposed in [23] don't cover test case generation and maximum logic coverage which we achieved in this research.
6. Conclusion

In this research we defined a set of rules for transforming ST programs into UPPAAL input model for the purpose of automatically generating test cases in order to satisfy logic coverage criteria. Results show that for certain type of ST programs, this approach produces expected and correct test cases. By annotating logic elements, defined in the transformation, we are able to achieve logic-based testing. Because of the UPPAAL’s C subset we also maintain the analogy between ST and UPPAAL to some extent. We point the term to some extent mainly because UPPAAL model checker does not contain possibility of computing with some data types which are defined in ST language (such as: Real, String, Date types etc.). Some of these data types can be transformed by using specified representations (presented in Section 6.1), but real numbers cannot be transformed because UPPAAL is defined as the timed automata network of integer clocks. However, the majority of industrial programs and standard function blocks such as: Timer on delay, Bistable blocks, Counters and Edge detectors are transformable by the approach defined in this research. It covers a majority of program spectrum used in industrial environment. Therefore, we will respond to the RQ2 and conclude that according to [5] this transformation can cover the second level of ST formalization – “Formalization of complete program”, assuming that the underlined ST program does not contain any of the elements that cannot be covered by UPPAAL transformation. This part is explained throughout the Section 4 (Technical Design) and therefore responds to the RQ1. Finally, we conducted several test case generations in order to validate the approach. For the specified programs (Figure 4.1 and Figure 5.6) it generated test cases which achieve maximum logic coverage. Also, we compared the test cases generated by the transformation proposed in this research with test cases generated for FBD diagrams, using the transformation shown by Enoiu et al. [15]. This comparison provided results that show performance improvements in test case generation (Table 5.2). It is especially evident when it comes to the generation of the minimal number of test cases that satisfy maximum logic coverage. Compared to our approach we also showed that the specified FBD transformation [15] generates larger number of states for constant test suite generations\textsuperscript{25} (Table 5.3). Therefore we can conclude that FBD diagrams should be translated to the ST language, if such translation is possible and can lead to the reduced number of function/FB timed automata in the final formalized model. With this translation we reduce the number of transitions and stored states which leads to the performance improvements in execution time and memory consumption.

6.1 Future work

Future work should address enlargement of the transformable ST data types in UPPAAL. String can be presented as the array of integer elements where each integer presents certain character from ASCII character set. Also, certain types of timed automata could enlarge range of integer values in order to satisfy long integer data type and similar extendable ones (unsigned integers etc.). For certain data types such as integer, clocks etc. there is a space for defining static program analysis methods which will decrease the

\textsuperscript{25} Constant test suite generation refers to breadth first and depth first search order generation. Those two search orders will always generate same test suite in UPPAAL for the underlined code.
input space for test case generation. Derivation of such methods can hugely improve the performance and probability for finding maximum logic coverage of underlined ST programs.
References:


