

MASTER'S THESIS

Happy-flow verification of Cyber-Physical Systems

Ketelaar, J.

Award date:
2022

[Link to publication](#)

General rights

Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain.
- You may freely distribute the URL identifying the publication in the public portal.

Take down policy

If you believe that this document breaches copyright please contact us at:

pure-support@ou.nl

providing details and we will investigate your claim.

Downloaded from <https://research.ou.nl/> on date: 19. Nov. 2022

Open Universiteit
www.ou.nl



HAPPY-FLOW VERIFICATION OF CYBER-PHYSICAL SYSTEMS

by

Jildert Ketelaar, MSc

in partial fulfillment of the requirements for the degree of

Master of Science
in Software Engineering

at the Open University of the Netherlands, Faculty of Science
Master's Programme in Software Engineering
to be defended publicly on Friday March 25, 2022 at 11:00 AM.

Student number:

Course code: IM9906

Thesis committee: Dr. Stefano Schivo (chairman/supervisor), Open University
Dr. Freek Verbeek (secondary supervisor), Open University
Dr. Jacques Verriet, ESI (TNO)

CONTENTS

Summary	iv
1 Introduction	1
1.1 Conquer the state space explosion	1
1.2 System under test: High speed Die Attach machine.	2
1.3 Goal of the research.	2
1.4 Research context.	3
1.5 Structure of the report	4
2 Background	5
2.1 Formal verification	5
2.1.1 Formal specification	5
2.1.2 Formal modelling.	6
2.1.3 Model checking	7
2.2 Uppaal.	7
2.2.1 Concepts & Syntax	8
2.2.2 Model verification	9
2.3 Source code to model translation.	9
2.4 Model Driven Engineering.	11
3 Related Work	12
3.1 Model checking of source code	12
3.2 Program Slicing	14
3.3 Research contribution	14
4 Method	15
4.1 Research questions	15
4.2 Research method	16
5 What to verify on the ‘Happy Flow’?	17
5.1 Definition of happy flow	17
5.2 Properties of interest	17
5.2.1 Deadlock & Livelock	17
5.2.2 Reachability properties	18
5.2.3 Safety properties	18
5.2.4 Liveness properties	19
5.2.5 Sequence validation	19
5.2.6 Cycle time analysis	19
6 ‘Happy Flow’ extraction	20
6.1 Determine the happy flow	20
6.2 Instrumentation	21
6.3 Executing the happy flow	22

6.4	Processing coverage data.	22
6.4.1	Gcov.	22
6.4.2	Gcovr and Cobertura.	23
6.5	Ada Metamodel.	24
6.6	Generate the intermediate representation.	26
6.6.1	Package body.	26
6.6.2	Subprocedure body.	27
6.6.3	Expression Function.	28
6.6.4	Task Body.	29
6.6.5	Accept statement.	30
6.6.6	Select statement.	31
6.6.7	Call expression.	31
6.6.8	If statement.	32
6.6.9	Identifier type.	34
6.6.10	Case statement.	34
6.6.11	Loop statements.	34
6.6.12	Return and Delay statements.	34
6.7	Resulting model.	35
7	Formal model generation	37
7.1	Transformation concept.	37
7.1.1	Model-to-Model transformation.	37
7.1.2	Model-to-Text transformation.	39
7.2	Linking calls and implementations.	39
7.2.1	Subprocedure calls.	39
7.2.2	Accepts, Events and Mutex linking.	40
7.3	Transformation step-by-step.	41
7.3.1	Subprocedure transformation.	41
7.3.2	Task transformation.	41
7.3.3	Statement transformation.	41
7.4	The full Uppaal model.	46
8	Formal verification	49
8.1	Deadlock.	49
8.2	Reachability analysis.	49
8.3	Liveness analysis.	50
9	Discussion	51
9.1	Validation.	51
9.2	Results.	55
9.3	Strengths & Limitations.	55
10	Conclusions & Future work	57
10.1	Summary.	57
10.2	Recommendations.	58
10.3	Future work.	59

11 Reflection	61
Bibliography	i
A Ada intermediate model	iv

SUMMARY

Cyber-Physical Systems (CPSs) are systems where physical components are controlled by computational components, e.g. computers or micro-processors. Since the early days of the introduction of computers, cyber-physical systems are increasingly around in our day-to-day life and getting more and more complex.

In order to make sure that these systems always do what they are intended to do, validating the software running on these systems is of paramount importance, as very high costs or even human lives can be at stake.

A promising technique to test software is formal verification. Formal verification aims at proving properties on the code such that it is guaranteed that a property holds on the system, in contrast to other testing methods where no guarantees can be given.

One of the main disadvantages of formal verification is the so-called state space explosion. This means that the number of possible system states reaches huge numbers, especially for concurrent systems where parallel processes interact with each other.

In order to limit the state space size and thus potentially enable formal verification on CPSs, we abstract parts of the original source code away. Our approach is to automatically extract the happy-flow code of our CPS and apply verification on this part of the code only. Our main research question is stated as: *To what extent is it possible to prove formal properties on the 'happy flow' code of a real-time, concurrent cyber-physical system with a high ratio of repetitive tasks?*

During our research a method is developed to automatically extract the happy-flow code by using code coverage tooling. The CPS is executed in a nominal way and by checking how many times each line of code is executed it is found what code is initialization or error handling code, and what code is part of the happy flow.

We designed a tool which translates the happy-flow code into an intermediate representation, or model, of the source code. By means of model-driven engineering, a model-to-model transformation has been created which subsequently transforms the model of the source into an Uppaal model.

Uppaal is a model checker which can be used to check properties on a formal model. Using the method and tooling developed during our research, a working Uppaal model of our cyber-physical system can automatically be generated. This makes the modelling process quick and removes the risk of human mistakes compared to creating a model manually.

Some simple properties have successfully been verified on the model, proving that the model mimics the behaviour of the CPS. However, verifying more complex properties still run into the state explosion problem. It is expected that improvements on our work can lead to a better approximation of the source code, which will result in a smaller state space, thus enabling the verification of more complex properties.

1

INTRODUCTION

Cyber-Physical Systems (CPSs) are systems where physical components are controlled by computational components, e.g. computers or micro-processors ([National Science Foundation, 2021]). Since the early days of the introduction of computers in the second half of the 20th century, cyber-physical systems are increasingly around in our day to day life and getting more and more complex.

In order to make sure that these systems always do what they are intended to do, validating the software running on these systems is of paramount importance. As can be imagined, very high costs are at risk when for example a robot cell destroys a valuable workpiece, or even human lives are at stake when looking at computer controlled aerospace systems.

Therefore, in parallel to the rise of software development tools, also a wide range of software testing methods was, and still is being, developed, think of unit testing, system testing, black-box testing, white-box testing, static testing, dynamic testing, etc.

One of the approaches which has been around for quite some time but which is still not mainstream in the software industry is testing by means of formal verification [Bjesse, 2005][Wayne, 2019]. A major benefit of this approach is that it can be automated, but the main reason not being used widely is that it does not scale well for larger or complex systems.

The research carried out in this thesis aims at finding a method to create a formal model of complex CPSs with a high ratio of repetitive and concurrent tasks, while limiting the model size by only looking at the ‘happy flow’ of the CPS, ultimately to lower the threshold for industry to use formal verification. Therefore our main research question is defined as:

To what extent is it possible to prove formal properties on the ‘happy flow’ code of a real-time, concurrent cyber-physical system with a high ratio of repetitive tasks?

1.1. CONQUER THE STATE SPACE EXPLOSION

A good introduction to model checking, past developments and the limitations due to the state space explosion is given by Clarke et al. [2012]. As indicated by the authors, a main limitation of formal verification is the tremendous number of states a software program can be in, especially for concurrent systems where different processes run in parallel and interact with each other. Even for an average sized concurrent program, the state space easily grows to numbers where formal verification is not possible anymore.

In order to conquer the state space explosion it can either be tried to design faster verification methods and tools, but it can also be tried to limit the number of states in the model. A lot of research has been conducted on the former option, which resulted for example in tools using symbolic verification methods instead of explicit methods.

In order to reduce the number of states in the model, different options have been used in the past, like removing unused variables or data, limit the possible values of variables, techniques like partial order reduction and symmetry reduction, or abstract away parts of the system behaviour.

Our research focuses on the latter option, i.e. abstract away parts of the system behaviour. By means of filtering the happy-flow code from the full code base, the non-happy flow is abstracted away and the number of states in the model is reduced.

The ‘happy flow’ is defined here as the nominal machine cycle without interference from the outside world. For the kind of machines with a high ratio of repetitive tasks targeted in this research, it is expected that less than 20% of the source code is used for the happy flow. Therefore a significant state space reduction can be achieved.

1.2. SYSTEM UNDER TEST: HIGH SPEED DIE ATTACH MACHINE

The research described in this document will be carried out in the context of the software development for a high speed die attach machine: Itec’s Adat3-XF¹, shown in Figure 1.1. This Cyber-Physical System is used for the production of semiconductor chips, and picks dies (bare chips) from a wafer and places the dies on a substrate.

The heart of the machine is the *transfer mill* which is a rotating device with multiple *pick & place heads*. On the backside of the wafer there is a *needle* which pushes a single die on the wafer a little bit up. A pick & place head picks the die from the wafer and places it on a substrate a few cycles later. The wafer is held in place by a *wafertable* which indexes the wafer in x and y direction. Furthermore multiple cameras are present in the machine inspecting a product before, between and after it has been picked and placed.

The machine is controlled by a single computer running software written in the programming language Ada. The software contains many real-time parallel processes, e.g. picking, transferring, inspecting, and placing a die, with *real-time* being in the order of microseconds. A full machine cycle takes around 75 milliseconds, and within each cycle many synchronisations between the different processes occur.

Due to the critical production process, formal verification of the software would add a lot of value to guarantee the correct working of the machine. However, because of the complexity of the software and the number of concurrent processes, it is assumed that a direct translation from the source code to a formal model will result in way too many states for formal verification to be possible. Therefore some attempt should be made to reduce the number of states.

1.3. GOAL OF THE RESEARCH

The goal of this research is to come up with a method which makes it possible to apply formal verification on the code of a complex cyber-physical system with a high ratio of repetitive tasks, without running into the state space limits of the available verification tools. In order to do so, a compromise is needed to limit the number of states which in this research

¹<https://www.itecequipment.com/products>

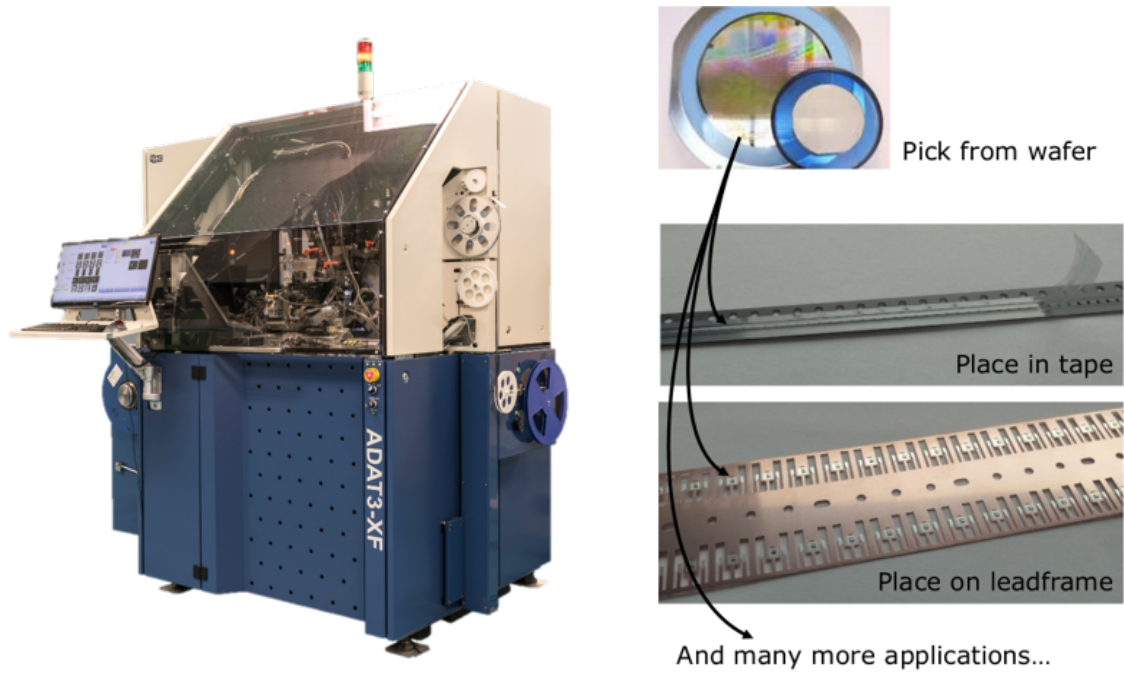


Figure 1.1: Left: Itec's Adat3-XF Die Attacher; Right: The application of the machine, pick chips from a wafer and place them e.g. in tape or on a leadframe.

will be done by only looking at the source code of the nominal machine cycle or 'happy flow' of the machine.

As stated, filtering the source code is a compromise because not all system behaviour is present anymore. Therefore the model becomes an under-approximation of the complete system. As a result when a formal property holds on the model, it cannot be guaranteed that this property also holds for the complete system. However, guarantees can be made the other way around, when a formal property does not hold on the model, it is guaranteed that the property will also not hold for the complete system.

For the developed method to be usable in an industrial environment, the tooling should do as much as possible in an automatic fashion such that no in depth knowledge of formal verification and model checking is required.

1.4. RESEARCH CONTEXT

This research is conducted as a graduation project for the master Software Engineering of the Open University. As the context for this project it is chosen to use the working environment of the author, i.e. the Adat3-XF machine of Itec such that a close relation between the research and the industry is established. At the same time ESI (TNO) carries out a project at Itec aiming at modelling the machine cycle of the Adat and improving the software and software development methods, in parallel to our research. The methods developed during our research can help to bridge ESI's efforts in modelling the machine cycle with foreseen improvements on the source code of the machine.

1.5. STRUCTURE OF THE REPORT

This thesis is structured as follows: Chapter 2 gives some background information on different subjects used throughout the research. Chapter 3 presents an overview of research related to our research and the contribution of our research. The main research question is divided into different sub questions which are given in Chapter 4, along with the research method per sub question. Chapters 5, 6, 7 and 8 describe the performed research per sub question. Chapter 9 discusses the validation and results of our research and summarizes the strengths and limitations of the work. The final conclusions and recommendations are given in Chapter 10. Finally Chapter 11 gives a personal reflection on the research and the process.

2

BACKGROUND

This chapter introduces different concepts and tools which are used throughout our research. If the reader is already knowledgeable on these subjects, the specific section can be skipped.

2.1. FORMAL VERIFICATION

Formal verification is about trying to mathematically prove specifications or properties on a given system [Bjesse, 2005][Clarke et al., 2012]. This is done by finding a formal proof on a mathematical model of this system. Typically the abstract mathematical models are constructed by means of finite state machines which unambiguously describe the behaviour of the system.

In order to prove properties on a system three steps need to be taken: (1) formally specifying properties of the system, (2) creating a formal model of the system, and (3) assessing whether the formal model behaves as described by the formal specification, also called verification or model checking [Clarke and Emerson, 1982]. The three steps are further described below.

2.1.1. FORMAL SPECIFICATION

Step one in the process of formal verification is specifying formal properties which describe the intended behaviour of the system. In order to reason about a formal model, some form of logic is required. When also the order of events should be considered, which is typically the case for Cyber Physical Systems, temporal logic gives the tools required.

Several kinds of temporal logics have been developed over the years, of which the most important ones are Linear Temporal Logic (LTL) [Pnueli, 1977] and Computation Tree Logic (CTL) [Clarke and Emerson, 1982]. Later, [Clarke et al., 1986] defined CTL* which is a superset of both LTL and CTL, combining linear and branching temporal logic.

CTL* formulas have both a path quantifier and a temporal operator. Two path quantifiers are used, specifying either all computation paths **A** or at least one path **E**. The temporal operator describes properties which hold along the path as specified by the path quantifier. CTL* has 5 of these temporal operators

X p in the next state, p holds

F p eventually p holds

G p p holds Globally

p U q eventually q holds, and until then p holds

p R q p releases q, or q always holds

Now formal specifications can be given like **AG(p)** meaning p holds in all states of all execution paths. Combinations of CTL* operators are also possible, one of the most common combination is **AG(p => AF(q))** which denotes the so called liveness property. This property evaluates to: for all execution paths, whenever p holds, then eventually q holds.

By means of CTL* the following formula could for example be defined:

$$AGEF(p) \tag{2.1}$$

This formula means: for all states of all execution paths it is true that there is an execution path where eventually p holds.

2.1.2. FORMAL MODELLING

Software programs are typically described by a model where each position in the program is described by a state. The program can go from one state to another in response to certain inputs. When the set of states of the program is finite, such a model is called a finite state machine or automaton [Rabin and Scott, 1959].

One way to describe finite automata is by means of Labelled Transition Systems [Burkart et al., 2001]. These systems have a finite set of states and between the states, transitions are given which define how the system can go from one state to another. Each transition is labelled by a proposition which should hold in order to take the transition. Figure 2.1 shows such a finite automaton of three states and proposition p and q.

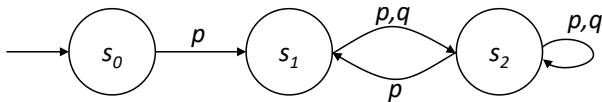


Figure 2.1: Finite automaton or Labelled Transition System

Assume Formula 2.1 is rewritten to $AGEF(s_2)$, meaning that from all states of all execution paths eventually state s_2 can be reached. When this formula is applied to the automaton of Figure 2.1 it is found that the formula holds on this state machine because from each state it is possible to reach the state s_2 . Note that the formula $AGEF(s_0)$ does not hold on this system because from state s_1 and s_2 it is not possible to reach state s_0 .

Timed Automata In order to reason about asynchronous timed systems, it is required to add a notion of time to the models of these systems. For this purpose [Alur and Dill, 1994] introduced the concept of Timed Automata. Timed automata are finite automata extended with a finite set of clocks representing continuous time. Clocks are declared by clock variables which are read-only except for resetting to zero. The value of the clock is the time since the last reset of the clock. Note that in timed automata, the state of the system is resembled by the 'location' of each automaton in the system and the value of all clocks in the system. Therefore in timed automata, instead of 'state', the term 'location' is used to denote a node in the automaton.

Now transitions between locations can have clock guards which define at which value of the clock the transition is valid. Besides transitions, locations can contain constraints about clocks as well; called an invariant. The invariant defines for what values of a clock it is allowed to be in that location. Furthermore a clock can be reset on a transition. If with the next time value it is not allowed to be in a certain location, a transition to another location has to be made, when this is not possible a deadlock situation is reached. It is assumed that transitions are instantaneous, hence time only progresses in locations, it is furthermore assumed that all clocks run at the same rate.

To be able to model concurrent processes, different timed automata can synchronize with each other on actions. For this purpose, transitions can be labelled with actions. When two automata have a transition with the same action, the automata need to wait to make the transition until both automata can take the transition at exactly the same time.

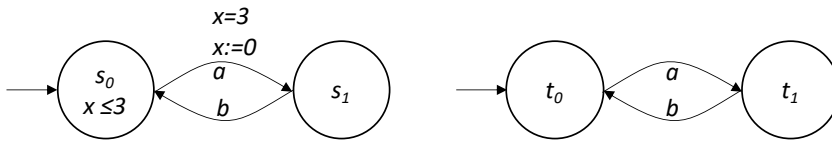


Figure 2.2: Two timed automata which synchronize on an a and b action

Figure 2.2 shows two timed automata with a clock x . They both start in their initial location, with clock variable $x = 0$. As soon as x reaches the value of 3, the left automaton is not allowed to stay in s_0 any longer and needs to take the transition to s_1 . This transition was also not allowed to be taken earlier due to the clock guard $x = 3$. Furthermore the clock variable x is reset to 0 during the transition. Because the two automata synchronize on a , as soon as the left automaton goes from the left to the right location, the right automaton does the same. Note that the automata are allowed to stay infinitely long in s_1 and t_1 respectively, but as soon as one of the automata takes the transition back, the other one will do the same due to the synchronization on b .

2.1.3. MODEL CHECKING

By having a formal specification language and a formal modelling method it is possible to verify whether the model satisfies the formal specification. Hence the question is, does model M satisfy formula f , or: $M \models f$.

Different algorithms have been developed to perform this step, both in an explicit and a symbolic manner. But as in both cases the verification is a matter of running the algorithm, a lot of tooling is developed to perform this step automatically. Well known tools are e.g. SPIN¹ ([Holzmann, 1991]), NuSMV² ([Cimatti et al., 2002]) and UPPAAL³ ([Behrmann et al., 2006]). Uppaal is specifically designed to verify Timed Automata and features an easy-to-use interface to construct and check models.

2.2. UPPAAL

For the research given in this thesis it will appear that Uppaal fulfills the requirements we have for a model checking tool. This section describes some more background on Uppaal

¹<http://spinroot.com>

²<https://nusmv.fbk.eu/>

³<https://uppaal.org/>

and the way to use it, in order to better understand the rest of this research.

Uppaal is specifically designed to model and verify Timed Automata. Figure 2.3 shows the left automaton of Figure 2.2, using Uppaal notation.

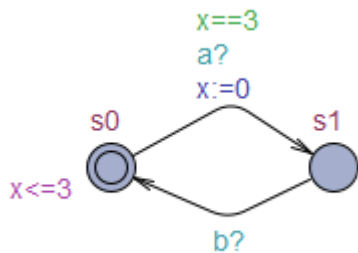


Figure 2.3: An example showing the left automaton of Figure 2.2 in Uppaal

2.2.1. CONCEPTS & SYNTAX

This section gives a quick introduction to the concepts and syntax of Uppaal.

TEMPLATES

A system as defined in Uppaal is composed of a network of timed automata (NTA), with all automata running in parallel. An automaton in Uppaal can be instantiated multiple times with different parameters when desired, therefore an automaton definition in Uppaal is referred to as a *Template*.

A template is composed of *locations* shown as nodes, and *edges* representing the transitions between the locations.

LOCATIONS

Figure 2.2 shows two nodes which are locations s0 and s1 of the automaton. An important property which can be given to a location is the *invariant*; an expression which should always hold in order for the system to be allowed to be in that state. Note that invariants are optional, in Figure 2.2 s0 has invariant $x \leq 3$ while s1 has no invariant.

One of the locations in an automaton should be marked as the initial state. In Uppaal this is visualized by the double circle. Referring again to Figure 2.2, location s0 is the initial state of this template.

Lastly there are two more optional properties for a location, being *Urgent* and *Committed*, note that a location can either be urgent or committed, not both. An urgent location represents a state of the system in which it is not allowed for time to pass. Before time is allowed to pass again, one of the outgoing transitions should be taken. Because of this, the state space of the model is largely reduced.

Committed locations go one step further compared to urgent locations and do not allow other processes to perform any (non-committed) transitions before an outgoing transition of the committed location is taken.

EDGES

Edges represent directed transitions from one location to another. Like locations, edges can be given different optional parameters to control the behaviour of the edge. These options are explained below.

Synchronization Edges can be given a synchronization label. This label identifies a synchronization channel which is used to synchronize processes with each other. A synchronization label can either be of the form $e?$ or $e!$ where the question mark defines the receiving end of the channel and the exclamation mark the sending end of the channel. It is like a rendezvous between two concurrent processes. Both transitions will always be taken at the same time. Note that there may be multiple receivers, but only one receiver actually synchronizes.

A special form of synchronization is the use of a broadcast channel. In that case there can be multiple receiving edges. Note that for broadcast channels the sender can always 'fire' while only the receivers which have enabled edges synchronize with the sender.

Guards Guards define when an edge is enabled or not i.e., when an edge can be taken or not. Often this involves clock variables which indicate at what value of the clock a transition can be taken or not. But guard expressions are not limited to contain clock variables.

Updates Update expressions are used to assign values to variables to moment the edge is taken. These variables can again be clock variables or other system variables.

2.2.2. MODEL VERIFICATION

In Section 2.1.1 we discussed temporal logics LTL, CTL and CTL*. Uppaal uses a subset of CTL to define properties on the model, or queries as they are called in Uppaal. The main properties which can be checked by Uppaal are:

Reachability properties These properties specify whether a state where property p holds, can be reached. In Uppaal this is written as $E\langle\rangle p$.

Safety properties Safety properties specify a property which should always hold. This is either given by $A[] p$ saying that p should hold in all states on all paths; or by $E[] p$ which defines that p should hold in all states of a certain path.

Liveness properties These properties specify that some property p will eventually hold. This can either be written as $A\langle\rangle p$, which means that for all paths p should hold eventually, or by $p \dashrightarrow q$ defining that when p holds, then eventually q will hold.

Note that in Uppaal the property p can also represent a location in one of the automata, or templates. This makes it for example easy to write a query which checks on the model whether a certain location is reachable, i.e.: $E\langle\rangle \text{template_name.location_name}$.

2.3. SOURCE CODE TO MODEL TRANSLATION

In order to apply model checking techniques to a code base, the code needs to be translated to a formal model, e.g. a finite state machine. As most of the time no straightforward translation of language constructs is possible, a mapping from one to the other is required.

In general translating the source code to a model is done in two steps. One, retrieving the semantics of the code in a form which can be easily parsed and traversed, and two, mapping the source code constructs to the available constructs in the modelling language.

Step one is often done by parsing the code, convert it to an Abstract Syntax Tree (AST) and from the AST, create a Control Flow Graph (CFG). An AST is, as the name implies, an

abstract representation of the syntax described in a tree. Each node in this tree maps to a construct in the source code, and any non-informative information like semicolons and parentheses are left out. Figure 2.4 show a simplified AST of the code from Listing 2.1.

```

if x == y
  z := x;
else
  z := y + 2;

```

Listing 2.1: Code for simple AST

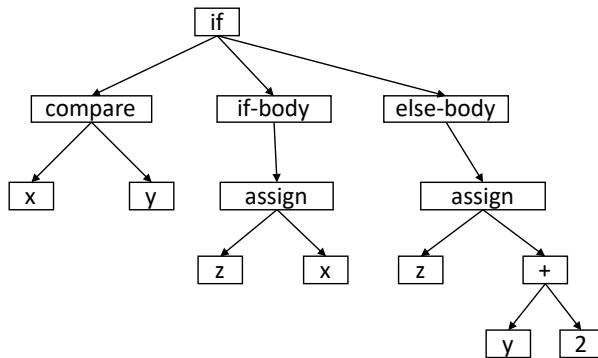


Figure 2.4: Simplified AST of the code from Listing 2.1

Because an AST provides an unambiguous mathematical representation of the source code, it can be used to create other representations like a Control Flow Graph. A CFG is a directed graph where each node represents a statement, and the edges represent the control flow. Figure 2.5 shows a simplified CFG for the AST of Figure 2.4 and the code from listing 2.1.

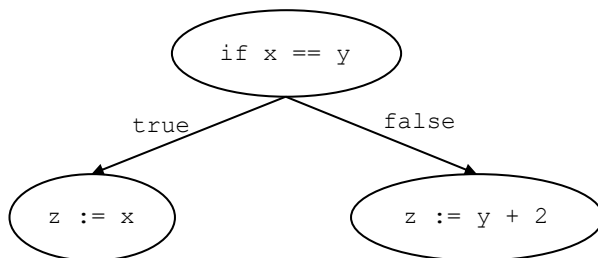


Figure 2.5: Simplified CFG of the code from Listing 2.1

The second step of the source code to model translation is mapping the code constructs to the available constructs in the modelling language. The possible approaches for this step heavily depend on the modelling language, but information from both the AST and CFG can be used to perform this translation. Often no one-to-one translation is available, therefore it is important to take the goal of the model into account to give directions on how to do certain translations.

In case of a direct translation, the model will mimic the CFG to a high extent, but in case of a less direct translation, the model abstracts away from the CFG. The further the model abstracts away from the CFG, the more important it is to validate whether the model is still a truthful abstraction of the software program in order to be usable for formal verification.

Both the AST and CFG concept will be used in this research as tools to end up with a formal model of the code.

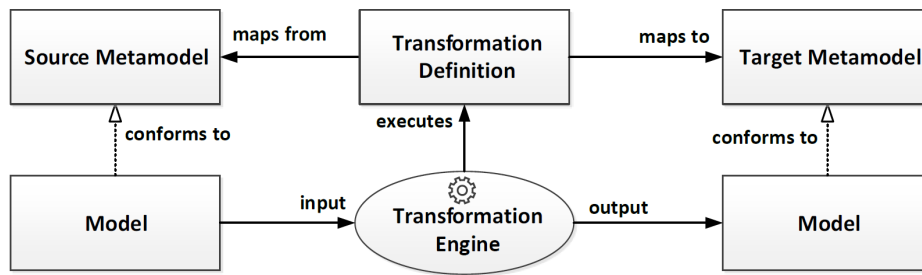


Figure 2.6: The concept of model transformation based on metamodels; from [Schivo et al., 2017]

2.4. MODEL DRIVEN ENGINEERING

Models are an abstraction of a system, giving a simplified description, or showing a part of the complete system. Different models can highlight different parts of a system, or different levels of details. Often these kind of models are used for documentation purposes and knowledge purposes. Model driven engineering is an approach where models are not just used for documentation, but also for the creation of new (software) artefacts [Rodrigues da Silva, 2015].

In order to perform engineering on models, some formalism is required describing how a model should look like, e.g. what parts can it contain, how are relations between parts described, etc. This is what is typically called a meta model, a description of the syntax of the eventual model.

Once a meta model is available, all kind of new possibilities arise like the automatic transformation of models from one domain to the other. An example of this is shown in Figure 2.6. Here both models on the left and right side have their own meta model, describing the respective model syntaxes. Now a transformation definition is given which defines how artefacts from the input meta model should be mapped or transformed to artefacts of the output meta model. Using this transformation definition, tooling can be used to automatically create an output model from an input model.

Different tooling is available supporting the Model Based Engineering approach. A common and widely used framework for this is the Eclipse Modelling Framework (EMF)⁴. In EMF, meta models are defined in so called Ecore files which can be registered inside the framework. An actual model can then be assigned to a meta model, making it an instance of the meta model. This allows Eclipse for example to check whether the model instance correctly adheres to the syntax described in the Ecore file.

Where EMF is build directly on top of Java, the Epsilon framework⁵ builds on top of EMF. The Epsilon framework provides an even more user friendly approach to define meta models in its Emfatic notation. Furthermore Epsilon includes notations to write transformations between meta models: the Epsilon Transformation Language (ETL).

ETL allows to write *rules* which define how an element type of the source (meta) model should be translated to one or more elements of the output (meta) model. EMF follows an object oriented flow in which rules can be defined abstract, and rules can extend on other rules.

⁴<https://www.eclipse.org/modeling/emf/>

⁵<https://www.eclipse.org/epsilon/>

3

RELATED WORK

This section describes previous research which is related, or has overlap with our research. A division is made between model checking source code (Section 3.1) and program slicing (Section 3.2). Section 3.3 discusses the contribution of our research with respect to the discussed related work.

3.1. MODEL CHECKING OF SOURCE CODE

Back in 1999 [Burns and Wellings, 1999] wrote about using model checking in order to verify concurrent Ada programs. For this they use the UPPAAL model checker which had just been released at that time. The authors explain some aspects of the Ada language which enables powerful concurrent programming, e.g. Tasking and Protected Types. They propose to use Finite State Models (FSM) to verify programs build upon these concurrent Ada aspects. It is noted that model checking can never give a 100% guarantee as there is always an informal relation between the source code and the model, and also between the source code and the compiled machine code. The authors state that model checking can be a good method to verify rather complicated concurrent programs, especially with more mature tools becoming available.

In contrast to the previous work, the research described in the remainder of this section apply automatic model extraction from source code.

[Bowman et al., 1999] present and compare three approaches to extract models from Java code. These approaches are parsing source code, disassembling byte code, and profiling where a model is built by monitoring run-time behaviour. Parsing source code, gives the most accurate model, but is the most complex and slowest of the three options. For the purpose of static analysis the disassembler will be sufficient and is simpler and faster. When information is to be found in the model which is only available from execution of the code, the profiler option works best.

[Corbett et al., 2000] designed a tool called Bandera which extracts and creates models from Java source code for different existing model checkers, e.g. Spin and SMV. In order to reduce the state space, the authors apply slicing to remove parts of the code which are not relevant for checking a given property. Furthermore the tool supports user controlled abstraction of variable values in order to reduce the state space as well.

The slicing is based on a slicing criterion. From the property to be verified it is found what variables and procedures have influence on this property and from there on all irrele-

vant parts are removed from the program. The slicing approach used is based on [Horwitz et al., 1988] and further described in [Hatcliff et al., 1999a].

[Holzmann and Smith, 2002] show a technique to derive models from ANSI-C source code for the SPIN model checker (also developed by Holzmann). This technique is specifically targeted at distributed and concurrent systems.

In order to reduce the state space, the authors use a map that defines which type of statements are relevant for the current property to be verified and which types are not. According to the authors the number of unique statement types for a given application is in the order of a few hundred, which is accepted to be crafted manually, especially compared to manually crafting a full model. This map is also used to map certain statements or variables to a default value (e.g. *true* or *skip*) which further limits the state space.

[Silva et al., 2011] and [Faria et al., 2012] both present a tool to construct models from Ada code. [Silva et al., 2011] developed a tool which is capable of automatically extracting UPPAAL models from Ada code. The developed tool is aimed at translating Ada code which complies to the Ravenscar profile [Burns et al., 2003]. This is a subset of Ada tasking, especially targeted to safety-critical hard real-time systems. The tool furthermore uses the Ada Semantic Interface Specification (ASIS) [Bladen et al., 1991] to retrieve the semantics from the Ada source code. Annotations on timing requirements are added in the code. These annotations give an upper bound to timing constraints of functions.

Based on the ASIS representation of the source code, a control flow graph (CFG) is created for each Ada construct and each CFG is eventually translated into an Uppaal template. This template contains the timed automata of the specific Ada construct. Each node in the CFG is therefore mapped to an Uppaal state, and each edge to a transition.

The clocks in the Uppaal templates are also generated from the source code. This is both done based on code constructs like *delay until* and variables of type *Ada.RealTime.Clock()*, but also by means of code annotations. For example the *delay until* construct gives a lower bound on a clock constraint, but there is no language construct which gives rise to an upper bound. Therefore the authors introduce a *deadline annotation* which results in an upper bound for a clock in the Uppaal template.

[Faria et al., 2012] present an approach to extract models from Ada code for the purpose of model checking. The research has a special focus on modelling concurrent programs. In this context a tool called ATOS (Ada TO SPIN) has been developed which generates PROMELA models from Ada code to serve as an input to the model checker SPIN.

The ATOS tool is capable of translating a subset of the Ada language. Again the Ada Semantic Interface is used to extract information from the source code. The tool is also able to automatically infer formal properties from annotations in the code, inspired by SPARK¹ annotations. The annotations can either represent a general assert statement, or pre- and post-conditions for functions, procedures and task entries.

[Yildiz et al., 2017] present a technique to derive Uppaal models from Java byte-code. Their approach is based on the Model-Driven Engineering (MDE) technique as presented by [Schivo et al., 2017]. This framework is a general approach to transform domain specific models to UPPAAL, and also to transform UPPAAL results back to a domain specific representation. [Yildiz et al., 2017] construct a Control Flow Graph (CFG) from the Java byte-code, which is enriched with loop information, timing information and recursion handling to support the transformation to a timed automata. The enriched CFG forms the domain

¹<https://www.adacore.com/about-spark>

specific model which is fed into the model transformation framework, resulting in an UP-PAAL model.

3.2. PROGRAM SLICING

Program Slicing was first introduced by [Weiser, 1984]. The idea of program slicing is to reduce a program such that it only contains those statements which have influence on the values of a certain variable or point of interest. One of the goals of the author for program slicing is to make it easier to debug code when it is known which variable or at which point in the code something goes wrong.

In the following years, many researchers adapted and used the idea of program slicing in different directions and for different purposes. [Silva, 2012] presents an overview of these directions, with one of the directions also being for the purpose of model checking, i.e. [Hatcliff et al., 1999b]. In the latter work the authors apply an adapted form of slicing in order to reduce the source code such that it becomes less costly to apply model checking on this code. They adapted the slicing criterion such that all information required to prove a certain LTL formula is in the slice, and nothing more than that.

[Korel and Laski, 1988] introduce the concept of Dynamic Slicing. This is a program slice which corresponds to a specific execution of the program for a certain input value, in contrast to *standard* slicing where all possible input values are considered, hence a dynamic slice is much smaller than a static one. [Korel and Laski, 1988] state in their definition that the path of a slice should be identical to the path in the full program. Hence, the slice has the same execution sequence as the full program, which is called a path-aware slice. The purpose of this work is to ease program debugging, as often it is known for which input value a certain bug is present so only this execution needs to be analysed.

[Agrawal and Horgan, 1990] also introduce the concept of Dynamic Slicing, but in contrast to [Korel and Laski, 1988], they do not require that the path of the slice should be equal to the path of the full program. This even further reduces the size of the slice compared to the previous approaches.

3.3. RESEARCH CONTRIBUTION

Existing research for translating Ada code into formal models as mentioned in the previous section all use a subset of the Ada language, i.e. the Ravenscar profile, or the Spark subset. Our research aims at translating the complete Ada language. The existing research furthermore uses the Ada Semantic Interface Specification (ASIS) to ‘process’ Ada code, this interface has been replaced by the LibAdaLang² which we will therefore use in our research.

Eventually the contribution of our research is to combine a form of dynamic program slicing (code filtering) with the concept of model checking. The dynamic slicing will be used as a means of source code reduction in order to conquer the state explosion problem such that model checking becomes a realistic option for complex CPS control software.

²<https://github.com/AdaCore/libadalang>

4

METHOD

This chapter describes the research method of this thesis in more detail. First the main research question and sub-questions are given and then the research method per question is described.

4.1. RESEARCH QUESTIONS

As given in the introduction of this thesis, the main research question of this project is:

To what extent is it possible to prove formal properties on the ‘happy flow’ code of a real-time, concurrent cyber-physical system with a high ratio of repetitive tasks?

Due to the problem of state explosion, it is generally considered impossible to translate a full code base of a cyber-physical system, featuring real-time and concurrent aspects, to a formal model. Therefore the goal of this research is to only try to model the ‘nominal’ machine cycle, or ‘happy flow’, and to prove formal specifications on this part only. As manual extraction of the happy flow from the code base is not only a lot of work, but also error prone, run-time information is used to filter the happy flow code from the full program code.

In order to validate the research results, the research is focused on an existing machine control system written in the programming language Ada.

The research has been divided over four research questions, listed below:

- RQ-1** Having only a model of the ‘happy flow’ code of a CPS, what properties of the system can be verified and can reveal interesting information for domain experts?
- RQ-2** How can the code of the machine’s ‘happy flow’ be extracted automatically, using run-time information?
- RQ-3** How to automatically translate a partial program written in Ada, as found by answering **RQ-2**, into a formal model that allows the properties found by answering **RQ-1** to be verified?
- RQ-4** What results can be found from verifying the properties defined by answering **RQ-1** on the model, found from answering **RQ-3**?

4.2. RESEARCH METHOD

Per sub-question different approaches are taken to find an answer to the questions. For each sub-question the research method is discussed here.

Method RQ-1 The main goal of the complete research is to be able to verify formal properties on a partial program. However, because of having a partial program, the properties to be checked should be well chosen in order to be meaningful for the full program. Deductive reasoning is used to find an answer to this question. The results are given in Chapter 5.

Method RQ-2 In order to answer this question, tooling is designed and built to perform the code extraction automatically. The code is instrumented such that the execution count of each statement can be recorded. After running the machine for a small amount of time in a ‘happy flow’ mode, it is found which code is used for the happy flow and which code is not. Based on this information, our tool generates an intermediate representation of the happy flow code in an XML based format, ready for further processing. The research to answer this research question is described in Chapter 6.

Method RQ-3 This question concerns the generation of a formal model from the intermediate code representation which was the result of RQ-2. As a first step, it has been defined what the formal model should look like, e.g. which artefacts of the system should be present in the model such that the properties found by answering RQ-1 can be verified. The next step is to translate the code to a formal model. For this step tooling is again designed which performs a model to model conversion from the intermediate Ada representation to an Uppaal model.

In order to be useful for the future and to be less error prone, the model generation is done in a fully automatic fashion. Chapter 7 describes the research done to answer this question.

Method RQ-4 After the formal model generation, the model can be used to verify the properties as defined by answering RQ-1. This reveals information of the system which was not known before. During this phase, first the defined properties haven been formalised to temporal logic in the Uppaal query language. These queries are verified on the formal model. The result of this research question is described in Chapter 8.

5

WHAT TO VERIFY ON THE ‘HAPPY FLOW’?

The first research question, **RQ-1**, of this research is stated as:

Having only a model of the ‘happy flow’ code of a CPS, what properties of the system can be verified and can reveal interesting information for domain experts?

In this chapter will try to give an answer to this question.

5.1. DEFINITION OF HAPPY FLOW

In order to answer **RQ-1**, it is required to further specify ‘happy flow’. For this research, the happy-flow code is defined to be the CPS’s source code which is actually executed when the machine performs its nominal machine cycle without any interruptions. Any source code not executed during the happy flow will not be part of the happy-flow model and therefore any formal property tested on the model will not consider this code.

In the case of our CPS under test, the Adat3-XF Die Attacher, the happy-flow cycle is the nominal pick & place sequence without e.g. a run out of materials, operator interventions or motion errors.

5.2. PROPERTIES OF INTEREST

This section elaborates on system properties which can be verified on a model of the source code given the fact that only the happy flow, as defined in Section 5.1, is available.

5.2.1. DEADLOCK & LIVELOCK

In general, interesting properties of concurrent systems are deadlock and livelock situations. Hence a state of the system where two or more processes wait for each other forever.

It is noted that a specific happy flow execution will itself not contain a deadlock situation (otherwise it would not be a *happy flow*), but the code extracted from the happy flow potentially *can* contain deadlocks.

Therefore deadlock and livelock situations are interesting properties to verify on a happy-flow model.

5.2.2. REACHABILITY PROPERTIES

Just checking whether a certain state in the model, i.e. a certain statement in the code, is reachable is not very meaningful for the happy flow. Due to the nature of the happy flow, only source code which is actually executed is part of the model, hence any state should also be reachable in the model.

For model validation this is however a useful property in order to check whether certain states are reachable as expected. Furthermore reachability analysis is in general faster than analysis of safety properties because a single trace satisfying the property is enough to prove the property, i.e. there is no need to search the complete state space.

As our CPS has concurrent tasks, it is interesting to check whether indeed all (main) tasks of the machine cycle get their start signal. This is a good indicator that the start-up behaviour of the machine is correctly captured in the model.

5.2.3. SAFETY PROPERTIES

Safety properties indicate that "something bad will never happen". Hence, this is a property which should hold in all states of the model. Because our CPS performs multiple complex moves where a wrong move can potentially damage products or the machine itself, different safety properties can be thought of related to the physical state of the machine.

Examples could be:

1. The wafertable is not allowed to move when the needle is up in order to not damage the wafer.
2. The transfer mill should not rotate when the pick & place heads are still in an outward position in order to not damage the heads.
3. The wafertable is not allowed to move when a pick & place head is close to the wafer in order to prevent damaged dies.

Situations like example 1 and 2 have a high observability as the machine would no longer function properly when this property is violated. For example when the wafer foil is damaged or the heads are damaged no pick & place is possible anymore. Therefore these situations will not be part of the happy flow and it is not meaningful to test the corresponding safety properties.

Situations like example 3 however, can be more subtle and might actually happen unnoticed. These situations are possibly part of the captured happy-flow execution and therefore very interesting to test on our model.

All three examples define properties related to the physical state of the machine. Therefore, somehow one or multiple states in the model should represent this physical state. Note that in this research we automatically capture the control flow from the source code. Hence knowledge on the physical state of the machine is not directly available and needs to be added to the model for example by adding it as code annotations and extracting it with the happy flow, or by adding it after model extraction.

The latter will not be part of this research, hence it will not be possible to verify safety properties on the model which consider a physical state of the machine. This is left as future work.

5.2.4. LIVENESS PROPERTIES

A liveness property means "something (good) will eventually happen", or when p holds, then eventually q will hold. The model we will eventually have is that of the happy flow of a repetitive machine cycle. Therefore it is tempting to think that liveness in general will be satisfied, but is this really the case?

We have modelled the source code corresponding to the happy flow of the CPS, but this one execution we used to capture the happy flow is not necessarily the only execution flow through the captured code. Therefore liveness properties can actually reveal potential issues in our source code, and therefore are an important category of properties to take into account.

Examples of liveness properties for our CPS are:

1. Is it always possible to pick up a next product?
2. When a pick & place head goes into an unsafe (outward) position, will it always return to its safe position again?

In the context of a repetitive machine cycle there is also another category of liveness properties. That is: is it always possible to return to a defined state in the cycle? This is a property which can be checked for each parallel task in the model and gives a good indication of the absence of deadlock.

5.2.5. SEQUENCE VALIDATION

A different category of domain properties are certain sequence of actions, e.g. is Task A always executed before Task B? Typically this concerns a sequence within one cycle of the machine, because in the example Task A will probably be again executed in the next cycle. Hence to verify these kind of properties, the model needs to have some notice about the separation of cycles.

Examples of interesting sequences in our CPS are:

1. Is the pick & place head in position before the needle pushes a die towards the head?
2. Are all inspections of a product OK before the product is placed?

A caveat for example 2) is that it should somehow be possible to identify a unique product. This is because the inspection of a certain product and the placement of that same product is spread over multiple cycles. It should be noted however that adding unique identifiers will significantly increase the state space of the model.

5.2.6. CYCLE TIME ANALYSIS

An important aspect of our CPS, and production machines in general, is machine speed. For a machine conducting a repetitive task the machine speed is determined by the time required for one cycle.

In this research it is decided not to add the dimension of time to the model for the sake of simplicity. With the choice for Uppaal it is however possible to extend the model in future with timing information of the happy flow.

If the execution time of functions will be retrieved from the code, it becomes possible to perform statistical analysis on the cycle time of the machine. Furthermore one can conduct a critical path analysis of the different parallel tasks.

6

‘HAPPY FLOW’ EXTRACTION

This chapter is about research question 2, or how to retrieve the code which is used for the happy flow of a Cyber-Physical System. RQ-2 is stated as:

How can the code of the machine’s ‘happy flow’ be extracted automatically, using run-time information?

The first part of the happy flow extraction is done by means of existing profiling and coverage tools. These tools instrument the source code and record the execution count of each line of source code while the program is running.

In case of a CPS with a repetitive task, code executed at each repetition is executed many times where initialization code and error handling is executed much fewer times. Therefore it is possible to use coverage data to give an indication about which code belongs to the normal machine operation, i.e., happy flow, and which code belongs to the non-happy flow code.

The second part of the extraction concerns the generation of a representation of the source code which only contains the happy flow. Tooling is designed and created to perform this part automatically.

Figure 6.1 shows the different steps taken to come from source code to the intermediate representation. This chapter explains the details of the top part of the figure. The bottom part of the figure is explained in the next chapter. The repository of our happy flow extraction tooling is available on GitHub¹.

6.1. DETERMINE THE HAPPY FLOW

As discussed in the introduction of this section, we look at the execution count of the source code to find the happy flow. When the execution count is above a certain threshold we consider it happy flow. Code which is executed below the threshold is considered to be part of the initialization code or to the non-happy flow like error handling.

For the system under test, the Adat3-XF, the threshold is put at a number equal to the amount of pick & place heads. This is because initialization code is typically executed for each pick & place head. All code executed more times than the amount of heads is thus considered part of the normal production flow, or ‘happy flow’.

¹<https://github.com/jildert17/happyflow-extractor>

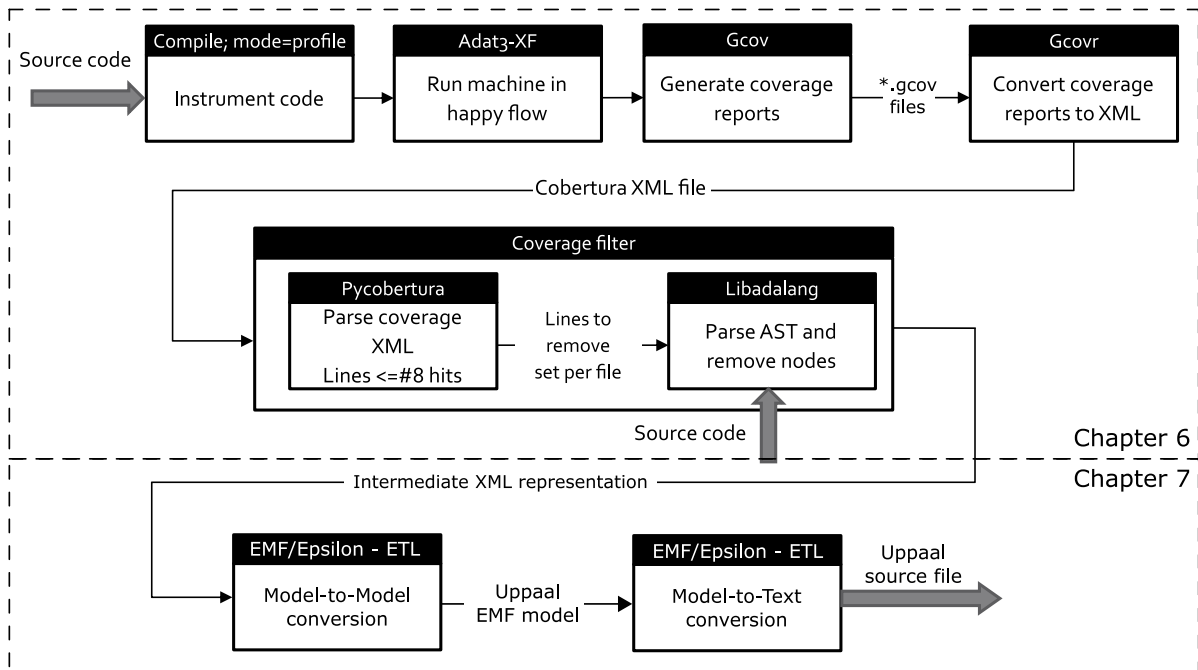


Figure 6.1: Schematic overview of the tool-chain to come from source code to a formal model; The top part is described in this capter, the bottom part in the next chapter.

6.2. INSTRUMENTATION

As a first step the source code needs to be instrumented in order to retrieve the execution count of each line. Because we want to minimize the state space, we choose not to take the full source code of the CPS into account. Support packages related to e.g. the user interface, system libraries and motion planning libraries, do not change often and are therefore less likely to contain issues. Therefore these parts are left out. Later it can be chosen to include more source code if it appears that our tooling works on the smaller code base. This is left as future work.

Initially a relatively new tool called GNATcoverage² was tried to find the coverage information of the source code. It appeared however that this tool only indicates whether a line is executed or not, while it cannot tell how many times a line is executed. Therefore we were forced to use older tooling called gcov³. gcov is the default coverage tool shipped with the GNU Compiler Collection (GCC) and therefore already available in the research setup. Our supplier of the Ada compiler, AdaCore, did however indicate that it will drop support for gcov in the near future in favor of GNATcoverage.

For both GNATcoverage and gcov instrumentation is done on object level which is inserted during compile time. Therefore the source code itself is kept untouched.

In order to enable gcov instrumentation on an Ada project, recompilation is required using compiler switches `-fprofile-arcs` and `-ftest-coverage` and linker switch `-fprofile-arcs`.

During compile time for each object file a `*.gcno` file is created containing information to later reconstruct blocks and link them to line numbers.

²<https://www.adacore.com/gnatcoverage>

³<https://gcc.gnu.org/onlinedocs/gcc/Gcov.html>

6.3. EXECUTING THE HAPPY FLOW

Once the source code is compiled including instrumentation, the software is ready to control the Cyber-Physical System and record coverage information. For this a (simulation) of the Adat3-XF machine is started and the nominal machine cycle is executed for a certain amount of time. For the experiment described in this thesis the nominal cycle of the CPS is executed for 30 seconds. The specific machine configuration used has a speed of around 600 units per minutes, hence when running the machine for 30 seconds, an execution count of 300 is expected for code that is called once per cycle.

The gcov coverage information is written to disk the moment the program receives the shutdown command; i.e., `C-exit()`. For each source file a separate file is saved with the `*.gcda` extension, containing the count data for the source code. It is important to note that this is cumulative data; i.e., when the program is run again, the coverage data is added to the data of the previous run.

6.4. PROCESSING COVERAGE DATA

Once the coverage data has been gathered and stored in the `*.gcda` files, gcov can be invoked to generate human-readable coverage reports, see Section 6.4.1. However, in this project we want the coverage data to be processed further, hence human-readable formats are less suitable. For that purpose the gcov output is translated to a machine-readable format which is described in Section 6.4.2.

6.4.1. GCOV

When gcov is invoked on the code, it scans for the `*.gcno` and `*.gcda` files and by using the original source code it generates a report for each original source file. Listing 6.1 shows a snippet of the gcov output of one of our source files.

The leftmost column indicates the coverage count of the specific line, e.g. 288 times for line 81, which corresponds to the expected 300 times as discussed above. A dash in the same column indicates that no coverage data is available for that line; lines marked by `#####` have not been executed.

Lines for which no coverage data is available can for example be lines which belong to a previous line which is the case for line 82 and 83 in listing 6.1, both lines belong to line 81. Other examples are blank lines, or lines which only contain a keyword and cases where lines are removed during compiler optimizations.

```

-: 80:  overriding
288: 81:  procedure Task_Entry (Mp      : access Attach_Process;
-: 82:                               E      : Entry_Type;
-: 83:                               Collet : Integer) is
-: 84:  begin
288: 85:      case E is
288: 86:          when Start => Mp.Process_Task.Start (Collet = 1);
-: 87:          when others => null;
-: 88:      end case;
288: 89:  end Task_Entry;
-: 90:
##### 91:  overriding procedure Task_Wait (Att : access Attach_Process;
-: 92:                               Result : out Boolean;
-: 93:                               Skip_Status : out Integer) is
```

```

-: 94:      pragma Unreferenced (Att);
-: 95:      begin
#####: 96:          Result := True;
#####: 97:          Skip_Status := Success;
#####: 98:      end Task_Wait;

```

Listing 6.1: gcov report snippet

6.4.2. GCOVR AND COBERTURA

The gcov output as shown in listing 6.1 is easily readable by humans but not very suitable for further processing. For this purpose gcovr⁴ is used. This is a Python tool which calls gcov and is able to translate the gcov output to different formats. One of these formats is the widely used XML-based Cobertura format⁵.

Listing 6.2 shows the gcovr output in Cobertura format of the same source lines as shown in listing 6.1. One Cobertura file is generated containing the coverage information for all source files. This format is suitable for further processing, especially because of the availability of different Cobertura format parsers.

```

<line number="81" hits="288" branch="true" condition-coverage="50% (1/2)">
  <conditions>
    <condition number="0" type="jump" coverage="50%" />
  </conditions>
</line>
<line number="85" hits="288" branch="true" condition-coverage="50% (2/4)">
  <conditions>
    <condition number="0" type="jump" coverage="50%" />
  </conditions>
</line>
<line number="86" hits="288" branch="true" condition-coverage="50% (1/2)">
  <conditions>
    <condition number="0" type="jump" coverage="50%" />
  </conditions>
</line>
<line number="89" hits="288" branch="false" />
<line number="91" hits="0" branch="true" condition-coverage="0% (0/2)">
  <conditions>
    <condition number="0" type="jump" coverage="0%" />
  </conditions>
</line>
<line number="96" hits="0" branch="false" />
<line number="97" hits="0" branch="false" />
<line number="98" hits="0" branch="false" />

```

Listing 6.2: gcovr XML output snippet corresponding to listing 6.1

Using pycobertura⁶ we developed a Python program which parses the Cobertura coverage data as shown in Listing 6.2. This program receives two parameters: the first one being the path to the Cobertura file, and the second one being a number indicating the execution-count-threshold for the happy flow. All line execution counts above this threshold are considered part of the happy flow, and all line frequencies equal or below this num-

⁴<https://gcovr.com/>

⁵<https://github.com/cobertura/cobertura>

⁶<https://pypi.org/project/pycobertura/>

ber are deliberately considered not being part of the happy flow. Note that there are also lines for which no coverage data is available.

The program returns a text file in JSON format which contains per source file a list of line numbers above the threshold and a list of line numbers equal or below the threshold. An example of this is shown in Listing 6.3.

```
{ "nr_of_files": 2, "line_nr_data": [{ "filename": "C:/SVN/Trunk_Clean/source/Applic/
  Adat/adat_thetaz-mill-attach_pos.adb", "below_thres": [40, 43, 91,96,97,98, 115,
    119, 120, 122, 124, 149, 154, 158, 159, 161, 162, 163, 164, 165, 166, 168, ...],
    "above_thres": [64, 81,85,86,89, 100, 102, 129, 132, 135, 139, 144, 145, 156,
    253, 256, 258, 260, 263, 268, 271, 275, ...] } ] }
```

Listing 6.3: Parsed Cobertura output in JSON format. The marked numbers correspond to the line numbers of the GCOV output in listing 6.1

The Python script saves the JSON text file to the C:/temp folder as a means of cache. By giving a '-f' flag to our Coverage_Filter (described below in Section 6.6.), regeneration of the JSON file is forced, otherwise the cache file is used if available.

6.5. ADA METAMODEL

The goal of this project is to end up with a formal model of the happy flow of our CPS and be able to verify properties on the model by a model checker. From the different model checkers listed in Section 2.1.3 it is chosen to use Uppaal [Behrmann et al., 2006] for this research.

The main reason for this choice is that Uppaal supports *timed* automata, although adding a notion of time to our models is considered as a next step and not part of the current research.

The second reason to choose for Uppaal is the availability of an Uppaal meta model and a corresponding model to text transformation in the Eclipse EMF framework, presented by [Schivo et al., 2017]. Refer to Section 2.4 for an introduction to EMF.

This framework is widely used to do model to model conversions, e.g. by [Yildiz et al., 2017] to translate Java byte code into Uppaal models. The latter work is taken as a starting point to do our Ada to Uppaal conversion.

The decision to use EMF leads to the need for some intermediate representation of the Ada code which complies to the EMF/Epsilon modelling syntax. For this purpose a meta-model of the Ada code is designed. The metamodel provides the syntax to construct the intermediate Ada representation or model. Next to that, the metamodel is also used to define the model-to-model transformation which is described in Chapter 7.

The full metamodel is shown in Figure 6.2, using standard UML class diagram notation. The main hierarchy of the model is that of a single Project class at top level which has leaves of type Package_body. A Package_body then contains Subprocedures and/or Tasks which both are the main containers of the abstract type Statement.

There are a few other relations in the model which are required to capture more subtle and infrequently used features of the Ada language.

On the left and the right hand side of the figure different kinds of statements are shown, reflecting most of the common statement types of the Ada language. Besides the common Ada constructs, also statements related to custom-defined concepts are added. These concepts are Events and Mutexes. These concepts have their own types in our metamodel

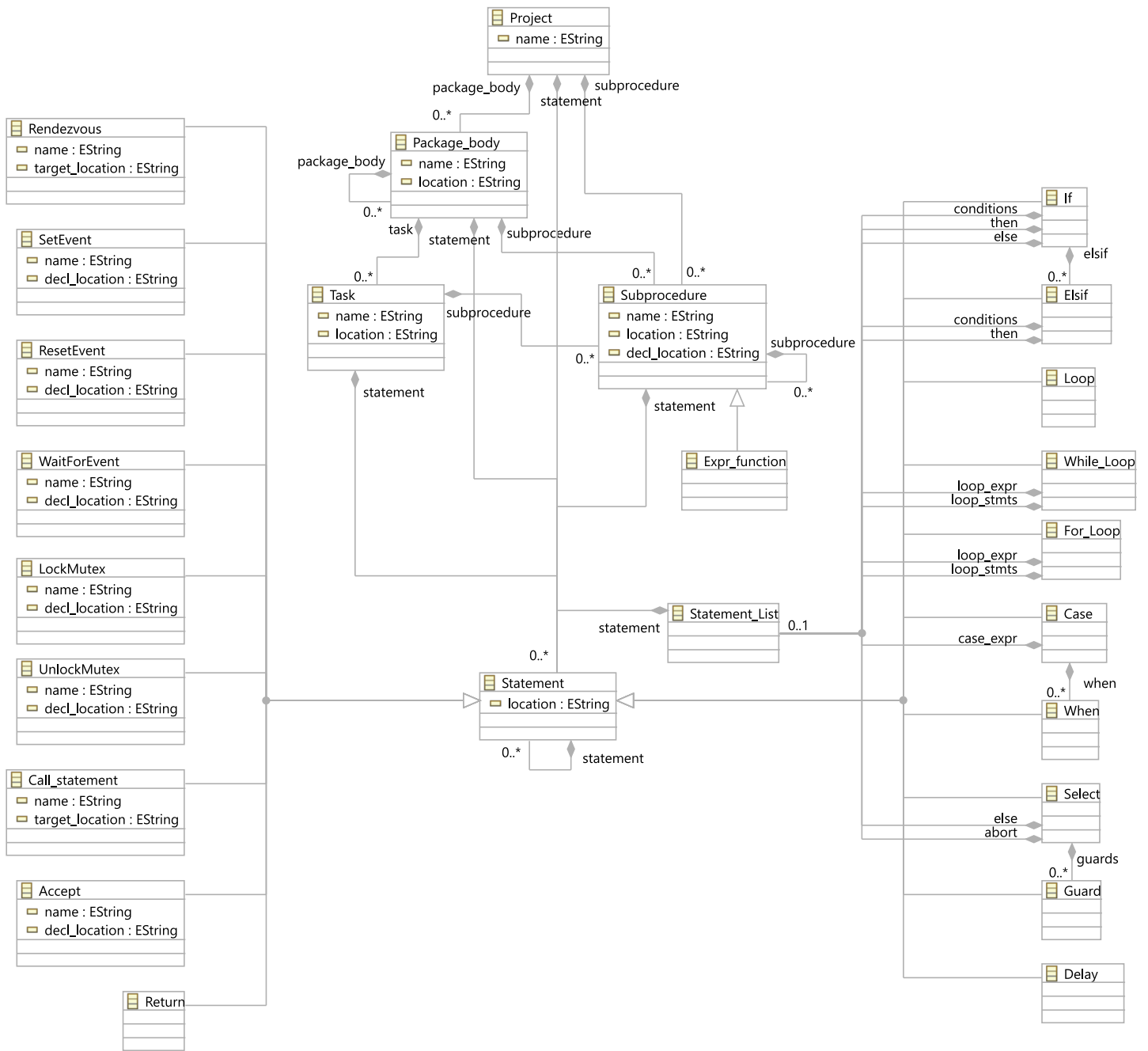


Figure 6.2: Diagram of the Ada metamodel in EMF/Epsilon

because they play an important role in our inter-process communication and will therefore impact our Uppaal model of the CPS.

The event construct is a kind of global signal which has three functions associated to it: `SetEvent`, `ResetEvent` and `WaitForEvent`. By means of this concept concurrent tasks can synchronize with each other.

Mutexes are widely used in computer science to lock certain resources in order to prevent race conditions in concurrent systems. The custom-defined mutexes in our CPS has two related functions, `LockMutex` and `UnlockMutex`. Also this concept will get its own representation in the Uppaal model and therefore has its own statement types in the meta-model.

All statement types inherit from the abstract `Statement` type. Note that statements can again contain statements, such that a tree of statements is supported.

In order to support grouping of child statements, a `Statement_List` type is added. This is for example used to be able to distinguish between the condition statements and else statements of an if statement.

6.6. GENERATE THE INTERMEDIATE REPRESENTATION

Now that it is known from the coverage data which lines we need to keep and which lines should be removed, and we have an EMF metamodel, we are able to filter the happy flow from the original source code and translate it to the metamodel syntax.

For this step we designed a custom tool called the ‘Coverage_Filter’ which reads the JSON file with coverage data and retrieves the AST of the source code. It then translates the happy flow code into an intermediate Ada model. This model is written in XML and complies to the metamodel from Section 6.5.

The tool is, like the CPS source code, written in Ada. To retrieve the AST from each source code file, the `LibAdaLang`⁷ libraries are used. Once the AST is available, the tree is traversed by means of a recursive function. Each node matching a language construct which needs to be translated is processed as described below.

6.6.1. PACKAGE BODY

For the AST nodes of type `Ada_Package_Body` it is not checked whether it is in the happy flow or not as these nodes are not executed and no coverage data is available.

These nodes are translated to an XML node with name `package_body` and attributes location and name.

Table 6.1 shows an example of the translation of the Package body source code via the AST to the intermediate Ada model.

⁷<https://github.com/AdaCore/libadalang>

1: 31:package body Adat_ThetaZ.Mill.Pickup_Pos is	
<pre> PackageBody[31:1-1983:33] f_package_name: DefiningName[31:14-31:41] f_name: DottedName[31:14-31:41] f_prefix: DottedName[31:14-31:30] f_prefix: Id[31:14-31:25]: Adat_ThetaZ f_suffix: Id[31:26-31:30]: Mill f_suffix: Id[31:31-31:41]: Pickup_Pos f_aspects: <null> f_decls: DeclarativePart[31:44-1973:1] ... </pre>	<pre> <package_body location="adat_thetaz-mill- pickup_pos.adb:31:1: " name=" Adat_ThetaZ.Mill.Pickup_Pos"> </pre>

Table 6.1: Package Body translation; Top: Source code, annotated with coverage information; Bottom-Left: AST as retrieved from LibAdaLang; Bottom-Right: translated XML code complying to the metamodel.

6.6.2. SUBPROCEDURE BODY

AST nodes of type `Ada_Subp_Body` represent the bodies of all procedures and functions in the source code. First the line number of a subprocedure is checked against the coverage information: only if the execution count is above the threshold, the node is processed, otherwise it is skipped. An exception is made for the subprocedure named 'Adat_Dietransfer-A3' as this procedure is only executed once, but does contain the main loop of the machine cycle.

A general point of concern when automatically extracting the happy flow is that the starting point of the cycle often gets 'lost'. Typically the machine cycle is triggered by some start procedure which is only executed once. Therefore it always falls below the happy-flow threshold, despite that it belongs to the happy flow. Manual action is required to make sure that this starting point ends up in the formal model.

When a subprocedure body node is to be kept, it is translated into an XML node with a name and a location. Furthermore the location in the source code where this subprocedure is declared is added as an attribute called `decl_location`. This is important to later on link a call statement to the implementation of the procedure.

Note that subprocedures which have no separate declaration do not have the `decl_location` attribute and are linked by the location of the body itself (as defined by the `location` attribute).

Special attention is needed for subprocedure bodies which override a parent implementation. For these bodies we do not need the local declaration location, but the location where the parent implementation is declared. This is because during compile time it is not known which implementation corresponds to a certain call statement, hence the parent declaration is needed to link call statements to subprocedure implementations.

Table 6.2 shows the translation of a subprocedure via the AST into the intermediate Ada representation.

<pre> 309: 81: overriding procedure Task_Entry (Pick : access Pickup_Process; -: 82: E : Entry_Type; -: 83: Collet : Integer) is </pre>	<pre> SubpBody[81:4-91:19] f_overriding: OverridingOverriding[81:4-81:14] f_subp_spec: SubpSpec[81:15-83:54] f_subp_kind: SubpKindProcedure[81:15-81:24] f_subp_name: DefiningName[81:25-81:35] f_name: Id[81:25-81:35]: Task_Entry f_subp_params: ... </pre>	<pre> <subprocedure decl_location="adat_thetaz-mill. ads:275:4: " location="adat_thetaz-mill- pickup_pos.adb:81:4: " name="Task_Entry"> </pre>
---	---	--

Table 6.2: Subprocedure Body translation; Top: Source code, annotated with coverage information; Bottom-Left: AST as retrieved from LibAdaLang; Bottom-Right: translated XML code complying to the metamodel.

6.6.3. EXPRESSION FUNCTION

A special form of a subprocedure is the expression function. This is a single-line definition of a subprocedure. Other than that, the behaviour is the same compared to the Subprocedure translation.

Because the Expression Function is a subtype of the Subprocedure type, the XML tag name of an expression function is again subprocedure, like the normal subprocedures. In order to indicate the subtype to the EMF framework, an `xsi:type` attribute with value "Expr_Function" is added to the XML node. Table 6.3 shows this translation.

<pre> 548: 64: function Bulk_Taping return Boolean is (Config.General.Ds_Bulk /= None); </pre>	
<pre> ExprFunction[64:4-64:76] f_overriding: OverridingUnspecified[62:80-62:80] f_subp_spec: SubpSpec[64:4-64:39] f_subp_kind: SubpKindFunction[64:4-64:12] f_subp_name: DefiningName[64:13-64:24] f_name: Id[64:13-64:24]: Bulk_Taping f_subp_params: <null> f_subp_returns: SubtypeIndication[64:32-64:39] f_has_not_null: NotNullAbsent[64:31-64:31] f_name: Id[64:32-64:39]: Boolean f_constraint: <null> f_expr: ... </pre>	<pre> <subprocedure location="adat_thetaz-mill- attach_pos.adb:64:4: " name="Bulk_Taping" xsi:type="Expr_function" /> </pre>

Table 6.3: Expression Function translation; Top: Source code, annotated with coverage information; Bottom-Left: AST as retrieved from LibAdaLang; Bottom-Right: translated XML code complying to the metamodel.

6.6.4. TASK BODY

The `Ada_Task_Body` type nodes represent the bodies of the Ada Tasks defined in the source code. Tasks are executed in a separate thread and are the main method in Ada to realize concurrent behaviour.

Because task bodies are initialized once and keep 'running' after that, they have an execution count of 1. This means that we cannot decide whether a task is part of the happy flow or not, hence all tasks are added to the intermediate representation. However, to prevent non-relevant tasks to be present in our eventual model, we omit all source files which does has not have any lines above the execution count threshold.

Table 6.4 shows the translation of a task body via the AST to the Ada model.

1: 612: task body Pickup_Task is	
<pre> TaskBody[612:4-1415:20] f_name: DefiningName[612:14-612:25] f_name: Id[612:14-612:25]: Pickup_Task f_aspects: <null> f_decls: DeclarativePart[612:28-1364:4] ... </pre>	<pre> <task location="adat_thetaz-mill- pickup_pos.adb:612:4: " name=" Pickup_Task"> </pre>

Table 6.4: Task Body translation; Top: Source code, annotated with coverage information; Bottom-Left: AST as retrieved from LibAdaLang; Bottom-Right: translated XML code complying to the metamodel.

6.6.5. ACCEPT STATEMENT

The `Ada_Accept Stmt` type nodes and `Ada_Accept Stmt_With_Stmts` type nodes define rendezvous points of Ada tasks. When an accept statement has sub-statements, the caller is blocked while the statements are executed. If there are no sub-statements in the accept statement, the caller continues execution immediately after the rendezvous has occurred.

During the research it appeared that `gcov` gives no coverage information on statements in task bodies. Therefore only statements which explicitly have execution count equal or below the threshold are removed. Hence, statements with no coverage information are kept.

As with subprocedures (Section 6.6.2), the location where the accept statement is declared is added as a `decl_location` attribute. This information is later used to link accept statements calls to the corresponding accept bodies.

Table 6.5 shows the translation of an accept statement with statements via the AST to the Ada model. Note that the statements contained in the accept statement are not shown here and will be added as child elements in the Ada model.

-: 1367: accept Start do	
<pre> AcceptStmtWithStmts[1367:13-1376:23] f_name: Id[1367:20-1367:25]: Start f_entry_index_expr: <null> f_params: EntryCompletionFormalParams[1367:25-1367:25] f_params: <null> f_stmts: HandledStmts[1367:28-1376:13] ... </pre>	<pre> <statement decl_location="adat_thetaz-mill- pickup_pos.ads:34:7: " location=" adat_thetaz-mill-pickup_pos. adb:1367:13: " name="Start" xsi:type=" Accept"> </pre>

Table 6.5: Accept statement translation; Top: Source code, annotated with coverage information; Bottom-Left: AST as retrieved from LibAdaLang; Bottom-Right: translated XML code complying to the metamodel.

6.6.6. SELECT STATEMENT

The select statement is tightly coupled to the accept statement. By means of a select statement it is possible to have a task accept multiple synchronization calls, i.e., it is open for calls to any of its accept statements. This is modelled by a select element having zero or more ‘Guards’ with each guard containing an Accept statement. Furthermore Accept statements can have an else and an abort element, see Section 9.7 of the Ada reference manual for more information.⁸

6.6.7. CALL EXPRESSION

Due to the internals of the Libadalang AST and the corresponding API, it was found that Call Statements could be best identified by parsing all identifiers in the source code and checking whether these identifiers refer to a subprocedure. This is described in Section 6.6.9. Though, to capture the special Event and Mutex statements as described in Section 6.5, the Call_Expression node type is parsed in order to translate these concepts to the Ada metamodel.

Table 6.6 shows the translation from a SetEvent statement in the source code, through the AST into the intermediate representation. Again also a decl_location attribute is added in order to link the SetEvent, ResetEvent and WaitForEvent procedures later on.

Table 6.7 shows the same translation for the Lock_Mutex statement. Also for the mutexes the decl_location attribute is added in order to link the corresponding Lock and Unlock calls.

280: 335: SetEvent (Xy_Permitted);	
<pre> CallStmt[335:7-335:31] f_call: CallExpr[335:7-335:30] f_name: Id[335:7-335:15]: SetEvent f_suffix: AssocList[335:17-335:29] ParamAssoc[335:17-335:29] f_designator: <null> f_r_expr: Id[335:17-335:29]: Xy_Permitted </pre>	<pre> <statement decl_location="adat_ramdata. ads:20:4: " location="adat_thetaz-mill -pickup_pos.adb:335:7: " name=" Xy_Permitted" xsi:type="SetEvent"/> </pre>

Table 6.6: SetEvent statement translation; Top: Source code, annotated with coverage information; Bottom-Left: AST as retrieved from LibAdaLang; Bottom-Right: translated XML code complying to the metamodel.

⁸<https://docs.adacore.com/live/wave/arm12/html/arm12/arm12-9-7.html>

283: 471: Lock_Mutex (Xy.Lock);	
<pre> CallStmnt[471:7-471:28] f_call: CallExpr[471:7-471:27] f_name: Id[471:7-471:17]: Lock_Mutex f_suffix: AssocList[471:19-471:26] ParamAssoc[471:19-471:26] f_designator: <null> f_r_expr: DottedName[471:19-471:26] f_prefix: Id[471:19-471:21]: Xy f_suffix: Id[471:22-471:26]: Lock </pre>	<pre> <statement decl_location=" adat_wafertable_xy.ads:368:10: " location="adat_wafertable_xy. adb:471:7: " name="Lock" xsi:type=" LockMutex" /> </pre>

Table 6.7: LockMutex statement translation; Top: Source code, annotated with coverage information; Bottom-Left: AST as retrieved from LibAdaLang; Bottom-Right: translated XML code complying to the meta-model.

6.6.8. IF STATEMENT

An if statement contains multiple groups of child statements. The condition part and then part are always present. Furthermore there can be zero or more elsif parts, and lastly there can be a possible else part.

Once an if statement is found during the AST traversal, an if element is added to the Ada model including the location attribute. Then a condition element is added as a child to the if element. All statements present in the condition part are again recursively parsed. The same is done for the then part and the else part of the if statement.

Any present elsif statement is parsed separately, resulting in an elsif element below the if element in the model. Note that an elsif statement, like an if statement, again has a condition part and a then part.

Table 6.8 shows the translation of an if statement including an elsif and else part.

<pre> 33 procedure If_Else_Sample_Function 34 is 35 begin 36 if If_Condition_Stmt = True then 37 If_Callstmt; 38 elsif Elsif_Condition_Stmt then 39 Elsif_Callstmt; 40 else 41 Else_Callstmt; 42 end if; 43 end If_Else_Sample_Function; </pre>	<pre> <statement location="if_else_sample.adb:36:7: " xsi:type="If"> <conditions> <statement location="if_else_sample.adb:36:10: " name="If_Condition_Stmt" target_location=" if_else_sample.adb:3:4: " xsi:type=" Call_statement" /> </conditions> <then> <statement location="if_else_sample.adb:37:10: " name="If_Callstmt" target_location=" if_else_sample.adb:15:4: " xsi:type=" Call_statement" /> </then> <statement location="if_else_sample.adb:38:7: " xsi:type="Elsif"> <conditions> <statement location="if_else_sample.adb:38:13: " name="Elsif_Condition_Stmt" target_location=" if_else_sample.adb:9:4: " xsi:type=" Call_statement" /> </conditions> <then> <statement location="if_else_sample.adb:39:10: " name="Elsif_Callstmt" target_location=" if_else_sample.adb:27:4: " xsi:type=" Call_statement" /> </then> </statement> <else> <statement location="if_else_sample.adb:41:10: " name="Else_Callstmt" target_location=" if_else_sample.adb:21:4: " xsi:type=" Call_statement" /> </else> </statement> </pre>
<pre> IfStmt[36:7-42:14] f_cond_expr: RelationOp[36:10-36:34] f_left: Id[36:10-36:27]: If_Condition_Stmt f_op: OpEq[36:28-36:29] f_right: Id[36:30-36:34]: True f_then_stmts: StmtList[37:10-37:22] CallStmt[37:10-37:22] f_call: Id[37:10-37:21]: If_Callstmt f_alternatives: ElsifStmtPartList[38:7-39:25] ElsifStmtPart[38:7-39:25] f_cond_expr: Id[38:13-38:33]: Elsif_Condition_Stmt f_stmts: StmtList[39:10-39:25] CallStmt[39:10-39:25] f_call: Id[39:10-39:24]: Elsif_Callstmt f_else_stmts: StmtList[41:10-41:24] CallStmt[41:10-41:24] f_call: Id[41:10-41:23]: Else_Callstmt </pre>	

Table 6.8: If statement translation; Top: Source code, annotated with coverage information; Bottom-Left: AST as retrieved from LibAdaLang; Bottom-Right: translated XML code complying to the metamodel.

6.6.9. IDENTIFIER TYPE

As discussed before, call statements are retrieved from the AST by parsing the `Ada_Identifier` node type. Every identifier in the source code is checked whether it refers a node of type `Ada_Classic_Subp_Decl`, `Ada_Subp_Body`, `Ada_Generic_Subp_Instantiation`, `Ada_Expr_Function` or `Ada_Subp_Renaming_Decl`. In that case a `Call_statement` element is added to the Ada model and the location where the identifier references to is added as a `target_location` attribute.

In Table 6.9 the identifier `Task_Entry` is checked for referring to one of the given node types. This results in the `Call_statement` element shown at the bottom right of the table.

<pre>270: 1015: Task_Entry (Pick.Mill.Proc (Pick.Connected_Inspect_P).Itask , 135: 1016: Prepare_TransInspect , Bh_AtTransInsp.Bh_Nr);</pre>	
<pre>CallStmt[1015:16-1016:72] f_call: CallExpr[1015:16-1016:71] f_name: Id[1015:16-1015:26]: Task_Entry f_suffix: AssocList[1015:28-1016:70] ParamAssoc[1015:28-1015:75] f_designator: <null> f_r_expr: DottedName[1015:28-1015:75] ...</pre>	<pre><statement location="adat_thetaz-mill- pickup_pos.adb:1015:16: " name=" Task_Entry" target_location=" adat_thetaz-mill.ads:275:4: " xsi:type ="Call_statement"/></pre>

Table 6.9: Call statement translation; Top: Source code, annotated with coverage information; Bottom-Left: AST as retrieved from LibAdaLang; Bottom-Right: translated XML code complying to the metamodel.

6.6.10. CASE STATEMENT

The case statement follows the same pattern as the if statement with a `case_expr` element for all statements in the Case expression and separate child elements for each case alternative, denoted by a `when` element.

6.6.11. LOOP STATEMENTS

The Ada language supports three kinds of loop statements; the for loop, while loop, and a loop statement without conditions. The loop statement without conditions is translated in a single `Loop` element in the Ada model containing child elements representing the statements within the loop.

The for and while loop both have two intermediate elements, i.e., a `loop_expr` element containing all the statements of the loop condition, and a `loop_stmts` element which contains all the statements of the loop body.

6.6.12. RETURN AND DELAY STATEMENTS

The Return and Delay node types have a 1-to-1 mapping to their corresponding Ada model elements. Note that both these nodes can contain child statements, which are then added as child elements to the parent return or delay element.

6.7. RESULTING MODEL

Applying the transformation explained in the previous sections, the source code's happy flow is translated in to one complete XML file representing the Ada model. Figure 6.3 shows a graphical view on the model in the Eclipse Modelling Framework with important language constructs like tasks, subprocedures, call statements, loops and if statements. At the end of each line in the model, the node type and parent reference is shown between brackets. The corresponding XML 'source code' of this model is given in Appendix A.
























- ▼  platform:/resource/AdaEmfModel/model_instances/Ada_Samples.model
 - ▼  Project Ada2Uppaal (Project)
 - ▼  Package body If_Else_Sample (Package_body@Project.package_body)
 - ▶  Subprocedure If_Condition_Stmt (Subprocedure@Package_body.subprocedure)
 - ▶  Subprocedure Elself_Condition_Stmt (Subprocedure@Package_body.subprocedure)
 -  Subprocedure If_Callstmt (Subprocedure@Package_body.subprocedure)
 -  Subprocedure Else_Callstmt (Subprocedure@Package_body.subprocedure)
 -  Subprocedure Elself_Callstmt (Subprocedure@Package_body.subprocedure)
 - ▼  Subprocedure If_Else_Sample_Function (Subprocedure@Package_body.subprocedure)
 - ▼  If if_else_sample.adb:36:7: (If@Subprocedure.statement)
 - ▼  Elself if_else_sample.adb:38:7: (Elself@Statement.statement)
 - ▶  Statement List (Statement_List@Elself.conditions)
 - ▶  Statement List (Statement_List@Elself.then)
 - ▶  Statement List (Statement_List@if.conditions)
 - ▶  Statement List (Statement_List@if.then)
 - ▶  Statement List (Statement_List@if.else)
 - ▼  Package body Case_Sample (Package_body@Project.package_body)
 - ▶  Subprocedure Case_Condition_Stmt (Subprocedure@Package_body.subprocedure)
 -  Subprocedure Case_Stmt_1 (Subprocedure@Package_body.subprocedure)
 -  Subprocedure Case_Stmt_2 (Subprocedure@Package_body.subprocedure)
 - ▼  Subprocedure Case_Sample_Procedure (Subprocedure@Package_body.subprocedure)
 - ▼  Case case_sample.adb:24:7: (Case@Subprocedure.statement)
 - ▶  Statement List (Statement_List@Case.case_expr)
 - ▶  When case_sample.adb:25:14: (When@Case.when)
 - ▶  When case_sample.adb:26:18: (When@Case.when)
 - ▼  Package body Task_Sample (Package_body@Project.package_body)
 -  Subprocedure Accept_1_Callstmt (Subprocedure@Package_body.subprocedure)
 -  Subprocedure Accept_2_Callstmt (Subprocedure@Package_body.subprocedure)
 - ▶  Subprocedure Task_Sample_Subp (Subprocedure@Package_body.subprocedure)
 - ▼  Task Task_Sample_Body (Task@Package_body.task)
 - ▼  Loop task_sample.adb:23:7: (Loop@Task.statement)
 - ▼  Select task_sample.adb:24:10: (Select@Statement.statement)
 - ▼  Guard task_sample.adb:25:13: (Guard@Statement.statement)
 - ▼  Accept Accept_1 (Accept@Statement.statement)
 -  Call statement Accept_1_Callstmt (Call_statement@Statement.statement)
 - ▶  Guard task_sample.adb:29:13: (Guard@Statement.statement)
 - ▶  Guard task_sample.adb:32:13: (Guard@Statement.statement)
 - ▼  Package body Event_Sample (Package_body@Project.package_body)
 - ▼  Subprocedure Event_Sample_Function (Subprocedure@Package_body.subprocedure)
 -  Rendezvous Accept_1 (Rendezvous@Subprocedure.statement)
 -  Delay event_sample.adb:24:6: (Delay@Subprocedure.statement)
 - ▼  Task Simple_Task (Task@Package_body.task)
 -  Accept Accept_1 (Accept@Task.statement)
 - ▶  Package body Mutex_Sample (Package_body@Project.package_body)

Figure 6.3: The source code's happy flow shown in the Eclipse model viewer.

7

FORMAL MODEL GENERATION

This chapter describes the research done to answer **RQ-3** which was stated as:

*How to automatically translate a partial program written in Ada, as found by answering **RQ-2**, into a formal model that allows the properties found by answering **RQ-1** to be verified?*

Different steps have been taken to come from the intermediate Ada model as described in the previous chapter, to a formal Uppaal model. This chapter describes these steps and the resulting Uppaal models. At the core of the transformation the Eclipse Modelling Framework (EMF) is used as described in Section 2.4.

Section 7.1 explains the main concept of the transformations done. Section 7.2 describes how the linking between call statements and their implementations is realised. Section 7.3 then explains the transformation itself. Section 7.4 finally shows a glimpse of the full generated Uppaal model of our CPS. The repository of our model-to-model transformation definition is available on GitHub¹.

7.1. TRANSFORMATION CONCEPT

The complete transformation from the intermediate Ada model to a formal Uppaal model is done in two steps. First the Ada model is transformed into an Uppaal model in EMF ‘syntax’, secondly this Uppaal-EMF model is transformed into a text format complying to the Uppaal syntax such that Uppaal can open this file. The latter is denoted as the Model-to-Text transformation.

7.1.1. MODEL-TO-MODEL TRANSFORMATION

As discussed in Section 2.4, a model based engineering approach was taken to transform the Ada code to Uppaal ‘code’. As shown in Figure 2.6, for the transformation we need a source metamodel, a target metamodel and a transformation definition.

The source metamodel is the metamodel of the Ada code as developed during this project and presented in Section 6.5. The target metamodel is the metamodel of the Uppaal language as presented by [Schivo et al., 2017], see Figure 7.2.

This section describes the designed transformation definition to have EMF transform our Ada model to an Uppaal model.

¹<https://github.com/jildert17/ada-to-uppaal>

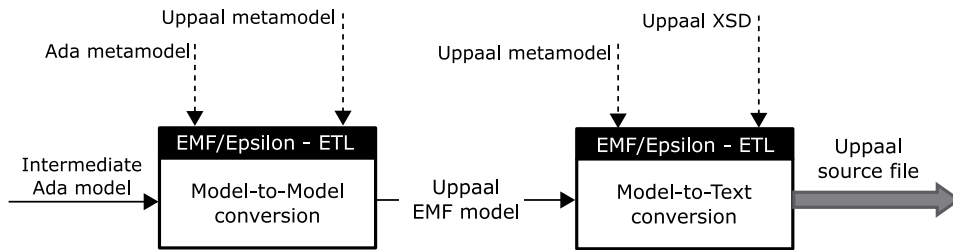


Figure 7.1: Model-2-model and model-2text transformation

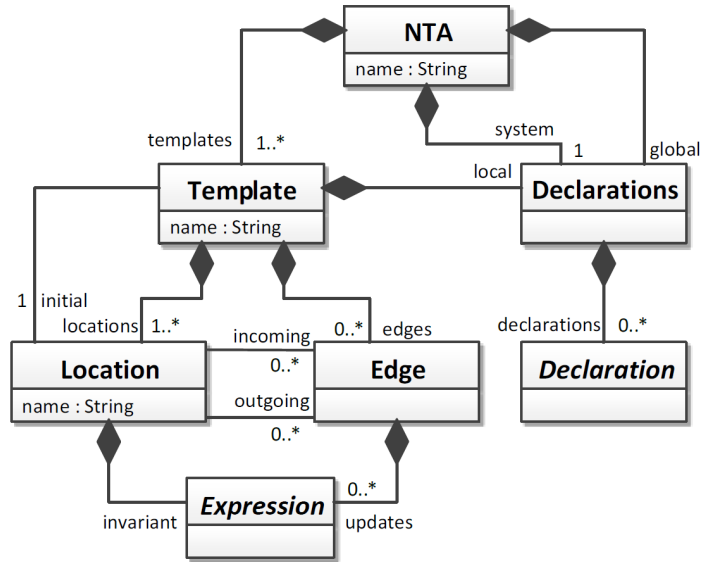


Figure 7.2: Uppaal metamodel; from [Schivo et al., 2017]; Note: Edges can contain select, guard, synchronize and update expressions (not shown in the figure)

Epsilon [Kolovos et al., 2008] is a family of languages specifically designed for model based engineering, which works on top of EMF. For defining model to model transformations, Epsilon provides the Epsilon Transformation Language² (ETL). In ETL, rules can be defined which describe how a certain type of input class should be transformed to one or more output classes.

The transformation definition is based on the transformation of [Yildiz et al., 2017] to translate Java Bytecode into an Uppaal model.

ETL rules can optionally be labelled Greedy and Lazy. Non-labelled rules are automatically applied to all classes which exactly match the given type. In contrast, when a rule is labelled Greedy, that rule is also applied to classes which are child classes of the referenced object. This is useful in our case where we want certain rules to apply to all objects inheriting from the statement type. Where the previous rules are always executed, rules labelled Lazy are only executed when specifically called. This feature is as well used in our transformation and discussed below.

7.1.2. MODEL-TO-TEXT TRANSFORMATION

The model to text transformation concerns the translation of the Uppaal model in EMF context to Uppaal source files. One of the main reasons for choosing the EMF framework for our model transformations was that this model-to-text transformation for Uppaal is already available from [Schivo et al., 2017].

This transformation, like the model to model transformation, uses an ETL file describing the translation from the Uppaal metamodel to the Uppaal source file definition. Uppaal source files are again written in XML and follow a syntax as defined in an XML Schema Definition (XSD). This XSD file serves as the metamodel for the Uppaal source file.

7.2. LINKING CALLS AND IMPLEMENTATIONS

Before the transformation itself is executed, some pre-processing needs to be done in order to prepare the correct linking between call statements and their corresponding implementations. This not only holds for regular subprocedure calls, but also for calls to Accept statements, setting and resetting Events as well as locking and unlocking Mutexes.

7.2.1. SUBPROCEDURE CALLS

During the transformation of subprocedure calls, we need to check whether there is an implementation available in the happy flow for the specific call. If there is no implementation, no further action is required, but in case the implementation is available, the subprocedure call and the implementation need to be linked. Important to notice is that due to inheritance, there can be multiple implementations for one subprocedure call. Because statically it cannot be determined which implementation is the right one, all implementations need to be considered.

The common denominator between a subprocedure call and its implementation is the unique location of the subprocedure definition, therefore this is used as the ‘connecting dot’. In order to find the connecting dot at the transformation step, a hash map is created during the pre-processing step. This map contains the subprocedure call statement as the key, and a list of the possible implementations as its value. The list is created by iterating

²<https://www.eclipse.org/epsilon/doc/etl/>

over all call statements in the happy flow which call a subprocedure. For each call statement the declaration location is retrieved and checked against the declaration locations of all subprocedure implementations, every match is added to the list of possible implementations.

A sample of this hash map is given in Listing 7.1.

```
Map {
  Call_statement [location=case_sample.adb:24:12: , name=Case_Condition_Stmt ,
    target_location=case_sample.adb:3:4: ]
    ->Sequence {Subprocedure [name=Case_Condition_Stmt , location=case_sample.adb
      :3:4: , decl_location=null , ]},
  Call_statement [location=case_sample.adb:26:32: , name=Case_Stmt_2 ,
    target_location=case_sample.adb:15:4: ]
    ->Sequence {Subprocedure [name=Case_Stmt_2 , location=case_sample.adb:15:4: ,
      decl_location=null , ]}}
```

Listing 7.1: An example of the subprocedure hashmap with the call statements as its key and a list of possible implementations as the value

7.2.2. ACCEPTS, EVENTS AND MUTEX LINKING

Like for the subprocedure calls, hash maps are created as well for accept statements, setting and resetting events and locking and unlocking mutexes. These maps are simpler than the subprocedure maps because there are no multiple possible implementations.

The hash maps for these constructs all look the same with their unique declaration location as the key and the corresponding object as its value.

Events The population of the events hash map is based on all `SetEvent` and `ResetEvent` statements. Furthermore, for each event, an Uppaal template is created having a set and an unset location, representing the two states of the event. The semantics of our event concept allows an event to be always set or reset, even if it was already in the set or unset state respectively. Furthermore a `WaitForEvent` can be called on an event which will be blocking as long as the event is in the unset state, and gets unblocked as soon as the event is in the set state.

To account for the semantics, three synchronization channels are created along with the template. One channel for setting the event, one for resetting the event and one for waiting on the event. An example of a generated event template with its channels is shown in Figure 7.3.

It should be noted that we have chosen to make the set location the initial location. This was needed in order to make the machine cycle actually ‘run’ while having a model of the happy flow only because some events are only set during initialization.

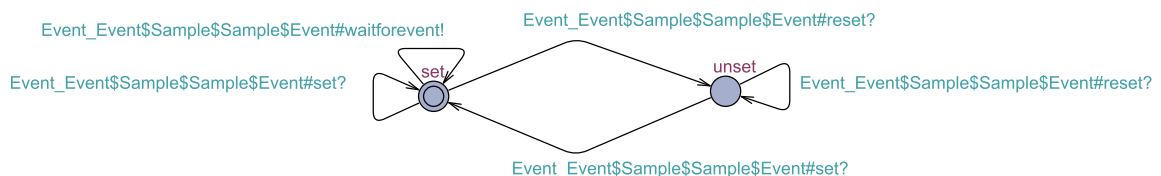


Figure 7.3: A generated Event Template

Mutexes For mutexes we chose to only use the `UnlockMutex` statements to populate the hash map. This is done so to guarantee that a mutex is always unlocked in the happy-flow code. Assume two locking calls for the same mutex are present in the happy flow, but the unlock call is not part of the happy flow (e.g. it is in one of the support packages), then the happy-flow model will contain a deadlock while this is not the case in the real system.

For each unique mutex added to the hash map, a template is created containing an unlocked and locked state with two edges going from one to the other. Both edges have a synchronization channel in order to be able to put the mutex either in locked or in unlocked state, see also Figure 7.4.

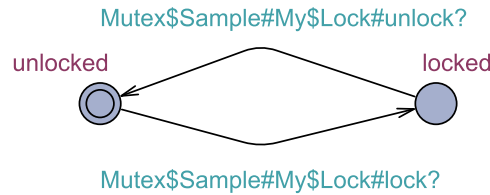


Figure 7.4: A generated Mutex Template

7.3. TRANSFORMATION STEP-BY-STEP

For readability, this section explains and shows the complete transformation (model-to-model + model-to-text) in one go per Ada construct, hence the intermediate Uppaal-EMF model instance is not shown.

7.3.1. SUBPROCEDURE TRANSFORMATION

As a first step in the transformation, a rule is defined which translates all Ada Subprocedures into Uppaal Templates. This rule calls a recursive function which transforms all statements in the subprocedure to Uppaal states, these transformations are described in Section 7.3.3. Furthermore the Uppaal templates are given synchronization points on the entry and return edges such that these templates can be ‘called’ from other templates, just like the original subprocedures. In order to create the required synchronization channels, another rule is defined which translates all Subprocedures to two Uppaal Synchronization channels, one for the call and one for the return.

7.3.2. TASK TRANSFORMATION

Tasks and Subprocedures are much alike. Both get translated into their own Uppaal template. The main difference between the two is that a task does not have call and return channels. No call is needed to start the task, the moment the model is ‘started’, all tasks are started as well.

7.3.3. STATEMENT TRANSFORMATION

For each statement present in a subprocedure, a recursive function is called which handles all elements derived from the abstract Statement node type. Due to the ‘recursiveness’, also all nested statements get translated. This is the main part of the transformation and effectively builds a Control Flow Graph from the intermediate Ada representation. Below it is described in detail how each different statement sub type is translated.

CALL STATEMENT

Call statements are handled by the lazy defined rule ‘ada_CallStatement2uppaal_location’. Because this rule is defined lazy, it is only executed when called from the recursive function.

This rule always creates a calling location and a returning location. Only when there is an implementation for the specific called procedure, also a ‘waiting’ location is created. In that case a call synchronization is added to the edge from the calling to the waiting state and a return synchronization to the edge from waiting to returning.

```
package body Call Stmt_Sample is
```

```
function Callee return Boolean
is
begin
  return True;
end Callee;
```

```
procedure Caller
is
  Foo : Boolean;
begin
  Foo := Callee;
end Caller;
```

```
end Call Stmt_Sample;
```

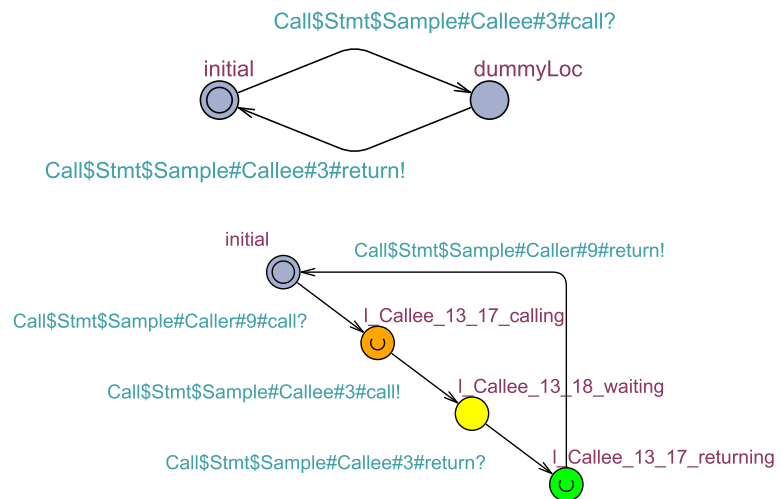


Figure 7.5: Call Statement transformation; Left: Original source code, Top right: Uppaal Callee template; Lower right: Uppaal Caller template

Important to note is that there can be multiple possible implementations, for example when multiple child classes implement a function which is defined abstract in a parent class. This means that statically it is not known which of the implementations should be called by the call statement. Therefore it is possible that there are multiple waiting states in the Uppaal model for one call statement. Uppaal will choose one of the possible implementations in a nondeterministic manner.

Figure 7.5 shows an example of a subprocedure calling another subprocedure. Both subprocedures are translated to Uppaal templates which synchronize with each other by means of a Call and a Return channel. Because the Callee function has no statements except for `return True` a dummy location is added automatically.

IF AND ELSIF STATEMENT

An important aspect of the If and Elsif statement transformation is the deduction of the correct control flow. Regardless of the branch chosen in the statement, first always the conditions of the If statement will be executed. After this either the statements in the then-part are executed, or the conditions of a possible Elsif statement, or the statements present in a possible Else-part. Note that when an Elsif part and an Else part are present, first the conditions of the Elsif are executed again before the statements in the Else are executed.

Figure 7.6 shows an Uppaal template of an If-Elsif-Else statement. Here the control flow of the conditions and the different branches is visualized. Uppaal makes a nondeterministic choice for one of the possible branches, but note that of course only the branches part of the happy flow are in the intermediate representation.


```

procedure If_Else_Sample_Function
is
begin
  if If_Condition_Stmt = True then
    If_Callstmt;
  elsif Elsif_Condition_Stmt then
    Elsif_Callstmt;
  else
    Else_Callstmt;
  end if;
end If_Else_Sample_Function;

```

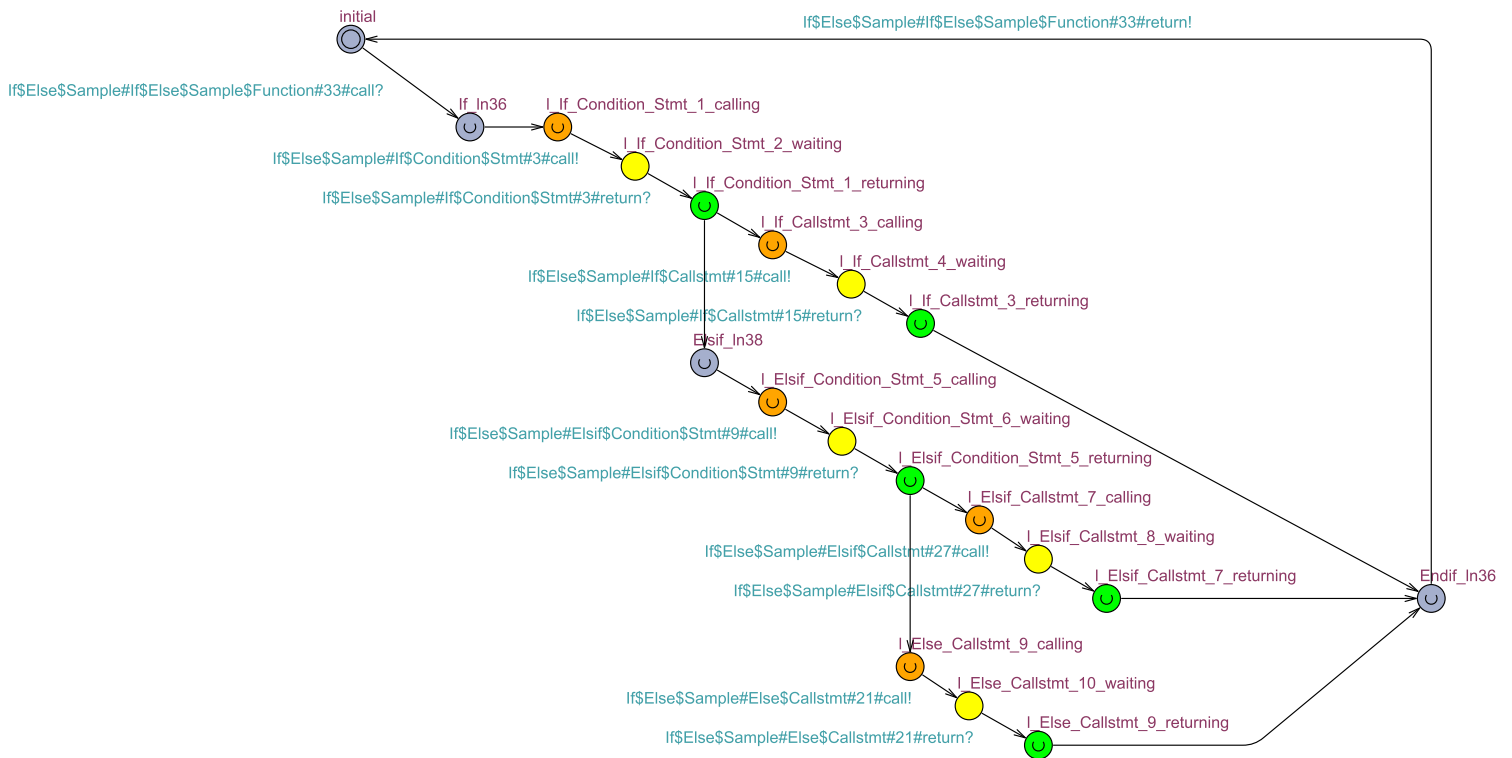


Figure 7.6: If-Else-Else statement transformation

CASE STATEMENTS

Case statements are modelled in the same way as If statements, first the case condition is executed and then one of the possible branches can be chosen by Uppaal. Again only the branches part of the happy flow are available.

EVENTS

As briefly introduced in Section 6.5 and Section 7.2.2, our event construct is a kind of global signal having two possible states: *Set* and *Unset*. Related to this construct, there are three functions which can be called on an event: `SetEvent`, `ResetEvent` and `WaitForEvent`. By means of this concept concurrent tasks can synchronize with each other, e.g. task *a* waits for signal *s* to be set by task *b*. The behaviour of these statements is rather straightforward and is modelled in Uppaal by a separate template per event with two locations representing the state of the event, either *set* or *unset*.

The `Set` statement is translated into two Uppaal locations with an edge in between containing a send synchronization on the `set` channel. In the same fashion the `Reset` statement is translated into two locations with a send synchronization on the `reset` channel. Note that the event template can always receive a set or a reset synchronization, so the former statements are never blocking.

The `WaitForEvent` statement is also translated into two locations connected by an edge, but in this case it has a *receive* synchronization on the `waitForEvent` channel. Because the event template can only send this synchronization in the `set` state, the `WaitForEvent` statement is blocked until the event is set. A piece of sample code translated into an Event template and related call statements is shown in Figure 7.7.

Unlike most other locations in our Uppaal model, the first location of the `WaitForEvent` statement is not labelled as Urgent because Uppaal should be able to wait in this location until the moment the event is set. It should be noted here that for a model without time, like our model, this has no influence, but in this way the model is at least already prepared for this.

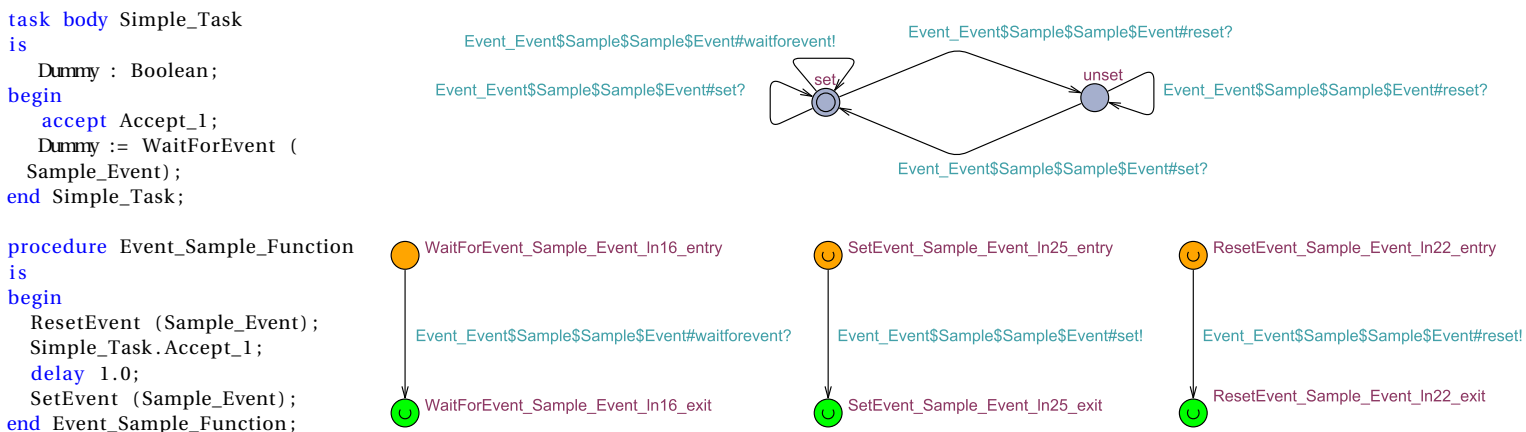


Figure 7.7: `SetEvent`, `ResetEvent`, and `WaitForEvent` example; left the original code; right the generated Uppaal models including the template representing the state of the event shown at the top.

MUTEXES

The second custom construct in the codebase as discussed earlier are mutexes. This concept is modelled in Uppaal by a simple template with two locations representing a *locked*

state and an *unlocked* state. When a process wants to ‘grab’ a lock while the mutex-template is already in the lock state, the calling process is blocked until the mutex is released and transitioned to the unlock state.

The mutex concept as defined in our code base does not prevent task *a* to unlock a mutex locked by task *b*. The Uppaal representation of the mutex mimics this behaviour. Subsequently our mutex concept cannot lock an already locked mutex (this results in an exception). In our model this behaviour is also covered but this will not result in an exception, instead it will block the ‘calling’ task until the mutex has been unlocked again.

As explained in Section 7.2.2, during the pre-processing step of the model transformation, a hash map is generated from all occurrences of `MutexUnlock` statements. This map is used to determine whether the lock is in the happy flow at all and, if so, to find the correct synchronization channel to either lock or unlock the mutex.

Figure 7.8 shows an example of a piece of Ada code translated into the corresponding Uppaal mutex-template and call statements.

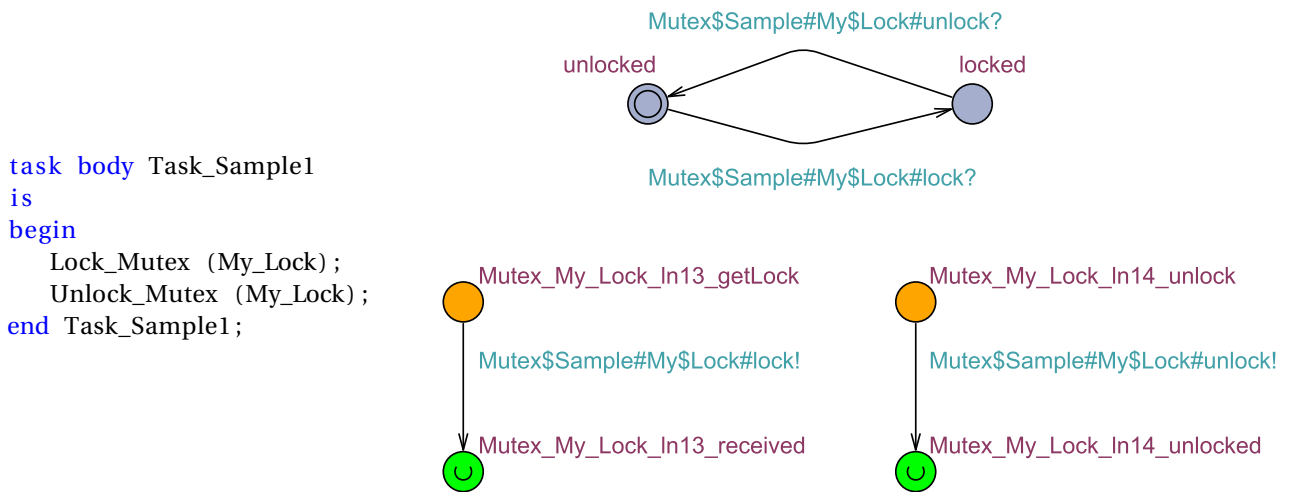


Figure 7.8: Uppaal example showing a Mutex Template and the Lock and Unlock calls

LOOPS

Loop statements are transformed in a straightforward manner where first the loop conditions are executed, and then the body of the loop. Finally from the last location two edges are added, one to the first location of the loop and one to an exit location.

One exception is made for the loop statement without conditions. This kind of loop is considered to run indefinitely and does not have an edge to the exit location.

In the current version of the tool, no information from the loop conditions is used to determine the number of times a loop is executed. It can be imagined to use coverage information to estimate this number. This is left as future work.

RENDEZVOUS STATEMENTS

Rendezvous statements follow the same approach as call statements. A calling and returning location are added for each rendezvous, and when a corresponding accept statement is available in the accepts hash map, a waiting location is added with the appropriate call and return synchronization channels.

Figure 7.9 shows a simple subprocedure with one rendezvous statement synchronizing with the `Accept_1` entry point of a task.

```

procedure Task_Sample_Subp
is
begin
  Task_Sample_Body.Accept_1;
end Task_Sample_Subp;

```

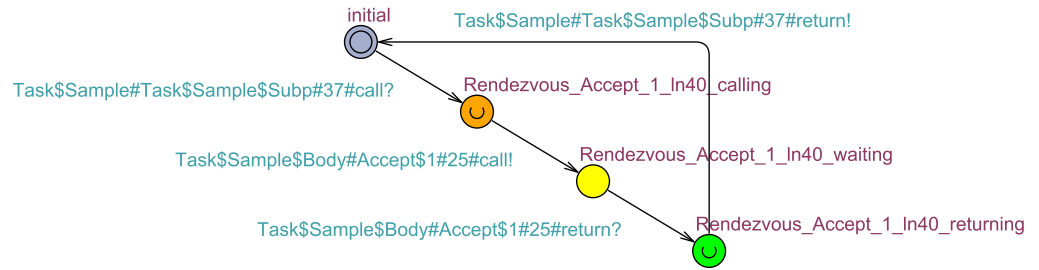


Figure 7.9: Subprocedure synchronizing with the Accept_1 entry point of the task shown in Figure 7.10; Left: original Ada code; Right: the generated Uppaal template

ACCEPT STATEMENTS

Accept statements are translated into two locations, a magenta location with postfix ‘calling’ and a dark blue location with postfix ‘returning’. Accept statements can be ‘called’ by rendezvous statements. Therefore a receiving synchronization channel is added on the edge to the first location, and a sending synchronization on the edge to the second location.

Remember from Section 6.6.5 that accept statements can have child statements but not necessarily. In the former case the caller blocks until all child statements have been executed by the task. In the latter case the caller can immediately continue. This difference is shown in Figure 7.10 where the first accept statement has a child statement and the second accept statement does not.

SELECT STATEMENTS

Figure 7.10 furthermore shows a select statement which make it possible for callers to either synchronize with Accept_1 or with Accept_2. The delay statement shown in the code is translated into a location in the Uppaal model, however the effective delay is currently not incorporated. Note that in this case the required delay of 10 seconds is obvious, but typically an expression is given evaluating to some number which is much harder to find statically.

7.4. THE FULL UPPAAL MODEL

At this point, all the parts needed to create a full Uppaal model from our CPS source code are in place. The Cobertura file contains the coverage data for 562 source files, parsing this file and generating the intermediate representation by our Coverage_Filter tool takes around 1.5 minutes. The model to model transformation by Eclipse takes around 2.5 minutes and the model to text transformation takes 1 minute.

Figure 7.11 shows a screenshot of the resulting Uppaal model, furthermore some numbers on the size of the model are given in Table 7.1.

```

task body Task_Sample_Body
is
begin
loop
select
accept Accept_1 do
Accept_1_Callstmt;
end Accept_1;
or
accept Accept_2;
Accept_2_Callstmt;
or
delay 10;
end select;
end loop;
end Task_Sample_Body;

```

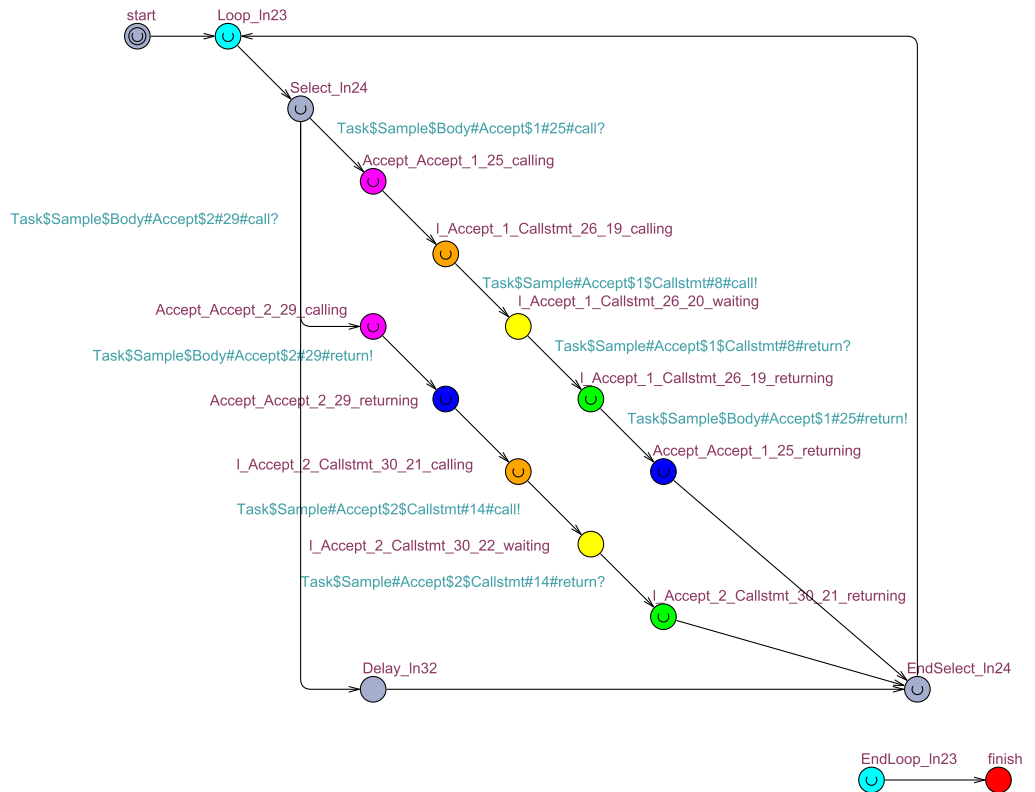


Figure 7.10: Task sample with infinite loop and accept statements

Uppaal model statistics	
Number of task templates	38
Number of event templates	15
Number of mutex templates	14
Number of subprocedure templates	757
Total number of templates	824
Total number of locations	11817

Table 7.1: Some numbers of the generated Uppaal model

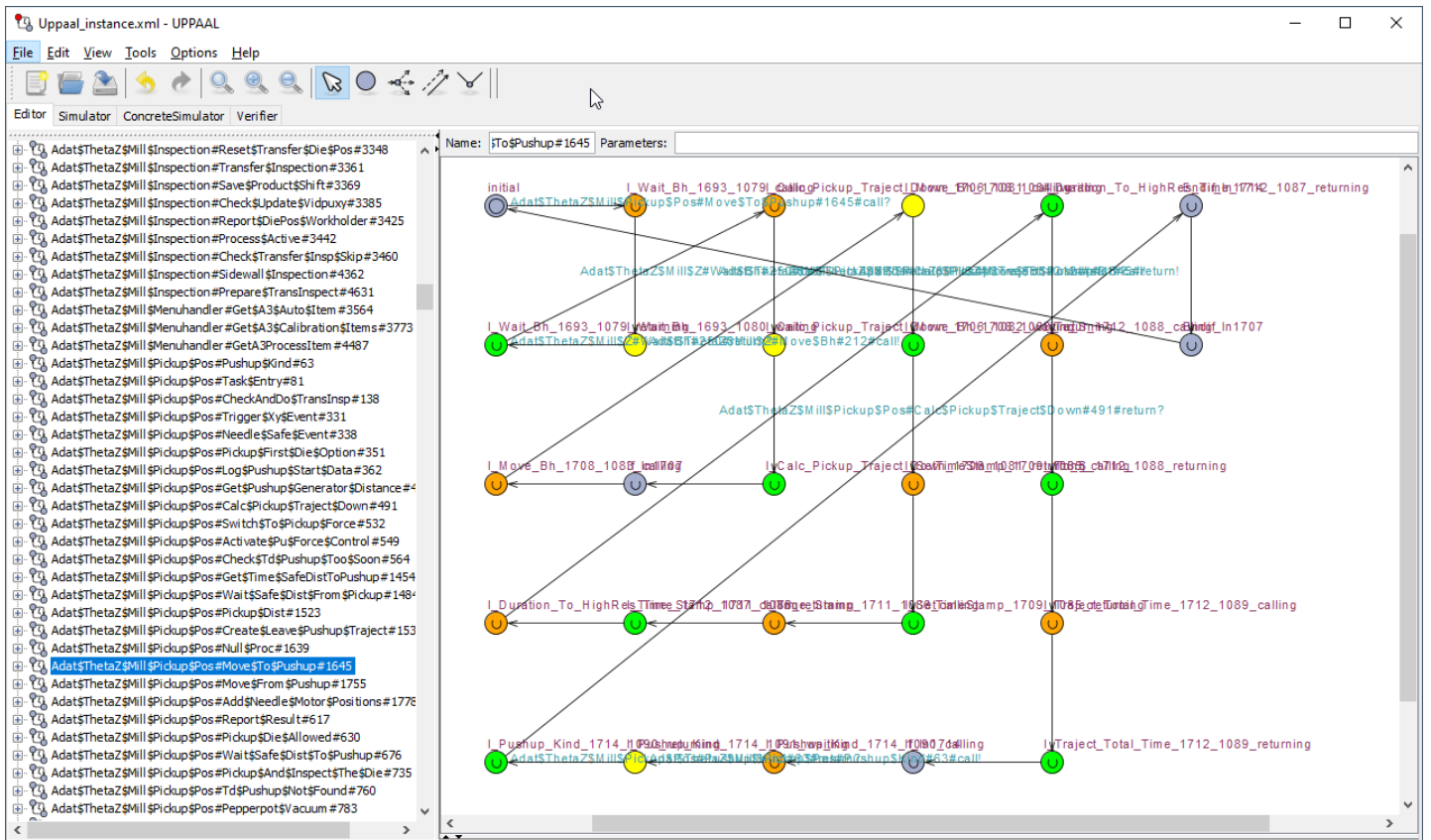


Figure 7.11: A screenshot of the resulting Uppaal model of the happy flow of our CPS.

8

FORMAL VERIFICATION

This chapter brings the work presented so far in practice. In other words, can we now use the generated formal model? The related research question (RQ-4) is stated as:

What results can be found from verifying the properties defined by answering RQ-1 on the model, found from answering RQ-3?

From Chapter 5 it is known that many interesting properties are related to a physical state of the CPS. However, augmenting the model with physical state information has not been part of this research as the main focus was on creating a method and corresponding tooling to automatically generate a formal model from source code. Therefore only the deadlock and reachability properties from RQ-1 are actually relevant to our model. This chapter gives the results of these properties.

8.1. DEADLOCK

Uppaal supports a specific query to look for deadlock situations in the model. This query is written as `A [] not deadlock`, i.e. for all states in all paths there is no deadlock.

As our model only models the happy flow of the CPS, absence of deadlock does not guarantee that the complete system will contain no deadlock. The other way around however, when deadlock does exist in the happy-flow model, indicates that the complete system will probably also contain a deadlock. It is mentioned ‘probably’ because our model contains nondeterministic behaviour which can lead to deadlock situations not actually being present in the real system.

In order to prove the absence of deadlock in our model, Uppaal has to look through the complete state space which might be a challenge in terms of memory and computing power. To prove that the property does not hold however only requires one ‘witness’, therefore this might be an easier task for Uppaal.

Executions for checking the deadlock property on the model resulted in swap file overflows after several hours without having found a deadlock situation in the model.

8.2. REACHABILITY ANALYSIS

As discussed in Chapter 5, reachability properties are not very interesting on a model of the happy-flow code as it is expected that all states are reachable. For model validation

purposes it can however be very interesting to verify reachability properties on the model in order to check whether our model actually does what we expect, and how much time is needed to verify this.

To specify a reachability property, we use the logic operator $EF(p)$, which defines the property that a path through the model exist which eventually satisfies p . In Uppaal this is written as $E\langle\rangle p$, where p can be some kind of property, but can also just represent a certain location in the model, which then translates to: *It is possible to reach a certain location.*

RESULTS

A couple of reachability properties checking whether the main tasks of the CPS reach the state in which a start trigger was received, have been verified on the model. These properties can be verified by Uppaal in a matter of seconds using the ‘Random Depth First’ search order.

It should be noted that sometimes executions of the same properties do not give an answer within minutes. This behaviour could be related to both the random search order as well as nondeterministic behaviour of the model. The results are summarized in Table 8.1. In Chapter 10 the results from RQ-4 are discussed further and different approaches to improve the results are given as recommendations and future work.

8.3. LIVENESS ANALYSIS

In Chapter 5 we discussed two categories of liveness properties: one related to the physical state of the machine and the other one related the liveness of the machine cycle, e.g. is it always possible to return to a defined state in the cycle?

The first category cannot be checked on our current model, but we tried to verify a property of the second category on the model. As a first step in this direction a simple *leads to* property is defined from the start trigger location used in the previous section, to its very next location. In Uppaal the leads to operator is written as $-->$, the complete query is shown in Table 8.1.

We found that this seemingly straightforward property is not satisfied for our current model. By again using the random depth first order, it takes between a few seconds and a couple of minutes to find this not-satisfied result.

From the witness trace of Uppaal we found that this property is not satisfied because the model can enter an infinite loop in one of the tasks, causing the other tasks to stall. This is possible because we have no notion of time in our current model and Uppaal can therefore decide to stay in one of these loops.

Property	Execution time	Satisfied?
$E\langle\rangle$ Adat\$ThetaZ\$Mill\$Pickup\$PosPickup_Task#612. Accept_Start_1367_calling	~2 sec.	Yes
$E\langle\rangle$ Adat\$ThetaZ\$Mill\$Attach\$PosAttach_Task#472. Accept_Start_1213_calling	~2 sec.	Yes
$(..)$.Accept_Start_1367_calling $-->$ $(..)$.l_SetTimeStamp_1368_4026_calling	~2 sec.	No

Table 8.1: Results of the reachability and liveness properties verified on our Uppaal model

9

DISCUSSION

In this chapter we will have a look at the validity of our research, the overall results, and the strengths and limitations of the work.

9.1. VALIDATION

The validation of our method is divided over three steps. The first step validates whether the coverage tooling indeed shows high execution numbers for the happy-flow code, and low numbers for initialization and error handling code. Secondly the translation from the source code to the intermediate Ada representation is validated. As a third step the model-to-model transformation Uppaal is validated. Furthermore this section discusses both scalability and generality in a validation context.

CODE COVERAGE VALIDATION

Below three code snippets are shown which represent three ‘coverage scenarios’ in our code, i.e., initialization code, error handling code, and happy-flow code. A correct coverage result for each of these scenarios is considered to be a valid prove that the coverage results for the whole code base are correct.

Listing 9.1 shows two subprocedures from our code base which are reported to be executed once. This corresponds to our expectation as these `Open()` and `Reset()` functions are only executed during initialization.

```
1: 53: function Open return Die_Transfer_Root_Class is
1: 54:     The_Dtrf : constant Die_Transfer_A3_Access := new Die_Transfer_A3;
-: 55: begin
1: 56:     ...

1: 965: function ThetaZ_Reset (Dtrf : access Die_Transfer_A3) return Boolean
-: 966: is
-: 967: begin
1: 968:     ...
```

Listing 9.1: Code snippets from the gcov tooling showing initialization code which have line execution counts of 1

Listing 9.2 shows two subprocedures of which we know are executed each machine cycle, this is correctly reflected by the line count of 281.

```

281: 985:  function Send_Collet_Down return Boolean is
-: 986:      Result : Boolean;
-: 987:  begin
281: 988:      ...

281: 735:  procedure Pickup_And_Inspect_The_Die is
-: 736:  begin
281: 737:      ...

```

Listing 9.2: Code snippets from the gcov tooling showing subprocedures which are executed each machine cycle

Finally, Listing 9.3 shows a subprocedure which again is executed each machine cycle, but it is shown that the exception handler is not executed (#####), resulting in this code being filtered out of the happy flow representation. This again corresponds to the expectation that error code is not executed in a happy-flow execution of the CPS.

```

280: 838:  function Wait_Pushup return Boolean
-: 839:  is
-: 840:  begin
280: 841:      Wait_Bh (Bh.Motor);
280: 842:      SetTimeStamp (Td_Pu_Too_Soon, Edge_Yellow);
-: 843:      return True;
-: 844:  exception
#####: 845:      when E : others =>
#####: 846:          Report_Motion_Exception (Bh.Motor, E);
-: 847:      return False;
-: 848:  end Wait_Pushup;

```

Listing 9.3: Code snippet from the gcov tooling showing a subprocedure which is executed each cycle, but the exception handler is never executed

INTERMEDIATE REPRESENTATION VALIDATION

The second step validates the translation of the source code to the intermediate Ada representation. This is done by carefully checking the translation of each Ada construct. Chapter 6 has already shown these translations per construct, like for call statements, if statements, etc. The correct translation for each construct is considered as valid prove that the complete translation is correct.

Besides the correct translation, it is also verified whether indeed only the happy-flow parts are translated, such that the intermediate representation does not contain non-happy-flow parts.

As an example, Listing 9.4 shows a subprocedure which is itself part of the happy flow (line count of 351), but of which one if statement is not executed in the happy flow. During the translation from source code, via the AST, to the intermediate representation, this if statement is omitted and therefore not part of the eventual model. Listing 9.5 shows the intermediate Ada representation of the same subprocedure, not containing the first if statement.

```

351: 2972:  function Wait_T (Tz : access Transfer_Config) return Boolean
-: 2973:  is
351: 2974:      Result : Boolean := True;
-: 2975:  begin
351: 2976:      if Tz.TM /= null then
#####: 2977:          Result := (Wait_T (Tz.TM));

```

```

-: 2978:      end if;
351: 2979:      if Tz.AM /= null and then Result then
351: 2980:          Result := Wait_T (Tz.AM);
-: 2981:      end if;
351: 2982:      if Tz.PM /= null and then Result then
351: 2983:          Result := Wait_T (Tz.PM);
-: 2984:      end if;
351: 2985:      return Result;
-: 2986:  end Wait_T;

```

Listing 9.4: Code snippet from the gcov tooling showing a subprocedure with parts in the happy-flow and parts in the non-happy-flow

```

<subprocedure decl_location="adat_thetaz.adb:96:4: " location="adat_thetaz-mill.
adb:2971:4: " name="Wait_T">
<statement location="adat_thetaz-mill.adb:2979:7: " xsi:type="If">
<then>
<statement location="adat_thetaz-mill.adb:2980:20: " name="Wait_T"
target_location="adat_thetaz-mill.adb:850:4: " xsi:type="Call_statement"/>
</then>
</statement>
<statement location="adat_thetaz-mill.adb:2982:7: " xsi:type="If">
<then>
<statement location="adat_thetaz-mill.adb:2983:20: " name="Wait_T"
target_location="adat_thetaz-mill.adb:850:4: " xsi:type="Call_statement"/>
</then>
</statement>
<statement location="adat_thetaz-mill.adb:2985:7: " xsi:type="Return"/>
</subprocedure>

```

Listing 9.5: Intermediate Ada representation of the code shown in Listing 9.4; it can be observed that the complete if statement from lines 2976 to 2978 is removed as it is not part of the happy-flow, while the rest of the lines is correctly translated.

MODEL GENERATION VALIDATION

The resulting Uppaal model of our CPS as presented at the end of Chapter 7 is way too big to verify whether the source code is correctly modelled. Therefore validation of the automatic model generation is carried out by translating small pieces of Ada code containing a specific language construct. This is done by carefully checking all the transformations which have been presented in Section 7.3. Especially for constructs like If-Elsif-Else statements a lot of care is taken in verifying the correct flow of the different statements contained in the If-Elsif-Else branches.

Figure 9.1 repeats Figure 7.6 from Chapter 7, showing how a piece of sample code containing an If-Elsif-Else construct is translated into an Uppaal template. The Uppaal template shows the control flow through the statement, which correctly mimics the control flow through the original source code. This check is done for each and every Ada construct which is translated into the Uppaal model.

As a second model validation step, the model checker itself is used. Statements in the code of which it is known are executed in the happy flow are translated into reachability properties and checked on the model. This is shown by the reachability properties described in Chapter 8.

```

procedure If_Else_Sample_Function
is
begin
  if If_Condition_Stmt = True then
    If_Callstmt;
  elsif Elself_Condition_Stmt then
    Elself_Callstmt;
  else
    Else_Callstmt;
  end if;
end If_Else_Sample_Function;

```

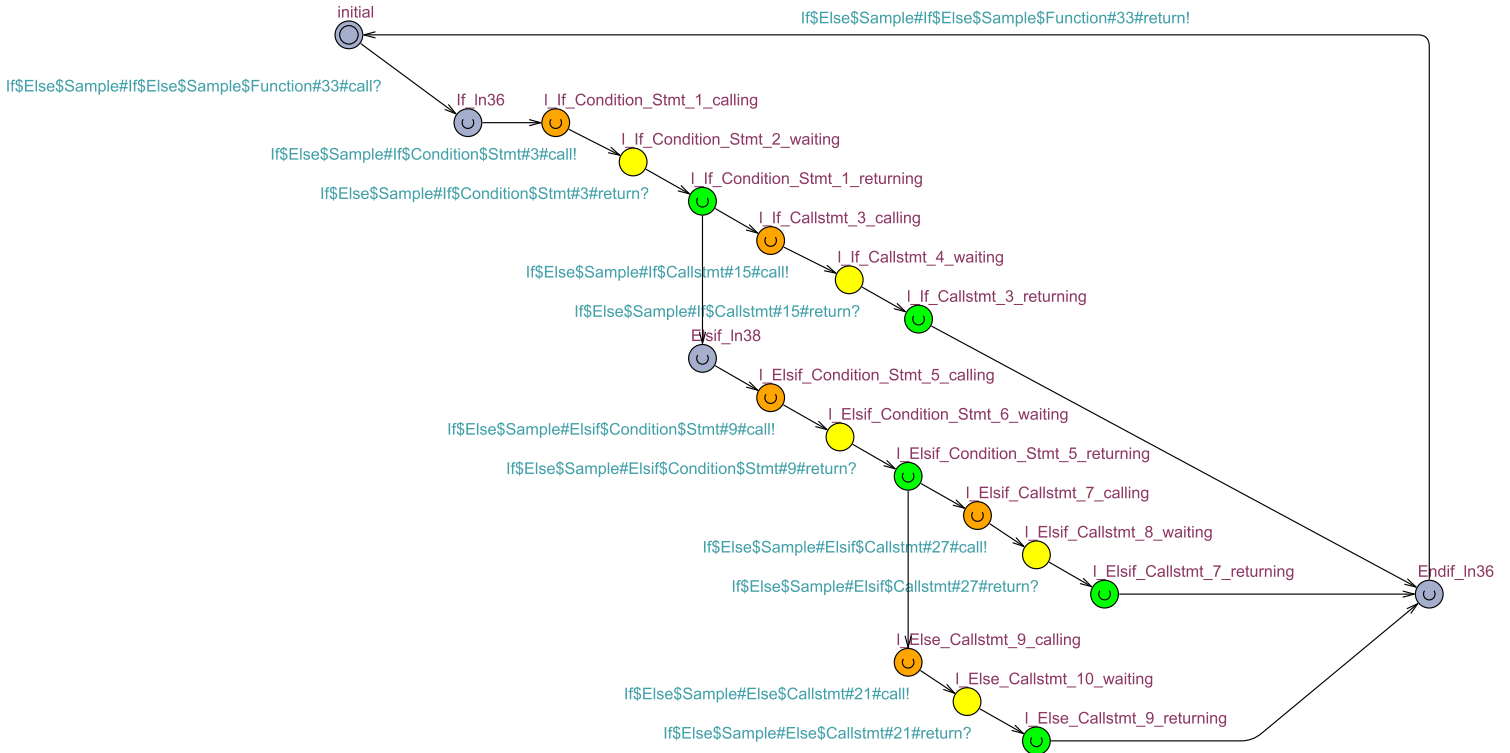


Figure 9.1: If-Elself-Else statement transformation

VALIDATING SCALABILITY

The goal of our research was to conquer the state space explosion which often limits the scalability of formal verification techniques. We did this by means modelling only the happy flow part of our CPS instead of the complete code base of the machine.

From Chapter 8 we learned that our model allows to verify some simple reachability properties, but memory limits prevent more complex properties to be checked on the model. More research is required in order to tell what category of properties are able to be verified on the current model and what categories are not. For properties which are still not able to run on the model it should be tried to limit the model size even further, for example by reducing the modelled call depth.

VALIDATING GENERALITY

The tooling developed during this research is specifically written for the Ada programming language, however, the method of finding the happy flow of a CPS and translate this code via a model-to-model transformation into an Uppaal model is generally applicable to other languages. Most languages will have tooling available to do code coverage analysis as well as tooling to extract an abstract syntax tree from the code, hence an intermediate model of the happy-flow code can be generated based on our designed method. From this point it should be a relative easy step to adapt our model-to-model transformations for the new language. Note however that custom concepts like our events and mutexes will require adaptations to other languages.

9.2. RESULTS

Our research has resulted in a method and corresponding tooling to extract the happy-flow code of a cyber-physical system and translate this code into a formal model in a highly automatic fashion. By modelling only the happy flow, a huge part of the original code base was sliced out, resulting in a smaller model than obtained otherwise.

By using a model-based engineering approach we designed a relatively light-weight transformation definition to translate an intermediate representation of the code into an Uppaal model. This transformation definition does contain aspects of the Ada programming language but can be easily adapted for other programming languages.

Some simple reachability properties have successfully been verified on the formal model, but a liveness property was not satisfied, mainly due to the lack of clocks in the model. Furthermore some more complex properties and the ‘no deadlock’ property still run into the problem of the state explosion. The current model is not suitable to verify domain-related properties corresponding to a physical state of the machine as no direct relation between the model and the state of the machine is present.

9.3. STRENGTHS & LIMITATIONS

This section summarizes the strengths and limitations of the work presented in this thesis. Options to reduce the limitations are given in the next chapter when discussing the recommendations and future work.

STRENGTHS

- By using code coverage tooling, a method is found to automatically extract the happy-flow code of a cyber-physical system.
- The Uppaal model representing the happy-flow code is generated in a fully automatic fashion, minimizing the risk of human mistakes.
- The developed method can be applied to other programming languages, only small changes to the tooling should be necessary to make it work for different languages.

LIMITATIONS

- As the model represents the happy flow, any property verified on the model does not necessarily hold on a potential model of the complete code base. This is inherent to

the under-approximation as applied in this work.

Furthermore due to nondeterministic behaviour currently present in our model, the model is also an over-approximation of the system. This can result in situations where the model violates certain properties while the actual system will not violate these properties.

- Although Uppaal is very suitable to add timing aspects to the formal model, this is not yet included in our work. This is deliberately left as future work.
- Loop conditions present in the code are currently not considered during the transformation. Often loop conditions depend on run-time information which is not present statically, but it can be imagined that coverage information is used to deduce the number of times a loop is executed during the happy-flow execution. More specifically the number of times a loop is executed during the run should be divided by the number of times the machine cycle is executed during the run.

Adding this information will reduce nondeterministic behaviour of the model and therefore results in a better approximation of the real system. Also it prevents tasks to stay in a loop forever, causing other tasks to stall.

- All templates in Uppaal, representing subprocedures and tasks from the source code, are instantiated exactly once. This means that when two parallel tasks call the same subprocedure at the same time, only one task will proceed and the other one has to wait. As we currently do not consider time, this behaviour is not considered as a problem. Although it can potentially cause a deadlock situation when the specific sub-procedure needs the task which is actually blocked to respond.
- No direct relation between the state of the code and the physical state of the machine is present. Therefore domain properties are not easily checked on our formal model.
- The designed tooling is capable of translating most common Ada constructs but some more advanced constructs of the language are not covered yet and are left for future work.

10

CONCLUSIONS & FUTURE WORK

The beginning of this chapter summarizes our research by answering our sub-questions and the main research question. After the summary, recommendations on our work and directions for future work are discussed. The recommendations are meant as improvements where the future work section focuses on extensions on the current work.

10.1. SUMMARY

Our main research question was stated as:

To what extent is it possible to prove formal properties on the 'happy flow' code of a real-time, concurrent cyber-physical system with a high ratio of repetitive tasks?

In order to give an answer to the main research question, we will first answer the four sub-questions.

Sub-question 1 *Having only a model of the 'happy flow' code of a CPS, what properties of the system can be verified and can reveal interesting information for domain experts?*

We found that although a model of the happy flow does not represent the complete source code, different categories of properties are still very useful to be verified on a happy-flow model, e.g. safety and liveness properties. On the other hand we found that many properties require some notion of the physical state of the machine. This information is not present when a formal model is automatically generated from the source code.

Sub-question 2 *How can the code of the machine's 'happy flow' be extracted automatically, using run-time information?*

A method and corresponding tooling has been developed to automatically extract the happy-flow code by means of code coverage tooling. This is proven to be effective and straightforward.

Sub-question 3 *How to automatically translate a partial program written in Ada, as found by answering RQ-2, into a formal model that allows the properties found by answering RQ-1 to be verified?*

Applying the model-driven engineering approach allowed us to write a model-to-model transformation at a high abstraction level and to make use of existing work for generating Uppaal models. Tooling to generate a working Uppaal model from the filtered source code has been created.

Sub-question 4 *What results can be found from verifying the properties defined by answering RQ-1 on the model, found from answering RQ-3?*

So far we managed to verify a couple of simple reachability properties using the model. The current work has not revealed new information from the code base.

Based on the answers to the four sub questions, our answer to the main research question is: It is possible to *some* extent to prove formal properties on a model of the ‘happy flow’ of a cyber-physical system, but more work is required to apply the method in practical situations.

10.2. RECOMMENDATIONS

From the limitations and the conclusions drawn from our research, different recommendations can be given to improve to designed method and tooling. This section focuses solely on improvements where the next section on future work should be considered as extensions to our method.

ANALYZE SCALABILITY

Our research was mainly focused on finding a method and creating a tool to generate a formal model of the happy-flow code of a CPS. A formal model has indeed been generated, but this model does not allow us to verify many different properties without running into the memory limits of our computer. Therefore it is recommended to further analyze and validate the scalability of the current tooling, i.e., which categories of properties can be checked on the current model and which categories can not? When less source code would be used as input will the model allow more complex properties to be checked? etc.

DEDUCT LOOP CONDITIONS

The loops in our model have no conditions, i.e., the decision to execute a loop another time is nondeterministic. By means of carefully looking at the coverage information it should be possible to automatically deduct the number of times a loop is executed per machine cycle. This information can then be used to adapt the Uppaal model to be no longer non-deterministic at this point. Making the model more deterministic will result in a better approximation of the system, but will also significantly reduce the state space and therefore improve scalability.

MODEL TIME-OUTS AND DELAYS

In our Ada code it is possible to define how long a task is allowed to wait for an event to be set and how long a task should wait before another task is able to synchronize. This behaviour is not modelled in our Uppaal model. Although it is not expected that a happy-flow execution contains time-outs, this information can be retrieved from the source code. By adding clocks and clock guards to the specific Uppaal templates this behaviour can be mimicked in the model.

NUMBER OF SUBPROCEDURE INSTANTIATIONS

At this moment all templates in our model have one instance while Uppaal allows templates to be instantiated more than once. Therefore two tasks calling the same sub-procedure (template) at the same time will cause one task to be blocked until the other task is ready. In order to avoid increasing the model size unnecessarily, it is not a good idea to just increase the number of instances. Therefore it is proposed to create some sort of monitor which generates a warning when a sub-procedure is called more than once at the same time. This can be used to incrementally increase the number of instantiations of the template representing this sub-procedure, i.e. until no more warnings occur.

10.3. FUTURE WORK

During our research different directions for extensions arose. Below a non-exhaustive list of possible extensions is given.

EXPLORE OTHER USE CASES

At this moment we only used the model to run properties on it to prove certain aspects of the system. One can however also think of other use cases of a formal model of the source code. It might for example be an option to generate a model twice, before and after a refactoring action of the source code. Typically a refactoring should give no functional changes, e.g. the interface of a component stays the same but the implementation changes. When these interfaces are available as well in the formal model, model based testing can be applied to verify that from the interface point of view the two implementations act the same.

PHYSICAL STATE AUGMENTATION

It is already stated multiple times that a limitation of our generated model is that no physical state information of the CPS is available. Adding this kind of information will allow many more interesting (formal) properties to be defined. This step is however considered to take a big effort because physical state information is not readily available. Options to add this information are for example to add it manually to the generated model, or by adding annotations to the source code and take this into account during the model transformation.

ADD TIMING INFORMATION

By adding a profiling step on top of the coverage tooling, it is possible to retrieve execution times for every function call. When this information is added to the Uppaal model by means of clocks and clock guards, things like timing analysis and critical path analysis become available. Two points of attention here are: 1) adding timing information to the model will considerably increase the state space; and 2) it should be verified that the profiler will not harm the real time behaviour of the system.

LOCAL VERIFICATION

In order to reduce the state space further and be able to verify more complex properties, one can isolate specific parts of the model. In order to verify such an isolated part, a context should however be available which can supply the required signals for the isolated part

to keep 'running'. Such a context can be modelled using another automaton running in parallel with the 'part-under-test'. Creating this automaton does require system knowledge on the expected signals. However, it may be possible to automatically derive the context by looking at the interface of the isolated part.

OBSERVERS

Instead of writing complex queries in a temporal logic, one can also define observer automata which run in parallel with the model under test. The observer models a certain desired behaviour or sequence of actions and listens to outputs of the main model. As long as the expected signals are received from the main model, the observer proceeds, but when an unexpected signal is received, the observer branches to an error state, indicating that the main model does not comply to the desired behaviour.

11

REFLECTION

We made it! After 3,5 years of hard work I am proud to have finalized my Software Engineering masters. I am thankful to my employer to gave me the change to do this masters next to my normal work. Many parts of the learned material was directly applicable to my day to day job which is also a big compliment to the curriculum of the Software Engineering master of the Open University.

But, I have to admit that the graduation project did take quite some discipline to make it to the end while also having a job and raising our newborn son. My initial planning was based on the same hours I spent during the courses of the masters, but it appeared that the workload of the courses was not comparable to that of the graduation project. This mainly accounts for the run out of 4 months on top of the initially planned 8 months.

Another reason for the run out were the struggles with the GNATCoverage tooling at the start of the project. I first tried to make the open source version of GNATCoverage, written for Linux, work on a Windows environment which didn't worked out. A couple of weeks later I received a trial license for the Windows version of the tool. After I finally managed to instrument the code with this tooling and retrieved the coverage information, it appeared that GNATCoverage does not reveal the line counts of the source code, but only tells whether a line is executed or not. At that moment I stepped back to the older gcov tool, which I considered to be deprecated, but giving me the coverage information I was looking for right away. The bottom line here is that the next time I should not spend so much time trying to get such a tool up and running, but should rather look for other directions.

On the result of the work it feels a little unsatisfying that I have not been able to really prove some interesting properties on the formal model. It is good to mention though that the fourth research question, which was about running properties on the generated model, was only added to the research proposal in a final stage with a remark that this would be a nice to have result when time allowed for it. The main focus of my research was on the method and tool creation to generate a formal model of the happy-flow code of a cyber-physical system.

I have learned a lot from this graduation project, both process wise and content wise. Especially subjects like AST processing and model based engineering were new and interesting topics to me which I believe will be very useful in my future career. I want to thank my supervisors Dr. Jacques Verriet and Dr. Stefano Schivo for all the time and effort they spent on guiding and supporting me through the process and giving me both constructive and positive feedback on my work!

BIBLIOGRAPHY

- H. Agrawal and J. R. Horgan. Dynamic Program Slicing. *ACM SIGPLAN Notices*, 25(6):246–256, June 1990. ISSN 0362-1340. doi: 10.1145/93548.93576. 14
- R. Alur and D. L. Dill. A Theory of Timed Automata. *Theoretical Computer Science*, 126: 183–235, 1994. 6
- G. Behrmann, A. David, K. Larsen, J. Håkansson, P. Pettersson, W. Yi, and M. Hendriks. Uppaal 4.0. In *Third International Conference on the Quantitative Evaluation of Systems, QEST 2006*, pages 125–126, Jan. 2006. doi: 10.1109/QEST.2006.59. 7, 24
- P. Bjesse. What is formal verification? *ACM SIGDA Newsletter*, 35(24):1–es, Dec. 2005. ISSN 0163-5743. doi: 10.1145/1113792.1113794. 1, 5
- J. B. Bladen, D. Spenhoff, and S. J. Blake. Ada semantic interface specification (ASIS). In *Proceedings of the Conference on TRI-Ada '91: Today's Accomplishments; Tomorrow's Expectations*, TRI-Ada '91, pages 6–15, New York, NY, USA, Dec. 1991. Association for Computing Machinery. ISBN 978-0-89791-445-1. doi: 10.1145/126551.126552. 13
- I. T. Bowman, M. W. Godfrey, and R. C. Holt. Extracting Source Models from Java Programs: Parse, Disassemble, or Profile? <http://plg.math.uwaterloo.ca/~migod/papers/1999/paste99.pdf>, Sept. 1999. 12
- O. Burkart, D. Caucal, F. Moller, and B. Steffen. CHAPTER 9 - Verification on Infinite Structures. In J. A. Bergstra, A. Ponse, and S. A. Smolka, editors, *Handbook of Process Algebra*, pages 545–623. Elsevier Science, Amsterdam, Jan. 2001. ISBN 978-0-444-82830-9. doi: 10.1016/B978-044482830-9/50027-8. 6
- A. Burns and A. J. Wellings. How to verify concurrent Ada programs: The application of model checking. In *Proceedings of the Ninth International Workshop on Real-Time Ada, IRTAW '99*, pages 78–83, New York, NY, USA, June 1999. Association for Computing Machinery. ISBN 978-1-58113-177-2. doi: 10.1145/329607.334743. 12
- A. Burns, B. Dobbing, and T. Vardanega. Guide for the use of the Ada Ravenscar Profile in high integrity systems. *ACM SIGAda Ada Letters*, XXIV, Apr. 2003. doi: 10.1145/997119.997120. 13
- A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV 2: An OpenSource Tool for Symbolic Model Checking. In *Computer Aided Verification*, Lecture Notes in Computer Science, pages 359–364, Berlin, Heidelberg, 2002. Springer. ISBN 978-3-540-45657-5. doi: 10.1007/3-540-45657-0_29. 7
- E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Logics of Programs*, Lecture Notes in Computer Science, pages 52–71, Berlin, Heidelberg, 1982. Springer. ISBN 978-3-540-39047-3. doi: 10.1007/BFb0025774. 5

- E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, Apr. 1986. ISSN 0164-0925. doi: 10.1145/5397.5399. **5**
- E. M. Clarke, W. Klieber, M. Nováček, and P. Zuliani. Model Checking and the State Explosion Problem. In B. Meyer and M. Nordio, editors, *Tools for Practical Software Verification*, volume 7682, pages 1–30. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. ISBN 978-3-642-35745-9 978-3-642-35746-6. doi: 10.1007/978-3-642-35746-6_1. **1, 5**
- J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Păsăreanu, Robby, and Hongjun Zheng. Bandera: Extracting finite-state models from Java source code. In *Proceedings of the 2000 International Conference on Software Engineering. ICSE 2000 the New Millennium*, pages 439–448, June 2000. doi: 10.1145/337180.337625. **12**
- J. M. Faria, J. Martins, and J. S. Pinto. An Approach to Model Checking Ada Programs. In *Reliable Software Technologies – Ada-Europe 2012*, volume 7308, pages 105–118. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. ISBN 978-3-642-30597-9 978-3-642-30598-6. doi: 10.1007/978-3-642-30598-6_8. **13**
- J. Hatcliff, J. Corbett, M. Dwyer, S. Sokolowski, and H. Zheng. A Formal Study of Slicing for Multi-threaded Programs with JVM Concurrency Primitives. In *Static Analysis*, volume 1694, pages 1–18. Springer Berlin Heidelberg, Berlin, Heidelberg, 1999a. ISBN 978-3-540-66459-8 978-3-540-48294-9. doi: 10.1007/3-540-48294-6_1. **13**
- J. Hatcliff, M. B. Dwyer, and H. Zheng. Slicing Software for Model Construction. In *Higher-Order and Symbolic Computation*, pages 105–118, 1999b. **14**
- G. Holzmann and M. Smith. An automated verification method for distributed systems software based on model extraction. *IEEE Transactions on Software Engineering*, 28(4): 364–377, Apr. 2002. ISSN 0098-5589. doi: 10.1109/TSE.2002.995426. **13**
- G. J. Holzmann. *Design and validation of computer protocols*. Prentice-Hall, 1991. **7**
- S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM SIGPLAN Notices*, 23(7):35–46, June 1988. ISSN 0362-1340. doi: 10.1145/960116.53994. **13**
- D. S. Kolovos, R. F. Paige, and F. A. C. Polack. The Epsilon Transformation Language. In A. Vallecillo, J. Gray, and A. Pierantonio, editors, *Theory and Practice of Model Transformations*, Lecture Notes in Computer Science, pages 46–60, Berlin, Heidelberg, 2008. Springer. ISBN 978-3-540-69927-9. doi: 10.1007/978-3-540-69927-9_4. **39**
- B. Korel and J. Laski. Dynamic Program Slicing. *Information Processing Letters*, 29(3):9, 1988. **14**
- National Science Foundation. Cyber-Physical Systems (CPS). <https://www.nsf.gov/pubs/2021/nsf21551/nsf21551.htm>, Jan. 2021. **1**
- A. Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science (SFCS 1977)*, pages 46–57, Oct. 1977. doi: 10.1109/SFCS.1977.32. **5**

- M. O. Rabin and D. Scott. Finite Automata and Their Decision Problems. *IBM Journal of Research and Development*, 3(2):114–125, Apr. 1959. ISSN 0018-8646. doi: 10.1147/rd.32.0114. **6**
- A. Rodrigues da Silva. Model-driven engineering: A survey supported by the unified conceptual model. *Computer Languages, Systems & Structures*, 43:139–155, Oct. 2015. ISSN 1477-8424. doi: 10.1016/j.cl.2015.06.001. **11**
- S. Schivo, B. M. Yildiz, E. Ruijters, C. Gerking, R. Kumar, S. Dziwok, A. Rensink, and M. Stoelinga. How to Efficiently Build a Front-End Tool for UPPAAL: A Model-Driven Approach. In *Dependable Software Engineering. Theories, Tools, and Applications*, volume 10606, pages 319–336. Springer International Publishing, Cham, 2017. ISBN 978-3-319-69482-5 978-3-319-69483-2. doi: 10.1007/978-3-319-69483-2_19. **11, 13, 24, 37, 38, 39**
- J. Silva. A vocabulary of program slicing-based techniques. *ACM Computing Surveys*, 44(3): 1–41, June 2012. ISSN 0360-0300, 1557-7341. doi: 10.1145/2187671.2187674. **14**
- N. Silva, N. Moreira, S. Melo de Sousa, and S. Broda. A Tool for Automatic Model Extraction of Ada/SPARK Programs, 2011. **13**
- H. Wayne. Why Don't People Use Formal Methods? <https://www.hillelwayne.com/post/why-dont-people-use-formal-methods/>, Jan. 2019. **1**
- M. Weiser. Program Slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, July 1984. ISSN 1939-3520. doi: 10.1109/TSE.1984.5010248. **14**
- B. M. Yildiz, A. Rensink, C. Bockisch, and M. Aksit. A Model-Derivation Framework for Software Analysis. *Electronic Proceedings in Theoretical Computer Science*, 244:217–229, Mar. 2017. ISSN 2075-2180. doi: 10.4204/EPTCS.244.9. **13, 24, 39**



ADA INTERMEDIATE MODEL

```
<?xml version="1.0" encoding="utf-8"?>
<Project name="Ada2Uppaal" xmi:version="2.0" xmlns="adaModel" xmlns:xmi="http://www.omg.org/XMI" xmlns:xsi
="http://www.w3.org/2001/XMLSchema-instance">
  <package_body location="if_else_sample.adb:1:1: " name="If_Else_Sample">
    <subprocedure location="if_else_sample.adb:3:4: " name="If_Condition_Stmt">
      <statement location="if_else_sample.adb:6:7: " xsi:type="Return"/>
    </subprocedure>
    <subprocedure location="if_else_sample.adb:9:4: " name="Elsif_Condition_Stmt">
      <statement location="if_else_sample.adb:12:7: " xsi:type="Return"/>
    </subprocedure>
    <subprocedure location="if_else_sample.adb:15:4: " name="If_Callstmt"/>
    <subprocedure location="if_else_sample.adb:21:4: " name="Else_Callstmt"/>
    <subprocedure location="if_else_sample.adb:27:4: " name="Elsif_Callstmt"/>
    <subprocedure location="if_else_sample.adb:33:4: " name="If_Else_Sample_Function">
      <statement location="if_else_sample.adb:36:7: " xsi:type="If">
        <conditions>
          <statement location="if_else_sample.adb:36:10: " name="If_Condition_Stmt" target_location="
if_else_sample.adb:3:4: " xsi:type="Call_statement"/>
        </conditions>
        <then>
          <statement location="if_else_sample.adb:37:10: " name="If_Callstmt" target_location="
if_else_sample.adb:15:4: " xsi:type="Call_statement"/>
        </then>
        <statement location="if_else_sample.adb:38:7: " xsi:type="Elsif">
          <conditions>
            <statement location="if_else_sample.adb:38:13: " name="Elsif_Condition_Stmt" target_location="
if_else_sample.adb:9:4: " xsi:type="Call_statement"/>
          </conditions>
          <then>
            <statement location="if_else_sample.adb:39:10: " name="Elsif_Callstmt" target_location="
if_else_sample.adb:27:4: " xsi:type="Call_statement"/>
          </then>
        </statement>
        <else>
          <statement location="if_else_sample.adb:41:10: " name="Else_Callstmt" target_location="
if_else_sample.adb:21:4: " xsi:type="Call_statement"/>
        </else>
      </statement>
    </subprocedure>
  </package_body>
  <package_body location="case_sample.adb:1:1: " name="Case_Sample">
    <subprocedure location="case_sample.adb:3:4: " name="Case_Condition_Stmt">
      <statement location="case_sample.adb:6:7: " xsi:type="Return"/>
    </subprocedure>
    <subprocedure location="case_sample.adb:9:4: " name="Case_Stmt_1"/>
    <subprocedure location="case_sample.adb:15:4: " name="Case_Stmt_2"/>
    <subprocedure location="case_sample.adb:21:4: " name="Case_Sample_Procedure">
      <statement location="case_sample.adb:24:7: " xsi:type="Case">
        <case_expr>
```

```

        <statement location="case_sample.adb:24:12: " name="Case_Condition_Stmt" target_location="
case_sample.adb:3:4: " xsi:type="Call_statement"/>
    </case_expr>
    <when location="case_sample.adb:25:14: ">
        <statement location="case_sample.adb:25:27: " name="Case_Stmt_1" target_location="case_sample.
adb:9:4: " xsi:type="Call_statement"/>
    </when>
    <when location="case_sample.adb:26:18: ">
        <statement location="case_sample.adb:26:32: " name="Case_Stmt_2" target_location="case_sample.
adb:15:4: " xsi:type="Call_statement"/>
    </when>
</statement>
</subprocedure>
</package_body>
<package_body location="task_sample.adb:1:1: " name="Task_Sample">
    <subprocedure location="task_sample.adb:8:4: " name="Accept_1_Callstmt"/>
    <subprocedure location="task_sample.adb:14:4: " name="Accept_2_Callstmt"/>
    <task location="task_sample.adb:20:4: " name="Task_Sample_Body">
        <statement location="task_sample.adb:23:7: " xsi:type="Loop">
            <statement location="task_sample.adb:24:10: " xsi:type="Select">
                <statement location="task_sample.adb:25:13: " xsi:type="Guard">
                    <statement decl_location="task_sample.adb:4:7: " location="task_sample.adb:25:13: " name="
Accept_1" xsi:type="Accept">
                        <statement location="task_sample.adb:26:16: " name="Accept_1_Callstmt" target_location="
task_sample.adb:8:4: " xsi:type="Call_statement"/>
                    </statement>
                </statement>
                <statement location="task_sample.adb:29:13: " xsi:type="Guard">
                    <statement decl_location="task_sample.adb:5:7: " location="task_sample.adb:29:13: " name="
Accept_2" xsi:type="Accept"/>
                    <statement location="task_sample.adb:30:13: " name="Accept_2_Callstmt" target_location="
task_sample.adb:14:4: " xsi:type="Call_statement"/>
                </statement>
                <statement location="task_sample.adb:32:13: " xsi:type="Guard">
                    <statement location="task_sample.adb:32:13: " xsi:type="Delay"/>
                </statement>
            </statement>
        </statement>
    </task>
    <subprocedure location="task_sample.adb:37:4: " name="Task_Sample_Subp">
        <statement location="task_sample.adb:40:24: " name="Accept_1" target_location="task_sample.adb:4:7:
" xsi:type="Rendezvous"/>
    </subprocedure>
</package_body>
<package_body location="event_sample.adb:3:1: " name="Event_Sample">
    <statement location="event_sample.adb:5:28: " name="Create_Event" target_location="general_events.
ads:11:4: " xsi:type="Call_statement"/>
    <task location="event_sample.adb:11:4: " name="Simple_Task">
        <statement decl_location="event_sample.adb:8:7: " location="event_sample.adb:15:7: " name="Accept_1"
xsi:type="Accept"/>
        <statement decl_location="event_sample.adb:5:4: " location="event_sample.adb:16:16: " name="
Sample_Event" xsi:type="WaitForEvent"/>
    </task>
    <subprocedure location="event_sample.adb:19:4: " name="Event_Sample_Function">
        <statement decl_location="event_sample.adb:5:4: " location="event_sample.adb:22:6: " name="
Sample_Event" xsi:type="ResetEvent"/>
        <statement location="event_sample.adb:23:18: " name="Accept_1" target_location="event_sample.
adb:8:7: " xsi:type="Rendezvous"/>
        <statement location="event_sample.adb:24:6: " xsi:type="Delay"/>
        <statement decl_location="event_sample.adb:5:4: " location="event_sample.adb:25:6: " name="
Sample_Event" xsi:type="SetEvent"/>
    </subprocedure>
</package_body>
<package_body location="mutex_sample.adb:3:1: " name="Mutex_Sample">
    <statement location="mutex_sample.adb:5:23: " name="Create_Mutex" target_location="eln-mutexes.
ads:15:4: " xsi:type="Call_statement"/>
    <task location="mutex_sample.adb:10:4: " name="Task_Sample1">
        <statement decl_location="mutex_sample.adb:5:4: " location="mutex_sample.adb:13:7: " name="My_Lock"
xsi:type="LockMutex"/>
        <statement decl_location="mutex_sample.adb:5:4: " location="mutex_sample.adb:14:7: " name="My_Lock"

```



```

    xsi:type="UnlockMutex" />
</task>
<task location="mutex_sample.adb:17:4: " name="Task_Sample2">
  <statement decl_location="mutex_sample.adb:5:4: " location="mutex_sample.adb:20:7: " name="My_Lock"
xsi:type="LockMutex"/>
  <statement decl_location="mutex_sample.adb:5:4: " location="mutex_sample.adb:21:7: " name="My_Lock"
xsi:type="UnlockMutex" />
</task>
</package_body>
<package_body location="call_stmt_sample.adb:1:1: " name="Call Stmt_Sample">
  <subprocedure location="call_stmt_sample.adb:3:4: " name="Callee">
    <statement location="call_stmt_sample.adb:6:7: " xsi:type="Return"/>
  </subprocedure>
  <subprocedure location="call_stmt_sample.adb:9:4: " name="Caller">
    <statement location="call_stmt_sample.adb:13:14: " name="Callee" target_location="call_stmt_sample.
adb:3:4: " xsi:type="Call_statement" />
  </subprocedure>
</package_body>
</Project>

```

Listing A.1: Intermediate representation of a sample program written in Ada containing the most frequent language constructs