

MASTER THESIS
SOFTWARE SCIENCE



RADBOUD UNIVERSITY

Using SysML for Model-Based Testing

Author:

Toine Hulshof

s1005649

t.hulshof@ru.nl

First supervisor/assessor:

dr. ir. G.J. Tretmans

tretmans@cs.ru.nl

jan.tretmans@tno.nl

Daily supervisor:

Timo Catoire

timo.catoire@vanderlande.com

Second assessor:

dr. M.T.W. Schuts

mathijs.schuts@ru.nl

December 18, 2022

Abstract

Model-Based Testing (MBT) is a black-box software testing technique where a System Under Test (SUT) is tested against an abstract model which describes the required behaviour. This testing technique uses the model to automatically generate test cases which can be used to thoroughly check the correctness of the behaviour of the SUT. Therefore, MBT is a viable testing option for increasingly complex systems where the traditional testing work grows exponentially.

Systems Modelling Language (SysML) is a semi-formal system architecture modelling language for system engineering based on the Unified Modelling Language (UML 2). SysML is used to make complex systems understandable for technical and non-technical people while being more precise than natural language.

In this thesis we research the feasibility of formalizing SysML Activity Diagrams for MBT. We provide a translation algorithm from activity diagrams to behavioural models that can be used for MBT. We also state the limitations of this approach. We apply this algorithm on Vanderlandes automatic storage system ADAPTO which is our SUT. Using our findings, we compose a recommendation for Vanderlande regarding the use of MBT. Finally, we provide a step-by-step guide for the implementation of the formalization algorithm for ADAPTO specifically.

Contents

1	Introduction	3
1.1	Problem Description	3
1.2	Research Question	4
1.2.1	Subquestions	4
1.3	Research Method	5
1.4	Outline	5
2	Background	6
2.1	Model-Based Testing	6
2.1.1	Labelled Transition System	6
2.1.2	Trace Equivalence	9
2.1.3	<i>ioco</i>	9
2.1.4	<i>uioco</i>	10
2.1.5	Symbolic Transition System	11
2.1.6	Model-Based Testing	12
2.1.7	Model-Based Testing Tools	13
2.2	SysML	14
2.2.1	Diagrams	15
2.3	Related Work	16
2.3.1	FormaSig	16
2.3.2	Activity Diagrams	18
2.3.3	SysML Activity Diagrams to TORXAKIS Translation Approach	28
2.4	ADAPTO	33
2.4.1	Testing	35
2.4.2	SysML at Vanderlande	37
2.5	Conclusion	38
3	Translation from SysML to Model-Based Testing	40
3.1	Translation Source	40
3.2	Translation Target	40
3.2.1	Labelled Transition System	41
3.2.2	Symbolic Transition System	41
3.2.3	mCRL2	42
3.2.4	TORXAKIS	42
3.2.5	Axini	43
3.3	Activity Diagram Translation Enhancement	44

3.4	Translation Algorithm	45
3.5	Reductions	53
3.6	Discussion	54
4	Case Study on ADAPTO	56
4.1	System Under Test	56
4.2	Adapter	57
4.2.1	Step Definitions	57
4.2.2	Granularity	57
4.3	First Attempts	58
4.3.1	Gherkin Tests	59
4.3.2	Applying Translation Procedure	60
4.4	Preprocessing Activity Diagrams	61
4.4.1	Preprocessing Steps	61
4.5	Application of the Translation Algorithm	63
4.5.1	Preprocessing	64
4.6	Results	64
4.6.1	Encoding Activity Diagrams	64
4.6.2	Encoding Activity Diagrams	65
4.6.3	Using the Resulting Model for MBT	66
4.7	Conclusion	67
4.7.1	Domain Knowledge	67
5	Translation Algorithm Implementation Guide	69
5.1	Step-by-step approach	69
5.1.1	Encoding Activity Diagrams	69
5.1.2	Reduction	71
5.1.3	Translating to TORXAKIS	71
5.2	Conclusion	72
6	Conclusion & Discussion	73
6.1	Discussion	74
6.2	Recommendation	75
7	Future Work	77
A	Derivation Rules Activity Calculus	82
B	Adapter	84
C	Coffee Machine XML Code	86

Chapter 1

Introduction

1.1 Problem Description

Testing is the most important part of the software development lifecycle to discover and fix bugs[17]. Guaranteeing that a safety-critical piece of software functions as intended becomes exponentially more challenging as the complexity of the system increases. This is because only certain combinations of inputs to a system might cause trouble. Increasing complexity also implies that the cost of ensuring that a complex software product works as intended increases exponentially[24]. When the time and costs of testing exceeds the amount that is budgeted for, a trade-off needs to be made between the quality of the product and the testing costs.

Model-Based Testing (MBT) tries to solve this problem. MBT is a software testing technique that tests the behaviour of its System Under Test (SUT) using a model. This model is used to automatically generate test cases. These test cases are executed on the SUT and are compared against the expected output. When using MBT, test cases are automatically generated. This implies that MBT can reach all states while testing without having to manually write exhaustive test cases, even when the complexity of the SUT increases[27]. To use the mathematical concept of MBT for a real world application, an MBT tool needs to be used. As part of the background of this thesis, we will describe in detail the underlying mathematical structures of MBT as well as tools that implement MBT.

Since MBT requires a model that describes the behaviour of the SUT, it is important that this model describes the desired behaviour of the SUT. Ensuring that the behavioural model of the SUT correctly describes the behaviour of the SUT might require a large up-front effort for larger software systems.

Systems Modelling Language (SysML) is a semi-formal system architecture modelling language for system engineering based on the Unified Modelling Language (UML 2). SysML is semi-formal since the syntax is well-defined in [12], but the semantics are not formally defined. SysML is used to make complex systems understandable for technical and non-technical people while being more precise than natural language. Part of a SysML model describes the behaviour of the software system[9]. In this research, we will explore the feasibility of using this part for the generation of a model for MBT. Achieving this makes using MBT for

complex systems accessible and cost-effective while taking advantage of the characteristics of MBT compared to traditional testing.

This research is conducted at Vanderlande¹. Vanderlande develops multiple logistics process automation systems including baggage handling systems at airports and automatic warehouse storage facilities. The latter product is called ADAPTO and this thesis is written within the ADAPTO team. Also for ADAPTO, it holds that testing is a large and important part of development costs. ADAPTO is a system developed by Vanderlande which retrieves and stores items in a warehouse using robots. To manage a complex system like ADAPTO, SysML models are created. These models are used to visually represent the functionality of the system in a way that is understandable for everyone in the team. More detailed information about ADAPTO can be found in section 2.4.

This thesis will yield

- A formalization of SysML activity diagrams in the TORXAKIS modelling language including the requirements for the input activity diagram and the shortcomings of the formalization algorithm.
- A case study of applying the aforementioned formalization on a SUT. This SUT will be a part of the ADAPTO system.
- A recommendation on using MBT at Vanderlande based on the case study mentioned in the previous point.
- A step-by-step guide on implementing the translation tool from SysML models to the TORXAKIS modelling language. The implementation described by the guide can in its basic form be used for SysML models that follow a restriction which will be mentioned and explained in chapter 5. The implementation of the translation algorithm can be augmented to also support SysML models that do not follow this restriction. In this thesis, the implementation is augmented to support the SysML models of ADAPTO.

1.2 Research Question

We are interested in the possibilities of using a SysML model as a basis for MBT. This will yield the benefits of MBT without having to develop the models in case a SysML model already exists for the system in question. Therefore, we will answer the following research question in this thesis:

To what extent can SysML models be used for Model-Based Testing?

1.2.1 Subquestions

We will subdivide our main research question in the subquestions listed below. The answers to these questions will contribute to answering the main research question. The questions are labelled to make referencing them in this thesis clearer.

¹See: <https://www.vanderlande.com>

1. **SQ1** Which formalization language is most suitable for SysML models in the case of MBT?
2. **SQ2** How much domain knowledge is required to effectively translate SysML models of real world system to a modelling language for MBT?
3. **SQ3** Which steps are required to implement the formalization algorithm to be used on a real world system?

1.3 Research Method

To answer our research question, we define a translation algorithm which translates parts of a SysML model to a modelling language which can be used for MBT. We will also discuss the limitations of this translation algorithm. Then, to research the applicability of the translation algorithm, we perform a case study. We will apply the translation algorithm on a subset of the SysML models of ADAPTO.

1.4 Outline

In chapter 2, we lay out the background work for this thesis. Chapter 3 contains the formalization of SysML models in the TORXAKIS modelling language. In our case study of chapter 4, we formalize the SysML models of the SUT ADAPTO in the TORXAKIS modelling language using the formalization from chapter 3. We show a step-by-step guide on implementing the formalization of SysML models in the TORXAKIS modelling language from chapter 3 in chapter 5. This guide can in its basis be applied on any SysML model, albeit with a restriction. We augment the implementation guide to make it applicable on the SysML models of ADAPTO. We conclude and discuss our findings in chapter 6. Finally, we propose future work ideas in chapter 7.

Chapter 2

Background

In this chapter, the relevant background of our research will be laid out. To understand how Model-Based Testing (MBT) works, we start by defining the basic underlying datastructure. Then, we show how MBT uses this datastructure to determine whether a System Under Test (SUT) complies to a model. Thereafter, we show how MBT is used in the tool TORXAKIS. In the remainder of this chapter we first show what exactly SysML models are after which we show what Gherkin tests are. Then we explore related work on formalizing SysML models and we take a look at the formal language mCRL2. Finally, we discuss how ADAPTO works, the SUT of our case study.

2.1 Model-Based Testing

2.1.1 Labelled Transition System

A labelled transition system is the underlying data structure for MBT. In its basic form it is mathematically defined as a four tuple $\langle Q, L, T, q_0 \rangle$ [27] where:

- Q is a countable, non-empty set of states;
- L is a countable set of labels;
- $T \subseteq S \times (L \cup \{\tau\}) \times S$ with $\tau \notin L$ is the transition relation;
- $q_0 \in Q$ is the initial state.

The notation $q \xrightarrow{\mu} q'$ indicates that a step can be taken (the transition with label μ) from state q to state q' . This transition is also represented by $(q, \mu, q') \in T$. These transitions can be composed. If we assume that transition μ' can be performed from state q' to q'' , we can write $q \xrightarrow{\mu} q' \xrightarrow{\mu'} q''$ or more compact as $q \xrightarrow{\mu \cdot \mu'} q''$.

In the definition of an LTS, τ is an internal step that can not be observed from outside the black box. Contrary, all actions corresponding to a label $l \in L$ can be observed. If in an LTS in the state q the sequence of actions $a \cdot \tau \cdot b \cdot \tau \cdot c$ can be performed, with $a, b, c \in L$, after which state q' is reached, then the τ -abstracted sequence of observable actions is written as $q \xrightarrow{a \cdot b \cdot c} q'$. In this example q can perform the *trace* $a \cdot b \cdot c \in L^*$.

The function $traces(q)$ returns all possible traces from state q . More precisely, $traces(q) =_{def} \{\sigma \in L^* | q \xRightarrow{\sigma}\}$.

In figure 2.1, a visual representation of an LTS is displayed. The states are displayed as circles, the transitions as an arrow from the source to the target state with its label and the initial state can be recognized by an incoming arrow without a source. This LTS represents a vending machine that dispenses a water bottle when the button is pressed or a chocolate bar after first a coin is inserted and then the button is pressed. In mathematical notation, this LTS looks as follows:

- $Q = \{s0, s1, s2, s3, s4, s5\}$;
- $L = \{coin, button, water\ bottle, chocolate\ bar\}$;
- $T = \{(s0, coin, s1), (s0, button, s4), (s1, button, s2), (s2, chocolate\ bar, s3), (s4, water\ bottle, s5)\}$;
- $q_0 = s0$.

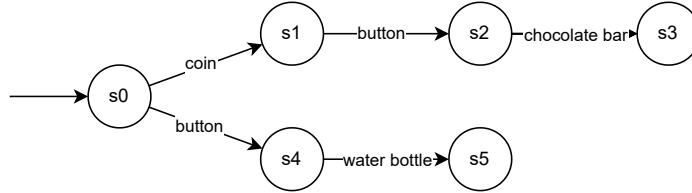


Figure 2.1: LTS that represents a simple vending machine. A water bottle is dispensed after the button is pressed and a chocolate bar is dispensed when first a coin is inserted and then the button is pressed.

For MBT, this definition is not sufficient, because we want to be able to distinguish between inputs and outputs. We need to make this distinction because for MBT we need to provide the SUT with an input and check whether an output is yielded. Therefore, we define an updated LTS.

An LTS with separate input and output labels is a five tuple $\langle Q, I, O, T, q_0 \rangle$ [27] where:

- Q is a countable, non-empty set of states;
- I is a countable set of input labels, O is a countable set of output labels where $I \cap O = \emptyset$;
- $T \subseteq S \times (L \cup \{\tau\}) \times S$ with $\tau \notin L$ is the transition relation;
- $q_0 \in S$ is the initial state.

Furthermore, a state without a τ transition is called *stable*. A state is *quiescent* when no internal τ action or output action can be performed from that state. A quiescent state is denoted with a transition to itself with the label δ . This label must not occur in $I \cup O$. We define $L_\delta = L \cup \{\delta\} = I \cup O \cup \{\delta\}$. Moreover, we define the class of LTSs with input labels \mathcal{I} and output labels \mathcal{O} as $\mathcal{LTS}(\mathcal{I}, \mathcal{O})$.

We update the LTS from figure 2.1 with input and output labels. For an input label, we add a question mark to the label and for an output label we add an exclamation mark. The states s_0 , s_1 , s_3 and s_5 are quiescent. The result can be seen in 2.2. In mathematical notation, this LTS looks as follows:

- $Q = \{s_0, s_1, s_2, s_3, s_4, s_5\}$
- $I = \{coin, button\}$
- $O = \{chocolate\ bar, water\ bottle\}$
- $T = \{(s_0, coin, s_1), (s_0, button, s_4), (s_1, button, s_2), (s_2, chocolate\ bar, s_3), (s_4, water\ bottle, s_5)\}$
- $q_0 = s_0$

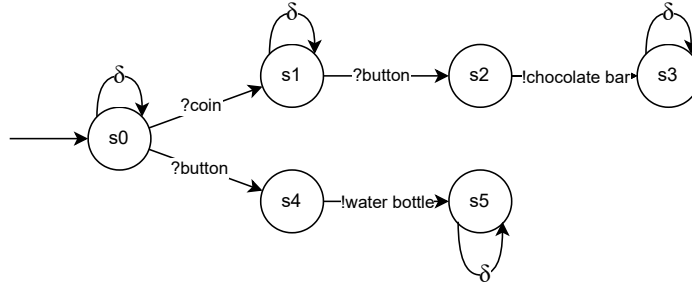


Figure 2.2: Updated LTS from figure 2.1 with the addition of input and output labels and the indication of quiescent states.

The earlier defined function *traces* does not contain sequences that include quiescence in its output. A trace that includes a δ transition is called a suspension trace. We introduce the function $Straces(q) =_{def} \{\sigma \in L_\delta^* | q \xrightarrow{\sigma}\}$ that returns all suspension traces for given state q .

We will define a function that determines which states can be reached in an LTS after a given trace. For LTS $S \in \mathcal{LTS}(\mathcal{I}, \mathcal{O})$ with initial state s_0 , we define the function *after* as S after $\sigma = \{s' | s_0 \xrightarrow{\sigma} s'\}$ which returns the set of states S can be in after evaluating trace σ .

As an example, in the LTS S from figure 2.2, S after $?coin = \{s_1\}$, S after $?coin \cdot ?button \cdot chocolate\ bar = \{s_3\}$ and S after $\epsilon = \{s_0\}$, where ϵ is defined to be the empty trace.

In addition to this function, we will define the function *out* that determines the set of observable outputs for a given state in an LTS. Given state s in LTS $S \in \mathcal{LTS}(\mathcal{I}, \mathcal{O})$ where \mathcal{O} is the set of output labels, we define $out(s) = \{o \in \mathcal{O} | s \xrightarrow{o}\} \cup \{\delta | s \xrightarrow{\delta}\}$. This function can also be applied to a set of states by taking the union of the *out* function of each individual states: $out(S) = \bigcup \{out(s) | s \in S\}$. When the function *out* is applied on LTS $S \in \mathcal{LTS}(\mathcal{I}, \mathcal{O})$ where s_0 is the initial state of S , we have $out(S) = out(s_0)$.

As an example, in the LTS S from figure 2.2, $out(S) = out(s_0) = \{!water\ bottle, !chocolate\ bar\}$, $out(s_2) = \{!chocolate\ bar\}$ and $out(S \text{ after } ?coin \cdot ?coin) = \emptyset$.

An LTS is *input enabled* if and only if each state has an outgoing transition for each input. With other words, the result of each input needs to be defined in each state of the LTS. For the set of input labels \mathcal{I} and set of output labels \mathcal{O} , the class of input enabled LTSs is defined as $\mathcal{IOTS}(\mathcal{I}, \mathcal{O})$. Since each input enabled LTS is an LTS itself, we have $\forall \mathcal{I}, \mathcal{O} \mathcal{IOTS}(\mathcal{I}, \mathcal{O}) \subseteq \mathcal{LTS}(\mathcal{I}, \mathcal{O})$.

2.1.2 Trace Equivalence

For MBT, we are interested whether the behaviour of an implementation is equivalent to what the specification specifies, as we will later see in section 2.1.6. One way to determine the equivalence between two LTSs is *trace equivalence*. Two LTSs are trace equivalent or \approx_{te} , when their *trace* sets are equivalent. Formally for LTSs l_1 and l_2 , $l_1 \approx_{te} l_2 \iff \text{traces}(l_1) = \text{traces}(l_2)$ where the traces are defined as $\text{traces}(l) = \{\sigma \in L^* \mid s \xrightarrow{\sigma}\}$. Conceptually, if two transition systems are trace equivalent, it means that without looking at the structure of either transition system, the same order of actions (traces) can be observed.

Figure 2.3 shows that $l_1 \approx_{te} l_2$, because their traces are equivalent. $\text{traces}(l_1) = \text{traces}(l_2) = \{\epsilon, a, a \cdot b, a \cdot c\}$.



Figure 2.3: LTS l_1 and LTS l_2 are *trace equivalent*.

2.1.3 ioco

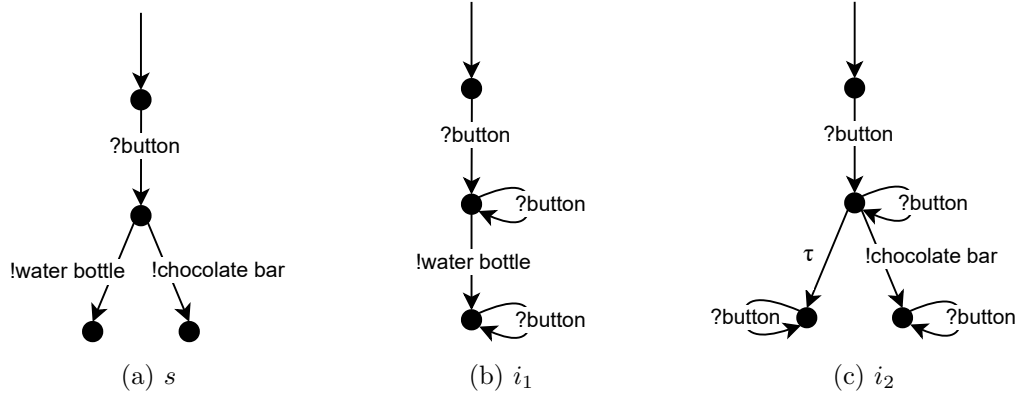
Using the definitions defined in section 2.1.1, we can define the *input output conformance relation* **ioco**. **ioco** is an implementation relation for an input enabled LTS[27] and is an adaptation to trace equivalence (section 2.1.2) for determining if two LTSs are equivalent.

Given a set of input labels L_I and a set of output labels L_O , the relation $\mathbf{ioco} \subseteq \mathcal{IOTS}(\mathcal{L}_I, \mathcal{L}_O) \times \mathcal{LTS}(\mathcal{L}_I, \mathcal{L}_O)$ is defined as follows:

$$i \mathbf{ioco} s \iff_{\text{def}} \forall \sigma \in \text{Straces}(s) : \text{out}(i \text{ after } \sigma) \subseteq \text{out}(s \text{ after } \sigma)$$

Conceptually, this means that implementation i can never produce an output after suspension trace σ that cannot be produced according to specification s after σ when $i \mathbf{ioco} s$.

Let us take a look at an example.

Figure 2.4: Specification s and implementations i_1 and i_2

i_1 **ioco** s , because for all suspension traces of s , the set of possible outputs after performing that trace on i_1 is a subset of performing that trace to s . This is proven in table 2.1.

σ	$out(i_1 \text{ after } \sigma)$	$out(s \text{ after } \sigma)$	$out(i_1 \text{ after } \sigma) \subseteq out(s \text{ after } \sigma)$
ϵ	$\{\delta\}$	$\{\delta\}$	✓
δ	$\{\delta\}$	$\{\delta\}$	✓
$?button$	$\{water \ bottle\}$	$\{water \ bottle, \ chocolate \ bar\}$	✓
$\delta \cdot ?button$	$\{water \ bottle\}$	$\{water \ bottle, \ chocolate \ bar\}$	✓
$?button \cdot !water \ bottle$	$\{\delta\}$	$\{\delta\}$	✓
$\delta \cdot ?button \cdot !water \ bottle$	$\{\delta\}$	$\{\delta\}$	✓
$?button \cdot !chocolate \ bar$	\emptyset	$\{\delta\}$	✓
$\delta \cdot ?button \cdot !chocolate \ bar$	\emptyset	$\{\delta\}$	✓

Table 2.1: Proof for showing that i_1 **ioco** s

i_2 **ioco** s , because for choosing suspension trace $\sigma = ?button$, $out(i_2 \text{ after } \sigma) = \{\delta, !chocolate \ bar\} \not\subseteq out(s \text{ after } \sigma) = \{!water \ bottle, !chocolate \ bar\}$.

2.1.4 **uioco**

Universal input-output conformance or **uioco** is a slightly weaker conformance relation than **ioco** from section 2.1.3. **uioco** is invented to circumvent a problem with **ioco** where it may be that for a given specification s , an input enabled implementation i for which i **ioco** s does not exist. The definition is as follows:

Given a set of input labels L_I and a set of output labels L_O , the relation **uioco** $\subseteq IOTS(\mathcal{L}_I, \mathcal{L}_O) \times \mathcal{LTS}(\mathcal{L}_I, \mathcal{L}_O)$ is defined as follows:

$$i \text{ **uioco** } s \Leftrightarrow_{\text{def}} \forall \sigma \in Utraces(s) \ out(i \text{ after } \sigma) \subseteq out(s \text{ after } \sigma)$$

The only difference is between **ioco** and **uioco** is that we consider so called *Utraces* with **uioco** instead of *Straces* with **ioco**.

Universal traces are suspension traces without the possibly underspecified traces[27]. Trace σ of s is underspecified if prefix σ_1 of σ ($\sigma = \sigma_1 \cdot a \cdot \sigma_2$) leads to a state of s where the remainder $a \cdot \sigma_2$ is underspecified (where a is refused). $Utraces$ is a function that, given LTS S , returns the set of universal traces of S . Formally, for specification s , $Utraces(s) = \{\sigma \in Straces(s) \mid \forall \sigma_1 ?i \sigma_2 = \sigma \text{ after } \sigma_1 \text{ must } ?i\}$. The additional condition of a $Utrace$ is that after performing each part of the trace, the state in which s is at that moment must have defined a transition for the next action of the trace[4]. This is best explained with an example.

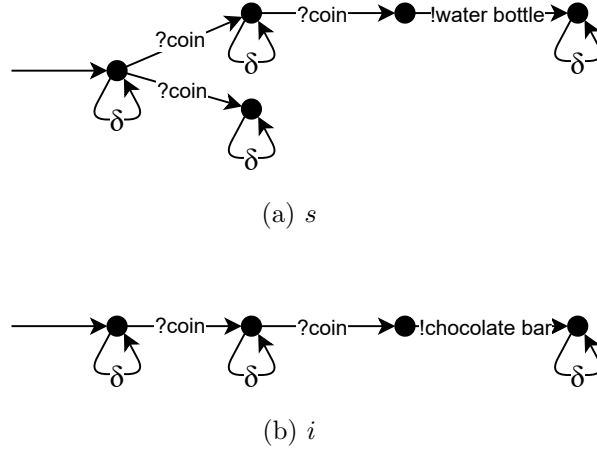


Figure 2.5: Specification s and implementation i

In figure 2.5a we see that after we input $?coin$, another $?coin$ input is not defined on all possible paths. Therefore $?coin \cdot ?coin \notin Utraces(s)$, but $?coin \cdot ?coin \in Straces(s)$. Since $uioco$ is a weaker relation than $ioco$ it holds that if implementation i is $ioco$ to specification s , this implies i **uioco** s , so **ioco** \subseteq **uioco**.

We will show with an example that $uioco$ is strictly weaker than $ioco$. In figure 2.5, i **ioco** s , because after the trace $?coin \cdot ?coin$ is performed on i , the only possible output is $chocolate\ bar$, which is not a subset of the possible output after performing the trace $?coin \cdot ?coin$ on s , namely $water\ bottle$. However, i **uioco** s , because the trace $?coin \cdot ?coin$ is not a $Utrace$, which is explained above and is therefore not considered in the $uioco$ relation. For all the traces that are in the set $Utraces(s)$, the $uioco$ relation does hold.

2.1.5 Symbolic Transition System

A Symbolic Transition System (STS) is an enhancement to an LTS. LTSs are inefficient at modelling data-intensive systems in the sense that it yields a large state space. STSs solve this problem by storing data outside of a state. An STS is mathematically defined as a six tuple $\langle L, l_0, \mathcal{V}, \mathcal{I}, \Delta, \rightarrow \rangle$ where:

- L is a set of locations;
- $l_0 \in L$ is the initial location;
- \mathcal{V} is a set of location variables;
- \mathcal{I} with $\mathcal{V} \cap \mathcal{I} = \emptyset$ is a set of interaction variables;

- Δ is the set of gates;
- The relation $\rightarrow \subseteq L \times \Delta_\tau \times \mathbf{F}(Var) \times \mathbf{T}(Var)^\mathcal{V} \times L$ is the switch relation with $\tau \notin \Delta$ denotes an unobservable gate, $\Delta_\tau = \Delta \cup \{\tau\}$, $Var =_{def} \mathcal{V} \cup \mathcal{I}$, $\mathbf{F}(Var)$ is the set of all first order formulas ϕ satisfying $\mathbf{free}(\phi) \subseteq Var$ where $\mathbf{free}(\phi)$ denotes the set of free variables of a first order formula ϕ , $\mathbf{T}(Var)$ are the terms over Var , with $\mathbf{T}(Var)^\mathcal{V}$ is a term mapping over Var to \mathcal{V} [8].

Figure 2.6 shows an example of an STS with:

- $L = \{l_0, l_1, l_2\}$;
- $l_0 = l_0$;
- $\mathcal{V} = \{\text{coin}\}$;
- $\mathcal{I} = \{\text{money}\}$;
- $\Delta = \{?\text{coin}, !\text{chocolate bar}\}$;
- \rightarrow is given by the directed edges linking the locations in figure 2.6.

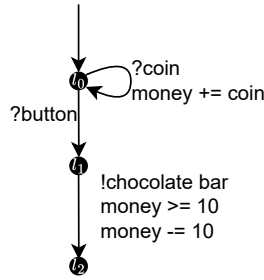


Figure 2.6: Symbolic transition system of a vending machine.

sioco is a conformance relation on STSs that coincides with *ioco* for LTSs. Conceptually, *sioco* can be interpreted as lifting *ioco* from LTSs to STSs by introducing operations on transition system with (possibly infinite) data sets. *sioco* does not add any meaning to *ioco*, it just allows for the reasoning of the conformance relation on STSs. [8] introduces in section 5 the necessary concepts to define *sioco* and that section defines the *sioco* implementation relation. For our research, the exact definition of *sioco* is not relevant, hence it is left out.

2.1.6 Model-Based Testing

In section 2.1.3 we have defined the input output conformance relation (*ioco*). This relation is used in MBT to determine if a SUT meets its behaviour model.

MBT is a *black-box* testing technique[27]. This means that, as opposed to white-box testing, the internal mechanisms of the SUT cannot be observed. It is important here that the model MBT uses complies to the implementation requirements. To test the conformance of a SUT to its model, we need to specify a conformance relation. Most often the aforementioned input

output conformance relation (**ioco**) or a variation on *ioco* is used. However, variations on *ioco* (*tioco* for example which adds time constraints to *ioco*) and other relations can be used as well.

To test a SUT using MBT, we use an MBT tool. These tools will be discussed in section 2.1.7. A tool uses the specified behaviour model to generate test cases. The test cases consist of inputs and expected outputs. The MBT tool stimulates the SUT by providing inputs and checks if the outputs of the SUT match the expectations. Then, the tool decides whether the test has passed or failed.

Since the messages generated from an MBT tool are usually not directly understandable by the SUT, we need to use an *adapter*. An adapter receives the messages from the tool and performs the corresponding action on the SUT. When the SUT emits an output, the adapter receives it and sends a message to the MBT tool in the format it expects. An overview of the test architecture can be found in figure 2.7.

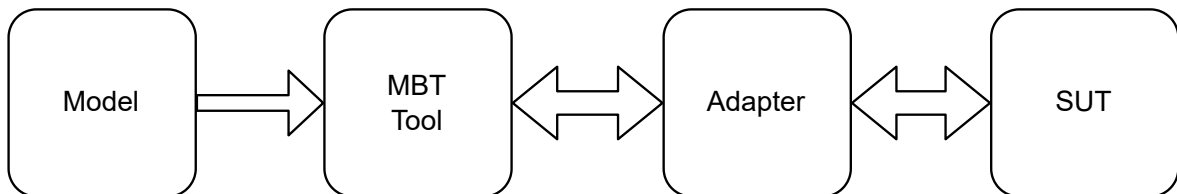


Figure 2.7: MBT Test architecture

2.1.7 Model-Based Testing Tools

In this section, we briefly introduce which MBT tools are considered during this thesis. These tools use the MBT testing technique described in section 2.1.6.

TorX

TORX is the first MBT tool based on the *ioco* theory. It is a prototype test tool that uses a formal specification of the SUT and tests the conformance of it to the implementation[25]. TORX was developed at the Formal Methods and Tools research group at the University of Twente (UT) in the Netherlands, in close collaboration with Eindhoven University of Technology (TUE), Philips Research Laboratories and Lucent Technologies[22]. TORX was developed in the research and development project *Côte de Resyste* with the goal of automatically generating tests for a SUT from its specification model.

JTorX

JTORX is the successor to TORX. In addition to the *ioco* conformance relation, JTORX also uses *uioco*, which is explained in section 2.1.4. JTORX can also check for $(u)ioco$ relations between models and check underspecified traces in a model. In addition to this, JTORX is easier to deploy, because it has an improved installation, configuration and usage process. These improvements allow for usage of JTORX outside educational purposes[2, 3].

TorXakis

TORXAKIS is the successor to JTORX. Instead of implementing *ioco* for LTSs, TORXAKIS implements *ioco* for *Symbolic Transition Systems* (STS). The *ioco* theory focuses on, and is therefore limited to, dynamic aspects of system behaviour. The dynamic aspect of a system is the control flow of the system using states. Static aspects of a SUT, like data structures and their operations and constraints, are not covered by the *ioco* theory. STSs add (infinite) data and data-dependent control flow, such as guarded transitions to LTSs[26]. More precisely, TORXAKIS uses symbolic *ioco* (*sioco*) which lifts *ioco* to the symbolic level. No expressiveness is added by STSs and *sioco* compared to LTSs and *ioco*, but STSs and *sioco* allow for symbolically representing and manipulating large or infinite transition systems. Since STS allow for constraints on transitions, TORXAKIS also supports this. To solve these constraints, the SMT solver Z3 is used[21]. TORXAKIS is an experimental tool for on-the-fly MBT, but can be used freely under a BSD3 license. TORXAKIS is open source and can be found on (<https://torxakis.org>)[8].

Axini

Axini is an MBT tool and company located in Amsterdam, The Netherlands¹. The MBT tool is developed for commercial purposes. Axini introduces the Axini Modelling Language (AML). This language resembles an imperative syntax as opposed to the TORXAKIS modelling language, which has a functional syntax. Functionally, Axini improves on TORXAKIS by allowing to model time constraints. Furthermore, Axini provides better support and maintains the tool, but naturally this comes at a cost for the customer[6].

2.2 SysML

The Systems Modeling Language, or SysML, is a general-purpose system architecture modeling language for systems engineering applications. Systems engineering is the process of designing the parts, communication between parts and their behaviour of a system. Such a system can be a hardware component, a software component, or a component which consists of hardware and software. SysML reuses a subset of the second generation of the Unified Modelling Language (UML 2[7]) while extending the language to make it suitable for system engineering. The goal of SysML is to make it quick to model the behaviour and structure of a system while making it easy to understand for all stakeholders that are involved with the designing, developing, testing and maintaining of a system. A SysML model is easy to understand, because the diagrams are graphically visualized and because no engineering knowledge is required. To model a system, a SysML model can be modelled in a supported application (Enterprise Architect² for example). Usually, such an application allows the modeller to graphically design the model by dragging, dropping and connecting components. A model can also be represented in a computer readable format, which is specified in the OMG SysML specification[14, 15, 12]. In section 2.2.1 the structure of a SysML model is explained.

¹See: <https://www.axini.com/>

²See: <https://sparxsystems.com>

2.2.1 Diagrams

A SysML model consists of zero or more instances of each of the 9 different diagram types. All diagrams are described and specified in the SysML OMG specification[12]. Like UML, SysML is not formal. This means that the syntax of a UML model and a SysML model are well defined, but the semantics of them are not formally defined[5].

Each diagram can be categorized into three categories: Structural Diagrams, Behaviour Diagrams and the Requirement Diagram, which is a single diagram. The category of each SysML diagram can be found in figure 2.8.

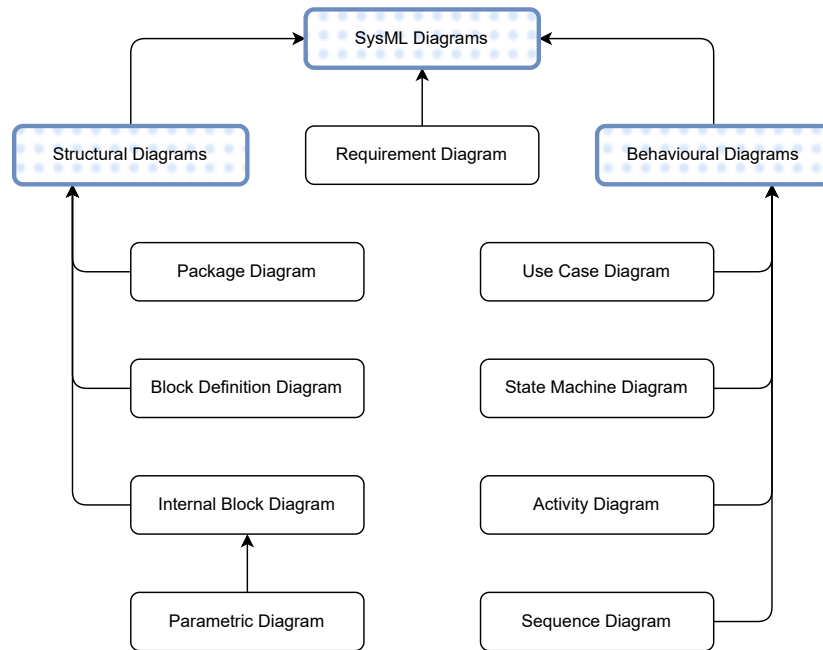


Figure 2.8: SysML diagrams grouped by their type[14]. The blue coloured and dotted blocks indicate a group (or an interface in UML).

Each diagram has its own function for modelling a system. The function of each diagram is:

- The **Requirement diagram** specifies requirements of the modelled system and the relationships between requirements and the modelled components that need to implement the requirement.
- The **Use case diagram** describes the relationship between actors (something or someone that interacts with the system) and the actions that an actor can perform on the system (the use case).
- The **Activity diagram** specifies the behaviour of a system component by modelling the control flow and data flow.
- The **Sequence diagram** models the communication between objects on a timeline. These objects can be actors or system components.

- The **State machine diagram** shows the sequences of states that an object or an interaction go through during its lifetime in response to *events* (or *triggers*), which may result in side-effects.
- The **Block definition diagram** represents a system component as a *block*. A block can contain *interfaces*, where the system component can communicate through ports, the contents of the system component which can be their properties or related state machine diagrams, *or* how a block relates to another block.
- The **Internal block diagram** is a diagram owned by a particular block that shows its encapsulated structural contents. An internal block diagram gives a white-box perspective of a black-box block.
- The **Parametric diagram** is a type of internal block diagram which ensures that its owner block adheres to specified mathematical formulas. Such a formula can for example be a law of nature.
- The **Package diagram** is used to organize the structure of the entire repository of the SysML model[12].

2.3 Related Work

This section describes the related work that has been done in this research field. We will look at research that attempts to formalize the semi-formal SysML models for various purposes.

The semantics of UML and SysML models have been formalized before in preceding academic research. In [20], the formalization of UML state machines is given in two steps. First, the structure of a state machine is translated to a term rewrite system. Secondly, the operational semantics of the state machines is defined. During these steps, decisions needed to be made regarding the behaviour of certain UML state machine constructs.

Since SysML is a subset and an extension of UML 2, the majority of the formalization can be re-applied. The formalization of SysML state machines is also defined in [1] and [5]. An attempt has also been done at formalizing SysML activity diagrams in [18]. The paper describes an *activity calculus* and an operational semantics for activity diagrams. This activity calculus is an algebraic-like language that captures the behaviour of an activity diagram. This paper is explored in more detail in section 2.3.2.

Finally, in section 2.3.3, we will take a look at an attempt on translating a SysML activity diagram to TORXAKIS. In [13], a step-by-step approach for this translation is provided. This step-by-step approach is the result of a case study where activity diagrams were manually translated to TORXAKIS for MBT.

In the following sections, we lay out the different related work areas in more detail.

2.3.1 FormaSig

The **FormaSig**³ project is a collaboration of the Dutch and German railway infrastructure managers, Prorail and Deutsche Bahn, *and* the Eindhoven University of Technology (TUE) and the University of Twente. The goal of the project is to formalize the *EULYNX* standard

³Formal Methods in Railway Signaling Infrastructure Standardization Processes.

so that the formalization can be used to check if it satisfies a collection of safety properties. **EULYNX** is an initiative of thirteen European railway infrastructure managers to specify a standard for interfaces between the various components of a railway signalling system such as a signal, point, level crossing and an interlocking. This standard is modelled in SysML. The goal of FormaSig is formalizing these SysML models into *mCRL2*.

mCRL2 (micro Common Representation Language 2) is a specification language in which the behaviour of distributed systems can be specified and analyzed[11]. The mCRL2 toolset provides over 60 tools which use a mCRL2 specification. FormaSig uses the model checking tool to check if the safety criteria are met with EULYNX. Moreover, FormaSig uses MBT to test if the EULYNX implementation conforms to the formal model. The mCRL2 toolset does not provide an MBT tool, but the MBT tool JTORX (see section 2.1.7) uses as input language an mCRL2 specification[10]. Figure 2.9 gives an overview of the test architecture of FormaSig.

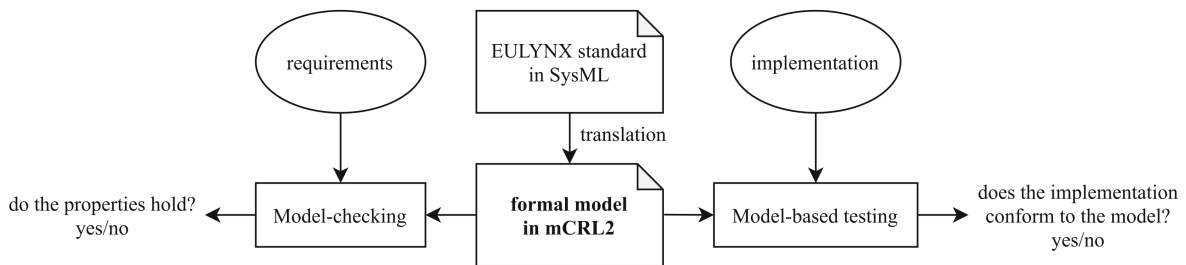


Figure 2.9: Test architecture of FormaSig. Using a formalized mCRL2 model from the EULYNX SysML model, model checking and MBT are applied[5].

Since the SysML model of EULYNX predominantly uses the state machine diagram for its behaviour specification, the FormaSig project formalizes this diagram in [5] and they applied this formalization on EULYNX in [28].

mCRL2 was chosen in favour of other formalization language, because mCRL2 is founded and developed in the TUE and because the mCRL2 toolset provided all needed tools for FormaSig, explained Mark Bouwman and Djurre van der Wal from FormaSig in an interview we conducted with them.

The formalization approach of FormaSig first formalizes SysML state machine diagrams generally after which this formalization is adapted to suit EULYNX. The goal of FormaSig is to achieve a high degree of modularity with formalizing SysML state machine diagrams in mCRL2. FormaSig achieves this by first abstracting away from the properties of a SysML state machine diagram that have ambiguous semantics according to them. These properties are for example the granularity of interleaving i.e. what is considered an atomic step on the SUT, the run-to-completion semantics, i.e. if a step in the state machine is executed from start to finish before the next step can be started, or not, *and* the syntax and semantics of the action language, the language used to describe the actions in a SysML state machine[28]. This generic formalization consists of two parts.

1. The semantics of SysML is formalized. This consists of formalizing a SysML state machine by creating data structures in mCRL2 that can represent the structure of a state machine *and* by formalizing how a state machine progresses from the current state to the next state.

2. The *action language*, the language used to specify guards and the effects of transitions in a state machine, is used to enhance the formalization of step 1 by taking into account guards on transitions when a state machine progresses from the current state to the next state and by taking into account the effects of the actions performed when transitioning from one state to another.

This formalization is adapted to suit EULYNX and its properties (granularity of interleaving, run-to-completion semantics and the action language). This adapted formalization is applied and used for model checking in [28].

2.3.2 Activity Diagrams

Preceding academic research has also been conducted on formalizing SysML activity diagrams. Yosr Jarraya et al. describe in their paper[18] an *Activity Calculus* (AC) for activity diagrams. This defined AC is a grammar in Backus-Naur-Form (BNF) that can be used to express activity diagrams. In addition to the AC, Structural Operational Semantics (SOS) for this AC is given using derivation rules. The SOS gives meaning or semantics to the activity diagrams in SysML.

Difference between Activity Diagram and State Machine Diagram

Activity Diagrams and State Machine diagrams are both used to model behaviour of a system. However, these diagrams are different. Figure 2.10 compares an activity diagram to a state machine diagram. An activity diagram describes a sequence of actions in a flowchart. An activity diagram automatically transitions to the next action if the preceding action has finished. In contrary to activity diagrams, state machine diagrams only transition to the next state when an event is triggered. A state machine diagram is used to model the lifetime behaviour of a single system component. Activity diagrams are used to model the control flow of an *activity*, a flow of action which may involve multiple system components, which can involve concurrency and synchronization.

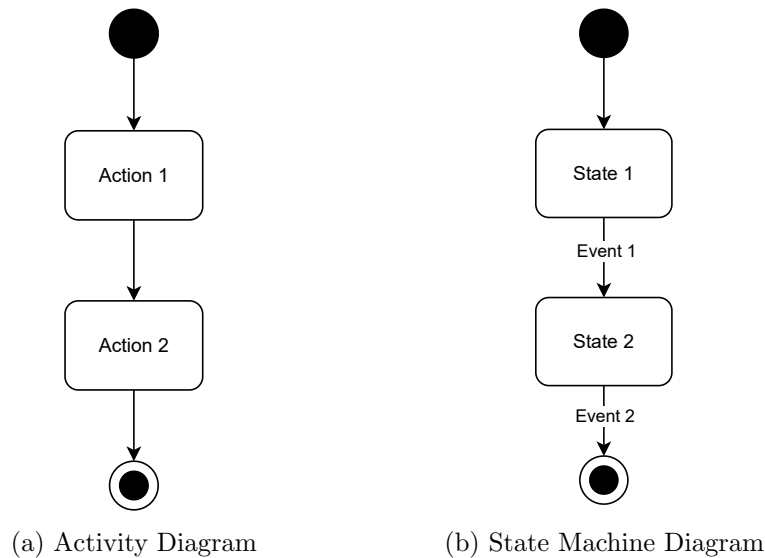


Figure 2.10: Comparison between an activity diagram and a state machine diagram.

Activity Calculus

The AC is defined on the basic constructs of an activity diagram which can be seen in figure 2.11. These activity diagram constructs are a subset of all SysML activity diagram constructs. Figure 2.15 shows other activity diagram constructs that will be considered later.



Figure 2.11: Basic constructs of a SysML activity diagram considered in the Activity Calculus.

The grammar that describes the structure of the activity diagrams is given in Backus-Naur-Form (BNF), conceptually a term rewriting system for context-free grammars, and can be seen in figure 2.12.

$$\begin{array}{ll}
 \mathcal{A} ::= \epsilon & \mathcal{B} ::= \overline{\mathcal{A}} \\
 \quad | \iota \mapsto \mathcal{N} & \quad | \iota \mapsto \mathcal{M} \\
 & \quad | \bar{\iota} \mapsto \mathcal{N} \\
 \\
 \mathcal{N} ::= \epsilon & \mathcal{M} ::= \mathcal{N} \\
 \quad | l : \otimes & \quad | l : Merge(\mathcal{M}) \\
 \quad | l : \odot & \quad | l : x.Join(\mathcal{M}) \\
 \quad | l : Merge(\mathcal{N}) & \quad | l : Fork(\mathcal{M}, \mathcal{M}) \\
 \quad | l : x.Join(\mathcal{N}) & \quad | l : Decision_p(\langle g \rangle \mathcal{M}, \langle \neg g \rangle \mathcal{M}) \\
 \quad | l : Fork(\mathcal{N}, \mathcal{N}) & \quad | l : Decision(\langle g \rangle \mathcal{M}, \langle \neg g \rangle \mathcal{M}) \\
 \quad | l : Decision_p(\langle g \rangle \mathcal{N}, \langle \neg g \rangle \mathcal{N}) & \quad | \bar{l} : a^n \mapsto \mathcal{M} \\
 \quad | l : Decision(\langle g \rangle \mathcal{N}, \langle \neg g \rangle \mathcal{N}) & \quad | \overline{\mathcal{M}^n} \\
 \quad | l : a \mapsto \mathcal{N} & \\
 \quad | l &
 \end{array}$$

(a) Unmarked (b) Marked

Figure 2.12: Grammar of the syntax of the Activity Calculus

The ι symbol represents the initial state. The \mapsto corresponds to an edge in an activity diagram. ϵ is the empty activity. The letter l denotes the label of the corresponding construct. The x in the *Join* step denotes the number of incoming edges. There are two *Decision* steps, a probabilistic case and a non-deterministic case. Both cases contain a boolean guard g . Either g is true or $\neg g$. For the probabilistic case, p denotes the probability the first case is chosen. The probability of choosing the second case is $1 - p$.

In their paper, Yosr Jarraya et al. also define grammar for the so-called *marked* syntax of the AC. The idea behind providing a marked syntax is the ability to express the state of the activity diagram. These marks are represented as overhead bars and we will encounter them

in the SOS.

Figure 2.13 shows the relation between the activity diagram constructs and the AC syntax.

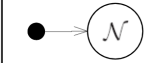


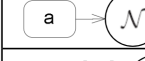
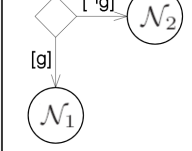
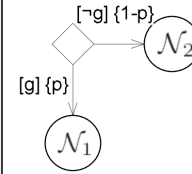
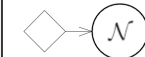
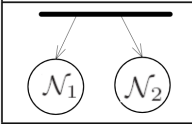
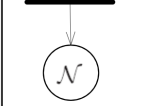
	AD Constructs	AC Syntax
\mathcal{A}		$\iota \mapsto \mathcal{N}$
\mathcal{N}		$l: \odot$
		$l: \otimes$
		$l: a \mapsto \mathcal{N}$
		$l: Decision(\langle g \rangle \mathcal{N}_1, \langle \neg g \rangle \mathcal{N}_2)$
		$l: Decision_p(\langle g \rangle \mathcal{N}, \langle \neg g \rangle \mathcal{N})$
		$l: Merge(\mathcal{N})$ or l
		$l: Fork(\mathcal{N}_1, \mathcal{N}_2)$
	$l: x.Join(\mathcal{N})$ or l (x is the number of incoming edges)	

Figure 2.13: Relation between activity diagram (AD) constructs and the activity calculus (AC) syntax[18].

We show an example of a SysML activity diagram with its corresponding AC representation to illustrate how this translation is performed.

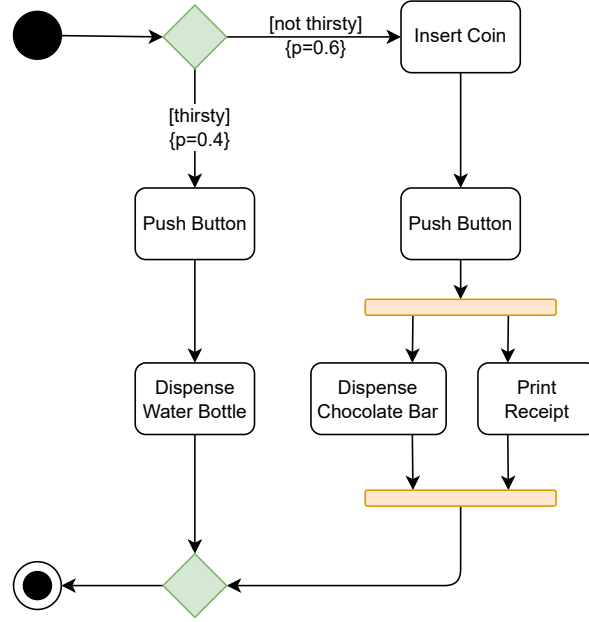


Figure 2.14: Activity diagram of a vending machine.

The activity diagram of figure 2.14 can be expressed using the unmarked term $\mathcal{A}_{\text{vending}}$ such that:

$$\begin{aligned}
 \mathcal{A}_{\text{vending}} &= \iota \succ l_1 : \text{Decision}_{0.4}(\langle \text{thirsty} \rangle \mathcal{N}_1, \langle \text{not thirsty} \rangle \mathcal{N}_2) \\
 \mathcal{N}_1 &= l_2 : \text{PushButton} \succ l_3 : \text{DispenseWaterBottle} \succ l_4 : \text{Merge}(l_5 : \odot) \\
 \mathcal{N}_2 &= l_6 : \text{InsertCoin} \succ l_7 : \text{PushButton} \succ l_8 : \text{Fork}(\mathcal{N}_3, \mathcal{N}_4) \\
 \mathcal{N}_3 &= l_9 : \text{DispenseChocolateBar} \succ l_{10} : 2.\text{Join}(l_4) \\
 \mathcal{N}_4 &= l_{11} : \text{PrintReceipt} \succ l_{12} : 2.\text{Join}(l_4)
 \end{aligned}$$

Structural Operational Semantics

To show meaning to the AC, a structural operational semantics (SOS) needs to be defined. SOS consists of rewrite rules that can be applied on AC terms. These rules formalize the meaning, or semantics, of AC terms and they therefore indirectly formalize the semantics of SysML activity diagrams. The rules can be found in appendix A.

It is important to note that the defined rewrite rules are not derived from the SysML specification, because no such semantics exist, but are derived from an interpretation of the behaviour of activity diagrams. Hence, these rewrite rules might suggest a different meaning than expected in another environment.

Conceptually, the meaning of the derivation rules (and therefore the behaviour of activity diagrams) is as follows.

1. To describe the state in which an activity diagram is in, the notion of *tokens* is used. A token correspond with one overbar in the grammar of figure 2.12b. It is possible to have multiple bar on a single construct, because loops in the diagram are permitted. For clarity, $\bar{t}^1 = \bar{t}$ and $\bar{t}^0 = \iota$.

2. The token flow starts by giving the initial node a token. If there are multiple initial nodes, we replace these by a single initial node which connects to all the targets of the initial nodes via a fork node. This is equivalent semantically, because the fork node allows all its descendants to run asynchronously, just like having multiple initial nodes would do.
3. When an action receives a token, it starts running the action. Only when an action is completely finished, it can pass its token to the next construct.
4. The flow of an activity diagram can be ended by a flow final node, which deletes a token. All flows end abruptly when an activity final node is reached, since then all tokens are deleted.
5. The fork node gives each of its descendants a token when it receives a token. The result of this is that tasks actions will be executed in parallel. Conversely, the join node waits until all incoming edges have ‘delivered’ a token, and only then forwards this token to the next node.
6. The decision node can pass an incoming token to one of its descendants. Conversely, a merge node can forward an incoming token, from one of the incoming edges, to the outgoing edge.

Updated Formalization

To the best of our knowledge, the formalization of SysML activity diagrams discussed in the previous subsection was the first academic work on this topic. Since then, there has been one notable contribution[23] to the formalization of the activity diagrams, which expanded on the previous work. The SysML activity diagram constructs considered in the formalization of [23] are displayed in figure 2.15.

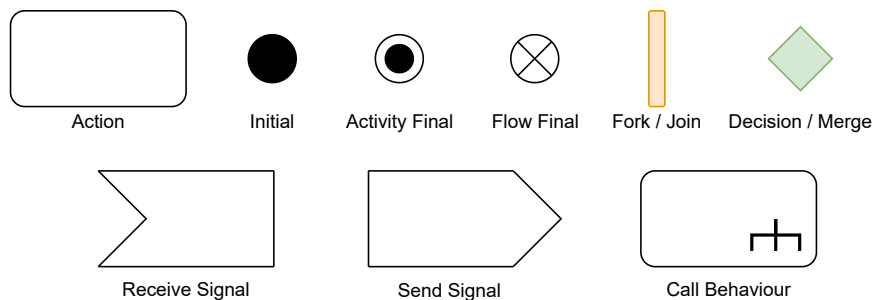


Figure 2.15: Constructs of a SysML activity diagram considered in NuAC[23]. A call behaviour \mathcal{A} is encoded to a separate NuAC term so that it can be referenced from other NuAC terms.

Table 2.2 shows the updated relation between the SysML activity diagram constructs and the NuAC terms. The terms *Decision*, *Merge*, *Fork* and *Join* have been abbreviated to D , M , F and J respectively.

Artifacts	NuAC Terms	Description
	$l : \iota \rightarrow \mathcal{N}$	Initial node is activated when a diagram is invoked.
	$l : \odot$	Activity final node stops the diagram's execution.
	$l : \otimes$	Flow final node kills its related path's execution.
	$l : a \rightarrow \mathcal{N}$	Action node defines an atomic action.
	$l : a \uparrow \mathcal{A} \rightarrow \mathcal{N}$	Call behavior node invokes a new behavior.
	$l : a!v \rightarrow \mathcal{N}$	Send node is used to send a signal/object.
	$l : a?v \rightarrow \mathcal{N}$	Receive node is used to receive a signal/object.
	$l : D(\mathcal{A}, p, g, \mathcal{N}, \mathcal{N})$	Decision node with a call behavior \mathcal{A} , a convex distribution $\{p, 1 - p\}$ and guarded edges $\{g, -g\}$.
	$l : M(x) \rightarrow \mathcal{N}$ or l	Merge node specifies the continuation and $x = \{x_1, x_2\}$ is a set of input flows.
	$l : F(\mathcal{N}_1, \mathcal{N}_2)$	Fork node models the concurrency that begins multiple parallel control threads. UML 2.0 activity forks model unrestricted parallelism.
	$l : J(x) \rightarrow \mathcal{N}$ or l	Join node presents the synchronization where $x = \{x_1, x_2\}$ is a set of input pins.

Table 2.2: Relation between activity diagram artifacts and the New Activity Calculus (NuAC) syntax[23].

The new constructs are the receive signal action (indicated with a ? symbol), the send signal action (indicated with a ! symbol) and the call behaviour action (indicated with a \uparrow symbol) which together cover all behaviour constructs of the SysML activity diagram. The send and receive actions are accompanied by an object v , the send or received object. The properties of v do not influence the control flow of a SysML activity diagram. The grammar of the AC has been updated to the New Activity Calculus (NuAC), which can be seen in figure 2.16.

\mathcal{A}	$::=$	ϵ
		$l : \bar{t}^n \mapsto \mathcal{N}$
\mathcal{N}	$::=$	$\bar{\mathcal{N}}$
		$l : Fork(\mathcal{N}, \mathcal{N})$
		$l : Decision(\mathcal{A}, p, g, \mathcal{N}, \mathcal{N})$
		l
		$l : \bar{\mathcal{X}}^n \mapsto \mathcal{N}$
		$l : \otimes$
		$l : \odot$
\mathcal{X}	$::=$	$a\mathcal{B}$
		$Join(x_1, x_2)$
		$Merge(x_1, x_2)$
\mathcal{B}	$::=$	$\uparrow \mathcal{A}$
		$!v$
		$?v$
		ϵ

Figure 2.16: Syntax of New Activity Calculus (NuAC): Updated and optimized syntax of AC

The derivation rules have also been updated for NuAC. The rules and their conceptual meaning can be found in table 2.3. We will use these derivation rules in our own translation algorithm in section 3.4. The derivation rules must be understood before being able to correctly use them when defining a formalization. The result of SOS is a derivation tree which models the possible execution paths of the NuAC term. The derivation rules specify how a NuAC term can transition to the final state, which indirectly specifies the flow (i.e. order) of actions in an activity diagram from an initial node to a final node, which is the behaviour of an activity diagram.

Σ is defined as the set of labels for the transitions in NuAC. For clarification, these labels are not related to the labels of the actions of an activity diagram. The executing active

node has label $\alpha \in \Sigma$, $\epsilon \in \Sigma$ is the empty action and p is a probability with $p \in [0, 1]$. A general transition is in the form $\mathcal{A} \xrightarrow{\alpha}_p \mathcal{A}'$, where \mathcal{A} and \mathcal{A}' are NuAC terms. p indicates the probability that the transition occurs, which is denoted by $P(\mathcal{A}, \alpha, \mathcal{A}')$. For simplicity $\mathcal{A} \xrightarrow{\alpha}_p \mathcal{A}' = \mathcal{A} \xrightarrow{\alpha} \mathcal{A}'$ for $p = 1$ which means that the transition is non-probabilistic. $\mathcal{A}[\mathcal{N}]$ means that \mathcal{N} is a sub-term of \mathcal{A} and $|\mathcal{A}|$ is term \mathcal{A} without tokens. For the call behaviour $a \uparrow \mathcal{N}$, $\mathcal{A}[a \uparrow \mathcal{N}]$ is denoted by $\mathcal{A} \uparrow_a \mathcal{N}$.

For space constraints, the terms *Decision*, *Merge*, *Fork* and *Join* in the derivation rules have been abbreviated to D , M , F and J respectively.

Identifier	Derivation Rule	Description
INT-1	$l : \iota \mapsto \mathcal{N} \xrightarrow{\epsilon} l : \bar{\iota} \mapsto \mathcal{N}$	This axiom starts the execution of the expression by putting a token on the initial node ι .
INT-2	$l : \bar{\iota} \mapsto \mathcal{N} \xrightarrow{l} l : \iota \mapsto \bar{\mathcal{N}}$	Propagate token from the initial node to \mathcal{N} .
ACT-1	$\forall n > 0, m \geq 0, l : \overline{a^m} \mapsto \mathcal{N}^n \xrightarrow{l} \overline{a^{m+1}} \mapsto \mathcal{N}^{n-1}$	Execute action a .
ACT-2	$\forall n, m \geq 0, l : \overline{a^{m+1}} \mapsto \mathcal{N}^n \xrightarrow{l} \overline{a^m} \mapsto \overline{\mathcal{N}^n}$	Propagate token to \mathcal{N} after execution of a .
ACT-3	$\forall n, m \geq 0 \frac{\mathcal{N} \xrightarrow{\alpha_p} \mathcal{N}'}{l : \overline{a^m} \mapsto \mathcal{N}^n \xrightarrow{\alpha_p} l : \overline{a^m} \mapsto \mathcal{N}'^n}$	Probabilistic version of ACT-2
BEH-0	$\forall n > 0, l : \overline{a \uparrow \mathcal{A}^n} \mapsto \mathcal{N} \xrightarrow{l} l : \overline{a \uparrow \mathcal{A}^{n-1}} \mapsto \mathcal{N}$	Execute the called behaviour \mathcal{A} from a .
BEH-1	$\forall n > 0, \frac{\mathcal{A} = l' : \iota \mapsto \mathcal{N}, \mathcal{A}' = l' : \bar{\iota} \mapsto \mathcal{N}}{l : \overline{a \uparrow \mathcal{A}^n} \mapsto \mathcal{N} \xrightarrow{l} l : \overline{a \uparrow \mathcal{A}'^{n-1}} \mapsto \mathcal{N}}$	Generalized case of BEH-0 where a is executed.
BEH-2	$\forall n \geq 0, \frac{\mathcal{A}[l' : \odot] \xrightarrow{l'} \mathcal{A} }{l : \overline{a \uparrow \mathcal{A}^n} \mapsto \mathcal{N} \xrightarrow{l} l : \overline{a \uparrow \mathcal{A} ^n} \mapsto \bar{\mathcal{N}}}$	Finish the called behaviour and continue with \mathcal{N}
BEH-3	$\forall n > 0, \frac{\mathcal{A} \xrightarrow{\alpha_p} \mathcal{A}'}{l : \overline{a \uparrow \mathcal{A}^n} \mapsto \mathcal{N} \xrightarrow{\alpha_p} l : \overline{a \uparrow \mathcal{A}'^{n-1}} \mapsto \mathcal{N}}$	Effect of the call behaviour when executing \mathcal{A}
BEH-4	$\forall n \geq 0, \frac{\mathcal{N} \xrightarrow{\alpha_p} \mathcal{N}'}{l : \overline{a \uparrow \mathcal{A}^n} \mapsto \mathcal{N} \xrightarrow{\alpha_p} l : \overline{a \uparrow \mathcal{A}'^{n-1}} \mapsto \mathcal{N}'}$	Effect of the call behaviour when executing \mathcal{A}
FRK-1	$\forall n > 0, l : \overline{F(\mathcal{N}, \mathcal{N})^n} \xrightarrow{l} l : \overline{F(\bar{\mathcal{N}}, \bar{\mathcal{N}})^{n-1}}$	Propagate incoming token of the fork to its sub-terms.
FRK-2	$\forall n \geq 0, \frac{\mathcal{N} \xrightarrow{\alpha_p} \mathcal{N}'}{l : \overline{F(\mathcal{N}, \mathcal{N})^n} \xrightarrow{\alpha_p} l : \overline{F(\mathcal{N}', \mathcal{N})^n}}$	Execute a sub-term of the fork.
DEC-1	$\forall n > 0, l : \overline{D(g, \mathcal{N}, \mathcal{N})^n} \xrightarrow{l, g} l : \overline{D(g, \bar{\mathcal{N}}, \mathcal{N})^{n-1}}$	Token flows to the edge satisfying guard g .
DEC-2	$\forall n > 0, l : \overline{D(p, g, \mathcal{N}, \mathcal{N})^n} \xrightarrow{l, g} l : \overline{D(p, g, \bar{\mathcal{N}}, \mathcal{N})^{n-1}}$	Probabilistic version of DEC-1 .
DEC-3	$\forall n > 0, \frac{\mathcal{A} = l' : \iota \mapsto \mathcal{N}, \mathcal{A}' = l' : \bar{\iota} \mapsto \mathcal{N}}{l : \overline{D(\mathcal{A}, p, g, \mathcal{N}, \mathcal{N})^n} \xrightarrow{l} l : \overline{D(\mathcal{A}', p, g, \mathcal{N}, \mathcal{N})^{n-1}}}$	Initiate an invoked behaviour.
DEC-4	$\forall n > 0, \frac{\mathcal{A}[l' : \odot] \xrightarrow{l'} \mathcal{A} }{l : \overline{D(\mathcal{A}, p, g, \mathcal{N}, \mathcal{N})^n} \xrightarrow{l', g} l : \overline{D(\mathcal{A} , p, g, \bar{\mathcal{N}}, \mathcal{N})^n}}$	Token flow of a guarded path while terminating behaviour \mathcal{A} .
DEC-5	$\forall n > 0, \frac{\mathcal{N} \xrightarrow{\alpha_q} \mathcal{N}'}{l : \overline{D(\mathcal{A}, p, g, \mathcal{N}, \mathcal{M})^{n-1}} \xrightarrow{\alpha_q} l : \overline{D(\mathcal{A}, p, g, \mathcal{N}', \mathcal{M})^{n-1}}}$	Change of a decision term when a sub-term changes.
MRG-1	$\forall n \geq 0, l : \overline{\bar{\mathcal{N}}} \mapsto l' : \overline{M(x, y)^n} \xrightarrow{l} l : \overline{\mathcal{N}} \mapsto l' : \overline{M(\bar{x}, y)^n}$	Propagate token from incoming on a merge term.
MRG-2	$l : \overline{M(\bar{x}, \bar{y})} \mapsto \mathcal{N} \xrightarrow{l} l : \overline{M(x, \bar{y})} \mapsto \bar{\mathcal{N}}$	Propagate token from one of the incoming edges to term \mathcal{N} .

MRG-3	$\mathcal{A}[l : M(x, y) \mapsto \mathcal{N}, \overline{l\mathcal{X}}] \xrightarrow{l} \mathcal{A}[l : M(\overline{x}, y) \mapsto \mathcal{N}, l\mathcal{X}]$	MRG-1 in a sub-term.
MRG-4	$\frac{\mathcal{N} \xrightarrow{\alpha_p} \mathcal{N}'}{\overline{l : M(x, y) \mapsto \mathcal{N}} \xrightarrow{\alpha_p} \overline{l : M(x, y) \mapsto \mathcal{N}'}}$	Change of a merge term when \mathcal{N} changes.
JON-1	$\forall n \geq 0, l : \overline{\mathcal{N}} \mapsto l' : J(x, y)^n \xrightarrow{l} l : \overline{\mathcal{N}} \mapsto l' : J(\overline{x}, y)^n$	Activate pin x from an outgoing join.
JON-2	$l : J(\overline{x}, \overline{y}) \mapsto \mathcal{N} \xrightarrow{l} l : J(x, y) \mapsto \overline{\mathcal{N}}$	Activate pin x from an incoming join to \mathcal{N} .
JON-3	$\mathcal{A}[l : J(x, y) \mapsto \mathcal{N}, \overline{l\mathcal{X}}] \xrightarrow{l} \mathcal{A}[l : J(\overline{x}, y) \mapsto \mathcal{N}, l\mathcal{X}]$	Fuse labels referring to the same join.
JON-4	$\frac{\mathcal{N} \xrightarrow{\alpha_p} \mathcal{N}'}{\overline{l : J(x, y) \mapsto \mathcal{N}} \xrightarrow{\alpha_p} \overline{l : J(x, y) \mapsto \mathcal{N}'}}$	Change of a join term when sub-term \mathcal{N} changes.
SND	$\forall n > 0, \overline{l : a!v^n} \mapsto \mathcal{N} \xrightarrow{l} \overline{l : a!v^{n-1}} \mapsto \overline{\mathcal{N}}$	Propagation of the token after sending object v .
REC	$\forall n, m \geq 1, \frac{\overline{l' : a!v^m} \mapsto \mathcal{N}' \xrightarrow{l'} \overline{l' : a!v^{m-1}} \mapsto \overline{\mathcal{N}'}}{\overline{l : a?v^n} \mapsto \mathcal{N} \xrightarrow{l} \overline{l : a?v^{n-1}} \mapsto \mathcal{N}}$	Token flow when receiving object v after sending, which is a synchronous communication.
COM	$\forall n, m > 0, \frac{\overline{l : a!v^n} \mapsto \mathcal{N}_1 \xrightarrow{l} \overline{l : a!v^{n-1}} \mapsto \overline{\mathcal{N}_1}, \overline{l' : a?v^n} \mapsto \mathcal{N}_2 \xrightarrow{l'} \overline{l' : a?v^{n-1}} \mapsto \overline{\mathcal{N}_2}}{\mathcal{A}[\overline{l : a!v^n} \mapsto \mathcal{N}_1, \overline{l' : a?v^n} \mapsto \mathcal{N}_2] \xrightarrow{l} \mathcal{A}[\overline{l : a!v^{n-1}} \mapsto \mathcal{N}_1, \overline{l' : a?v^{n-1}} \mapsto \mathcal{N}_2]}}$	Sending and receiving object v .
FFin	$\mathcal{A}[l : \overline{\otimes}] \xrightarrow{l} \mathcal{A}[l : \otimes]$	Remove token when a flow final node is reached.
AFin	$\mathcal{A}[l : \overline{\odot}] \xrightarrow{l} \mathcal{A} $	Remove all tokens when activity final is reached.
PRG-1	$\frac{\mathcal{N} \xrightarrow{\alpha_p} \mathcal{N}'}{\mathcal{A}[\mathcal{N}] \xrightarrow{\alpha_p} \mathcal{A}[\mathcal{N}']}$	Apply another other rule to a sub-term.
PRG-2	$\frac{\mathcal{N}_1 \xrightarrow{\alpha_{p_1}} \mathcal{N}'_1, \mathcal{N}_2 \xrightarrow{\alpha_{p_2}} \mathcal{N}'_2}{\mathcal{A}[\mathcal{N}_1, \mathcal{N}_2] \xrightarrow{\alpha_{p_1 \times p_2}} \mathcal{A}[\mathcal{N}'_1, \mathcal{N}'_2]}}$	Apply another rule on two sub-terms.

Table 2.3: NuAC derivation rules and their conceptual meaning[23].

2.3.3 SysML Activity Diagrams to TorXakis Translation Approach

In her document [13], Kyra Hameleers, has documented her findings on translating a SysML activity diagram to TORXAKIS. Even though this procedure is not formally defined, this document is relevant for our research, because the approach described in [13] is based on the SysML models of ADAPTO, which we will be using for our case study. Furthermore, TORXAKIS is one of the contending formal languages for using MBT with SysML we will explore in this thesis.

The approach consists of the *boilerplate* code, the code that can be generated independently of the activity diagram, and the translation algorithm. This part consist of two sections: translating diagrams without parallelism and with parallelism.

Boilerplate

To get a TORXAKIS model up and running, the communication channels and how they connect and communicate with our adapter and SUT need to be defined. To define the channels, the type of the messages used for the communication needs to be chosen. One option is to use the `String` type, which is chosen, because this option gives the most flexibility since messages are not restricted. The channel definition can be found in listing 2.1.

```

1 CHANDEF Channels ::= In :: String
2                               ; Out :: String
3 ENNDEF

```

Listing 2.1: Channel definition. Over the channels strings can be communicated

To define how we communicate with the adapter, a connection definition needs to be created. Since TORXAKIS and the adapter will be run on the same computer, TORXAKIS is hosted locally. The unoccupied port 7890 is chosen for the communication. The resulting connection definition is laid out in listing 2.2.

```

1 CNECTDEF Sut
2   ::=
3     CLIENTSOCK
4
5     CHAN OUT In          HOST "localhost" PORT 7890
6     ENCODE In ? s      -> ! s
7
8     CHAN IN Out        HOST "localhost" PORT 7890
9     DECODE Out ! s    <- ? s
10 ENNDEF

```

Listing 2.2: Connection definition. Port 7890 is used to send and retrieve messages

TORXAKIS needs a model to test. The model definition can be created in advance, in which the translated activity diagram will be executed. The starting procedure will be called `start_process`. This procedure will contain the actual translations. The resulting model can be found in listing 2.3.

```

1 MODELDEF Model
2   ::=
3     CHAN IN In
4     CHAN OUT Out
5

```

```

6     BEHAVIOUR start_process[In, Out]()
7 ENDDDEF

```

Listing 2.3: Model definition

Now that we have defined the boilerplate code, we can start the translation process. There are two options for translating. If the diagram has a linear control flow i.e. does not contain a Fork-, Join-, Decision-, or Merge node, the *Linear* translation option is used. Otherwise, the *Non-Linear* translation option is used.

Linear

When the to be translated SysML activity diagram is linear, i.e. does not contain parallelism or non-linear control flow, which can be recognized by the parallelize or synchronize construct, as can be seen in figure 2.17 *or* decision or merge symbols, as can be seen in figure 2.18.

In this case the State Automaton Definition, or **STAUTDEF** in TORXAKIS can be used. This definition corresponds with the mathematical Labelled Transition System (LTS). To define a state automaton definition, the possible states, the transitions between these states and the initial state are needed. A **STAUTDEF** also supports variables. The translation procedure goes as follows.

- We start at the initial activity. From here, there should be a linear path through the diagram, since we did not encounter parallelization constructs. We will call the initial activity `init` and we will add it to the **STATE** and **INIT** sections of the state automaton definition.
- We can now consider each arrow (control flow) between the activities as a state. Since these arrows are not named, we need to choose a unique name. We choose to call the state the same as the transition name, but starting with a lowercased letter.
- Each state name has to be added to the **STATE** section of the state automaton definition.
- For each action construct, we need to add a transition to the **TRANS** section of the state automaton definition. To create a transition, we need the source state, the target state and the communication. The communication consists of a channel and the message that will be sent over that channel. For the **Send** signal action (figure 2.15), the **Output** channel is used, because a signal is outputted. Otherwise, the **Input** channel is used, because the termination of any action is the input for the next action. This also holds for the **Receive** signal action, because this action is started by an input trigger.
- When the final activity is reached, we are done.

Listing 2.4 shows a complete state automaton definition that result from these steps.

```

1 STAUTDEF start_process[In :: String; Out :: String]()
2   ::=
3     STATE
4       init, activity1, activity2
5     INIT
6       init
7     TRANS
8       init      -> In ! "Activity1" -> activity1

```

```

9      activity1 -> Out ! "Activity2" -> activity2
10  ENDDEF

```

Listing 2.4: State automaton definition.

Non-Linear

When the to be translated SysML activity diagram is non-linear, a different procedure is required. A non-linear activity diagram contains one or more of parallelism or control flow symbols which can be recognized by symbols in figures 2.17 and 2.18.



Figure 2.17: Symbols used for parallelism



Figure 2.18: Symbols used for control flow

Instead of defining a `STAUTDEF`, we define the model using the process definition `PROCDEF`. A process definition in TORXAKIS supports parallelism and control flow transitions. The following steps need to be taken in the updated translation procedure:

1. When we encounter the parallelize symbol, shown in figure 2.17a, we split up the consecutive paths and translate these recursively. We connect these paths with the parallel operator `|||`.

When we encounter the synchronize symbol, shown in figure 2.17b, we have reached the end of the parallelization. To show this in TORXAKIS, we use the `EXIT` keyword for each of the paths that are synchronized. We need to mark the recursive translations with the keyword `EXIT` as well so that TORXAKIS knows that this procedure can end. We use the `>>>` operator to concatenate two procedures. In this case, we use the operator to continue after we have synchronized. This operator can only be used when the preceding process is marked with the `EXIT` keyword.

The result can be found in listing 2.5.

```

1  PROCDEF parallel_procedure[In :: String; Out :: String]() ::=
2    (left_path[In, Out]() ||| right_path[In, Out]()) >>> continuation
3  ENDDEF
4  PROCDEF left_path[In :: String; Out :: String]() EXIT ::=
5    -- recursive translation
6    >>> EXIT
7  ENDDEF
8  PROCDEF right_path[In :: String; Out :: String]() EXIT ::=

```



```

9      -- recursive translation
10     >>> EXIT
11     ENDDDEF
12

```

Listing 2.5: Parallel procedure

2. The process is similar for the control flow symbols, but we use the operator `##` to indicate that either the left or the right path can be chosen. When the decision node contains a guard in one or more of its outgoing paths, we can use the guard statement in TORXAKIS. The guard statement is a boolean expression between double scare brackets. It connects to the procedure it is guarding for with the `=>>` operator. Assuming the left path does not have a guard but the right one does, the result will look like listing 2.6.

```

1     PROCDEF decision_procedure[In :: Transition; Out :: Transition]() ::=
2         (left_path[In, Out]() ||| ([[ guard ]] =>> right_path[In, Out]()))
↪     >>> continuation
3     ENDDDEF
4     PROCDEF left_path[In :: Transition; Out :: Transition]() EXIT ::=
5         -- recursive translation
6         >>> EXIT
7     ENDDDEF
8     PROCDEF right_path[In :: Transition; Out :: Transition]() EXIT ::=
9         -- recursive translation
10        >>> EXIT
11    ENDDDEF
12

```

Listing 2.6: Parallel procedure with a guard on the right branch

3. There are two different SysML constructs used to indicate the end of the procedure. The Flow Final node (2.15) indicates that the path has reached its final state. The Activity Final node (2.15) indicates that the entire activity diagram has reached its final state.

All actions in the activity diagram are translated to TORXAKIS according to the *Linear* approach. To indicate that the translated actions are executed in order, they are joined with the TORXAKIS sequence operator (`>->`).

This concludes the translation procedure.

Example

To illustrate the effect of applying the translation procedure, we apply the translation procedure on the activity diagram shown in figure 2.19.

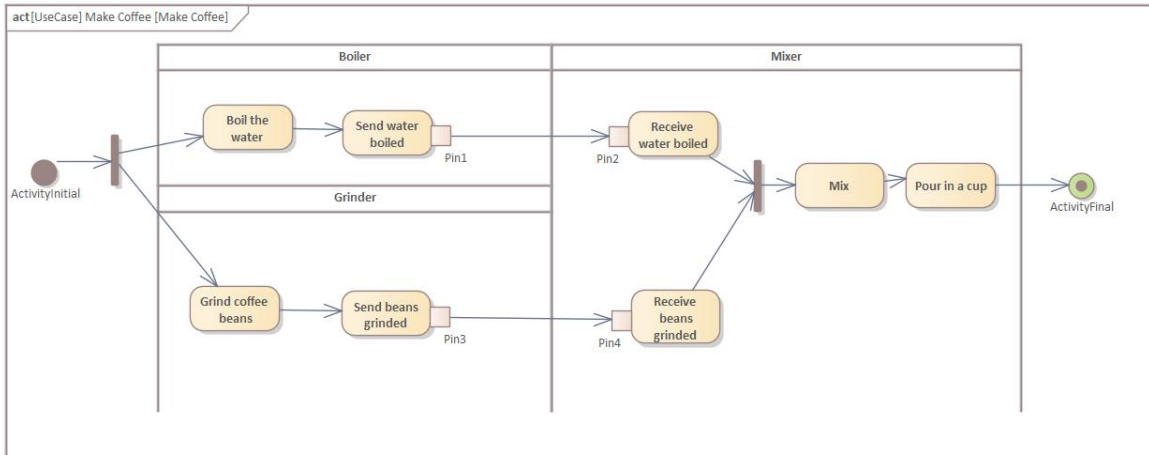


Figure 2.19: Coffee machine activity diagram

We observe that the coffee machine activity diagram uses parallelization, which can be recognized by the fork node and synchronize node. Therefore, we will use the non-linear translation procedure. Listing 2.7 shows the result of the application of the non-linear translation procedure on the coffee machine activity diagram.

```

1 PROCDEF Make_Coffee[In, Out :: Transition]() EXIT
2 ::=
3     (parallel_1[In, Out]() ||| parallel_2[In, Out]())
4     >>> continuation[In, Out]()
5 ENDDF
6
7 PROCDEF parallel_1[In, Out :: Transition]() EXIT
8 ::=
9     In ! "Boil the water"
10    >-> In ! "Send water boiled"
11    >-> In ! "Receive water boiled"
12    >-> EXIT
13 ENDDF
14
15 PROCDEF parallel_2[In, Out :: Transition]() EXIT
16 ::=
17    In ! "Grind coffee beans"
18    >-> In ! "Send beans grinded"
19    >-> In ! "Receive beans grinded"
20    >-> EXIT
21 ENDDF
22
23 PROCDEF continuation[In, Out :: Transition]() EXIT
24 ::=
25    In ! "Mix"
26    >-> In ! "Pour in a cup"
27    >-> EXIT
28 ENDDF
29
30 CHANDEF Channels ::= In :: String
31                  ; Out :: String
32 ENDDF
33

```

```

34 MODELDEF Model
35 ::=
36     CHAN IN      In
37     CHAN OUT    Out
38
39     BEHAVIOUR   Make_Coffee [In,Out] ( )
40 ENDDDEF
41
42 CNECTDEF Sut
43 ::=
44     CLIENTSOCK
45
46     CHAN OUT In          HOST "localhost" PORT 7890
47     ENCODE   In ? qop    -> ! qop
48
49     CHAN IN  Out        HOST "localhost" PORT 7890
50     DECODE   Out ! s    <-  ? s
51 ENDDDEF

```

Listing 2.7: Result of translating the coffee machine activity diagram from figure 2.19 using the non-linear translation procedure.

After applying the translation procedure, we observe the following:

1. It depends on the interpretation of the viewer of the activity diagram what action can be categorized as an output, because this is not specified in the activity diagram. The ‘*Pour in a cup*’ action for example could be interpreted as an observable output from the coffee machine, but this is not expressed in the activity diagram and is therefore translated to an input.
2. It depends on the interpretation of the viewer of the activity diagram what action can be categorized as an input or as an internal action, because this is not specified in the activity diagram. The ‘*Send water boiled*’ action for example could be interpreted as an internal action from the coffee machine, because it might not be observable from the outside. Since this is not expressed by the activity diagram, the action is translated to an input.

When defining our own translation algorithm, we take these observations into account.

2.4 ADAPTO

In this section, we will explain what ADAPTO is and how it works. ADAPTO is an automated storage and retrieval system developed by Vanderlande. ADAPTO consists of storage racks with rails in between them. Battery powered *shuttles* move over these racks to pickup items at specific instructions communicated and received by the shuttles over Wi-Fi. Figure 2.20 shows how a shuttle picks up an item from a rack. At the front of the racks is a crossrail on each level. This allows for shuttles to move to different aisles on the same level. ADAPTO can be configured to have a fixed number of shuttles per level *or* to allow shuttles to change levels via shuttle lifts.



Figure 2.20: A single ADAPTO shuttle

Usually ADAPTO is used in a large warehouse that needs to store and retrieve items efficiently from its large storage facility. ADAPTO solves the following problems for its customers:

- Items are automatically retrieved and stored in the system. This saves on manual labour.
- Since ADAPTO works automatically, it is less prone to errors when storing and retrieving orders.
- More items can be stored in the same volume than in a traditional warehouse where items need to be reachable by humans. Figure 2.22 shows that ADAPTO is scalable in all three dimensions and that storage is very dense.

Figure 2.21 shows how an aisle of ADAPTO looks like, including the item elevator.

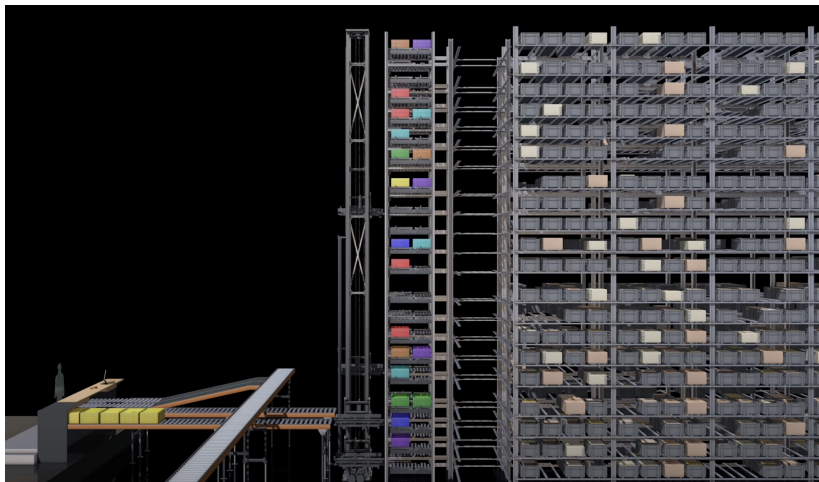


Figure 2.21: Side view of a single aisle of ADAPTO. The shuttles retrieve an item in the warehouse and delivers it to the item elevator.

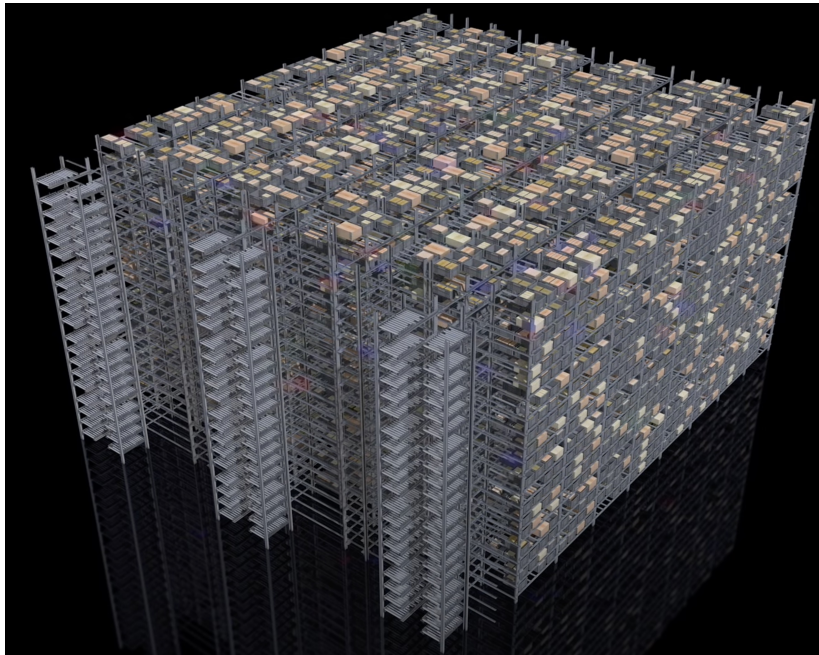


Figure 2.22: ADAPTO is scalable

2.4.1 Testing

ADAPTO is currently used in multiple facilities in several countries with the Netherlands as the greatest market and this number is growing. To save costs on customer service and to appeal to potential customers, Vanderlande profits from having reliable software. To make sure their software and hardware systems are reliable, they test it.

Currently, ADAPTO is tested using three main concepts, namely automatic gherkin tests, endurance testing and manual testing.

Automatic Gherkin Tests

The first way ADAPTO is tested, is with automatic tests. Gherkin tests are used for this. They are handwritten tests that can be executed automatically. The goal of a Gherkin test is to test a flow of the tested system while making it understandable for non-technical people to understand the test. This is done by defining a test as a list of sentences that resemble natural language. Each sentence represents one of three possible clauses which can occur in a Gherkin test[19].

The possible clauses and their meaning from the ADAPTO Gherkin tests are:

- The **Given** clause indicates a precondition.
- The **When** clause indicates an action done on the SUT.
- The **Then** clause describes the expected post condition of the SUT. This statement is checked to check if the test passes.

- The **And** keyword has the same semantics as the keyword from the line above in the test. This works recursively. This keyword is used to make the test cases more similar to the English language and easier to read for people without a technical background.

A Gherkin test is a list of clauses. For ADAPTO, each clause in this list is executed in sequence. The next clause starts only when the current clause is finished executing. This also holds for the statements of lines 5, 6 and 7; these states are reached after each other in sequence.

An example of a Gherkin test can be found in listing 2.8.

```

1 Scenario: Make coffee
2   Given coffee machine is started
3   When pressing the start button
4   Then coffee is poured in a cup

```

Listing 2.8: Gherkin test that describes a simple coffee machine.

To make a Gherkin test actually work, additional steps have to be undertaken, since a computer can not yet understand such a Gherkin test and use it on the SUT. Regular expressions are used to link each line in a Gherkin test to a function in the ADAPTO software. Listing 2.9 shows an example of such a regular expression.

```

1 [When(@"(pressing|holding) the start button with id ""(.*)""")]
2 public void InteractingWithStartButton(InteractionType
3     ↪ interactionType, int id)
4 {
5     // Perform the corresponding actions on the SUT.
6 }

```

Listing 2.9: Regular expression that links the clause on line 3 of listing 2.8 to an action on the SUT.

The body of the function performs the associated actions of the Gherkin clause on the SUT. Now the Gherkin test can be executed.

For ADAPTO, each of the linked functions corresponds to a predefined action on the SUT. These actions are called the step definitions and together these form an abstraction layer over the underlying implementation. These step definitions are used in all parts of the test automation where actions on the SUT are communicated.

We will analyze the structure of a Gherkin test and how it is related to an MBT test, because we found similarities between Gherkin tests and MBT tests that we can use in our case study. We do this to better understand the semantics of a Gherkin test. This way, a Gherkin test written for ADAPTO may be used in combination with SysML models for MBT. As we have seen, a Gherkin test contains three possible clauses.

- **Given**
This clause describes the assumed precondition of the test. We have two options when translating this clause into an MBT test. We can make sure that the SUT satisfies the precondition before we start the test which allows us to ignore this clause. The other option is to consider this clause an input which configure the SUT according to the precondition.

- **When**
This clause describes an input on the ADAPTO system. In model based testing, this would be an input as well.
- **Then**
This clause describes the expected post condition of ADAPTO. In model based testing, we can consider this an output by observing when the post condition holds[19].

Endurance Testing

The other way ADAPTO is currently tested is with endurance testing. Endurance testing is running a system how it normally would be used for a longer period of time. When running a system for a longer period, it is more likely to enter rare states. This way edge cases are encountered and therefore tested which, if the system keeps running, indicates a reliable system[16]. If a failure occurs during an endurance test, the problem can not only be in the software, but also in the hardware. Some hardware failures are taken into account by the software, but significant hardware defects or failures are not. For example, when ADAPTO stops running after an earthquake and the whole system collapses, the software cannot realistically be held accountable.

At Vanderlande an ADAPTO system is running most of the time and is checked regularly to check if the system is still running as usual.

2.4.2 SysML at Vanderlande

Currently, Vanderlande is in the process of modelling SysML models. At the moment, *System/Subsystem Design Descriptions* (SSDDs) are used to design the behaviour of the to be implemented software. These SSDDs are used as a basis for translation to SysML. For future system design, SysML models will be designed directly from the determined system behaviour. The modelling of SysML models is outside the scope of this thesis. The goal of the SysML modelling is to use the models for the software development, because the models are clearer than SSDDs. In addition to this, the SysML models can be used to explain (future) employees how the ADAPTO system works by providing them a visual overview with the models as opposed to explaining in natural language or by showing code. At the moment the development of the SysML models is still in an experimental phase.

Not all diagrams of SysML are used to model ADAPTO. Figure 2.23 shows the 9 SysML diagrams structured by the type of diagram from figure 2.8 with the diagrams that are used in the SysML models of ADAPTO coloured in.

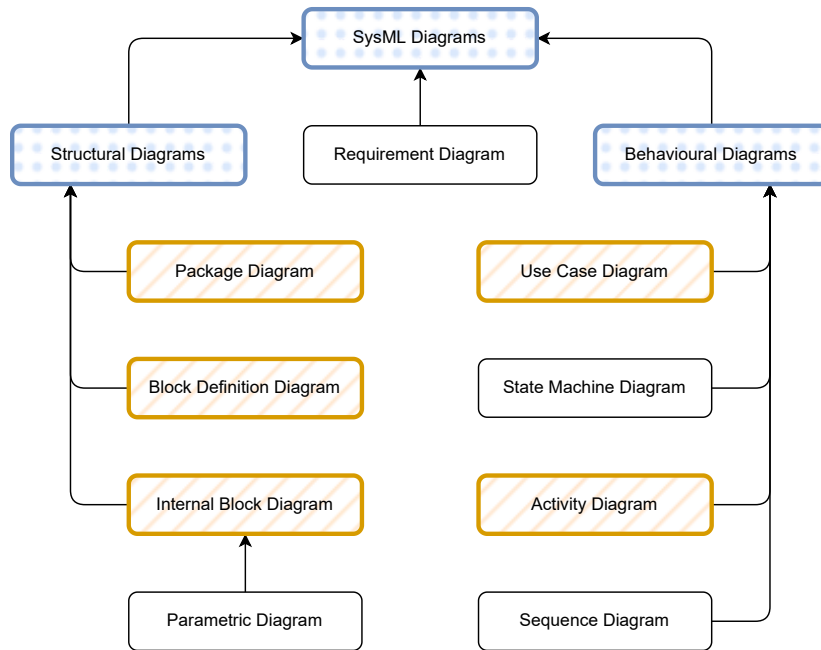


Figure 2.23: SysML diagrams grouped by their type[14]. The orange coloured and striped diagrams are used in the SysML models of ADAPTO. The blue coloured and dotted blocks indicate a group (or an interface in UML).

Activity diagrams are used for modelling the behaviour of ADAPTO in favour of state machine diagrams, because activity diagrams allow for modelling parallel behaviour in a system, which the SysML models of ADAPTO takes advantage of. According to [12], activity diagrams are reducible to state machine diagrams. This implies that state machine diagrams can model parallel behaviour as well. Modelling parallel behaviour in state machine diagrams is however less trivial than in activity diagrams. Furthermore, Vanderlande has decided to model case studies for ADAPTO, which is better suited for activity diagrams, in contrary to state machine diagrams which are generally used to model the behaviour of a specific system component.

2.5 Conclusion

In this section, we conclude the background by providing a summary including how each part of the background will be used in the remainder of this thesis.

In section 2.1, we introduced the underlying principles of Model-Based Testing. We started with a Labelled Transition System; the datastructure used in Model-Based Testing. With Model-Based Testing, we want to check if the behavioural model of the SUT (specification) matches its implementation. The specification and implementation can both be considered as an LTS. To determine whether one LTS behaves according to another LTS, a relation needs to be defined. We covered trace equivalence, *ioco* and *uioco*. Model-Based Testing tools implement such a relation. We have covered four Model-Based Testing tools. One of which is TORXAKIS, which implements the *ioco* relation.

In section 2.2, we discussed SysML and the nine diagrams that make up a SysML model. Since the goal of our research is to use SysML models for Model-Based Testing, we discussed which SysML diagrams are best suited for Model-Based Testing. We concluded that at least a SysML diagram that models behaviour should be used, since Model-Based Testing tests behaviour. Depending on the SysML model, it should be determined which (behavioural) diagram(s) should be used for Model-Based Testing.

In section 2.3 we discussed related work for the research topic of this thesis. We discussed that SysML state machine diagrams have been formalized and used for Model-Based Testing before in the FormaSig project. We used the research of FormaSig to get an idea of how a SysML diagram could be formalized for Model-Based Testing, and to compare the translation approach to our own approach. We also looked at two papers that formalized the SysML activity diagram. [18] proposes a grammar for the activity diagram and derivation rules on the grammar terms to formalize the behaviour. [23] expands on this work by supporting all SysML activity diagram constructs. We will use these papers as a basis for our translation algorithm in section 3.4 where we will use the same concept of defining a grammar defining the semantics of each term. Moreover, we laid out the translation procedure that was created at Vanderlande by manually translating SysML activity diagrams to TORXAKIS. We propose an enhancement for this procedure in section 3.3.

In section 2.4 we explained what ADAPTO is, which components there are and what functions these components have. We laid out how ADAPTO currently gets tested. We focused on Gherkin tests, since we will use these tests in a case study in section 4.3.1 where we attempt to translate Gherkin tests of ADAPTO to TORXAKIS. Finally, we discussed how SysML models are modelled for ADAPTO. We will use this information for determining the translation source of our translation algorithm from SysML to an Model-Based Testing tool in section 3.1.

Chapter 3

Translation from SysML to Model-Based Testing

In this chapter, we will define a translation algorithm from SysML models to a translation target that can be used for Model-Based Testing (MBT). Before we define this algorithm, we first determine what translation source and translation target are best suited to achieve our goal: applying MBT on systems with SysML models. The algorithm we define is based on previous work from [18], [23] and [13], but is adapted to suit our translation. Furthermore, we propose an enhancement to translation procedure in [13].

3.1 Translation Source

From section 2.2.1, we learn that there are 9 types of diagrams a SysML model can consist of. Four of these diagrams are used to model the behaviour of the system. Since MBT requires a behaviour model of the System Under Test (SUT), we naturally want to incorporate at least one of the behavioural diagrams of SysML.

In the FormaSig project[28], the SysML diagrams that are used for the translation to mCRL2 are the Internal Block Diagram, the Block Definition Diagram and the State Machine Diagram. The first two diagrams model the structure of the SUT while the latter one describes the behaviour. State machine diagrams are chosen for the formalization in mCRL2, because that is the main diagram that is used to model behaviour of the EULYNX system in the FormaSig project.

For our case study, we will apply our algorithm on the ADAPTO system. The SysML models that are developed for ADAPTO use the Activity Diagram (AD) to model the behaviour. Section 2.4.2 explains why the decision to use ADs is made. Since the AD is the main diagram used to model behaviour for ADAPTO, we choose to use this diagram as our basis for the translation algorithm.

3.2 Translation Target

We consider the following five translation target contestants for our translation. Since the goal is to use model based testing, we naturally need to choose a target that supports this.

1. Labelled Transition System
2. Symbolic Transition System
3. mCRL2
4. Axini
5. TORXAKIS

3.2.1 Labelled Transition System

Labelled transition systems (LTSs) are the underlying models of the model based testing theory (section 2.1.1). For flexibility purposes, it would make sense to formalize a SysML activity diagram to an LTS. Hereafter, this formalized activity diagram can be converted to a modelling language that an MBT tool accepts as an input.

Pros

The ability to choose an MBT tool is the main benefit of this approach.

Cons

The cons of using LTSs as the target of our translation are:

1. The flexibility is also the main drawback of this approach; the resulting LTS needs to be translated to a language that an MBT tool understands before we can apply MBT.
2. Usually, complex software systems use large or unbound data domains which results in a state space explosion. This can result in a long runtime for a Model-Based Testing tool.

3.2.2 Symbolic Transition System

Symbolic transition systems counter the problem of a state space explosion that LTSs have. This is done by treating data symbolically. This means that data is stored outside the transition system. A transition can alter this data or can have a guard which is used to check if that transition is allowed according to the stored data. If not specified, the initial value of an integer value is 0.

Figure 2.6 shows an STS of a vending machine. It only outputs a *chocolate bar* after the button is pressed when there is enough money inserted. The figure also shows the guard `money >= 10`, which indicates that that transition can only be taken if the variable `money` is greater than or equal to 10. The action `money- = 10` is also shown in the figure. This actions indicates that each time the transition is performed, the `money` variable is decremented by 10.

This STS has three states. The LTS equivalent would have significantly more states if there are just a few different coins that can be inserted. This is because the state in the LTS would represent the *money* variable in the STS. If a coin could have any integer value, an LTS for this STS would not even exist, because the LTS would need a countably infinite number of states; at least one for each value the coin can have.

Pros

The pros of using STSs as the target of our translation are:

1. As with LTSs, choosing STSs as the source of the translation algorithm ensures that any MBT tool can be used.
2. As stated above, STSs counter the problem of state space explosion that LTSs have.

Cons

As with LTSs, an STS needs to be translated to a language that an MBT tool understands before we can apply MBT.

3.2.3 mCRL2

The mCRL2 language is a formal language that describes models. A toolkit is included that can do model checking. To use mCRL2 for model based testing, the JTorX model based tester is a popular choice since this application can read mCRL2 as input. As Van der Wal and Bouwman explain in their research, it is possible to translate the EULYNX SysML models to mCRL2[28]. The translation in mCRL2 would be the process calculus that works on the derivation rules of (Nu)AC.

Pros

The pros of using mCRL2 as the target of our translation are:

1. No additional translation is needed, provided that you use the JTorX model based tester.
2. There is currently a tool in development by Van der Wal and Bouwman that translates SysML to mCRL2 developed. However, as of writing this, this tool is not available yet and is only applicable on state machine diagrams. We did not consider this approach in this thesis, because we opted for another translation target.

Cons

The cons of using mCRL2 as the target of our translation are:

1. JTorX is the predecessor of TORXAKIS and has not been updated and maintained for several years.
2. It is not straightforward to use a different model based testing tool for mCRL2 models unless an additional translation layer is developed to translate these models to the language of a different tool. Since this is the question that we are trying to answer with this comparison, this requirement defeats the purpose of using mCRL2 in the case we do not want to use JTorX.

3.2.4 TorXakis

TORXAKIS is initially developed for research purposes. The model based testing tool is based on the *ioco* theory introduced by Tretmans[27]. The translation in the TORXAKIS modelling language would be the process calculus that works on the derivation rules of (Nu)AC. We did not consider the MBT tool TORX, because it is the predecessor of JTORX and TORXAKIS.

Pros

The pros of using the TORXAKIS language as the target of our translation are:

1. TORXAKIS is open source.
2. TORXAKIS is free for commercial use under the BSD 3-Clause License.
3. Preceding work from [13] has been done inside Vanderlande which is based on TORXAKIS. This work can be used as a basis for our translation algorithm.

Cons

The cons of using the TORXAKIS language as the target of our translation are:

1. Since the software is free to use, there is no customer support available.
2. As we learned from 2.1.7, TORXAKIS does not have the ability of modelling time. Since the activity diagrams of ADAPTO occasionally use timer constructs, it can be an issue that modelling time is not supported in TORXAKIS. One option to overcome this limitation is by ignoring the occasional timer constructs in the activity diagrams. Another solution is to allow a transition that has a timer, but increase the time TORXAKIS waits on the next output to at least the time the timer waits.

3.2.5 Axini

Axini (section 2.1.7) is a modern model based testing tool developed for commercial use.

Pros

The pros of using the Axini language as the target of our translation are:

1. In addition to modelling expressiveness of TORXAKIS, Axini allows for the modelling of time. This could for example be used to check for an output after a given amount of time.
2. Since Axini is developed for commercial use, the tool is maintained and updated regularly.
3. Support is available from the Axini team.

Cons

The cons of using the Axini language as the target of our translation are:

1. The tool is not open source and can therefore not be inspected. This might be a con when we are concerned about the behaviour of the underlying source code.
2. The tool is not free to use.

Considering all observations and preceding academic work, we have decided to build on [18] and [23] by proposing a translation algorithm that is suitable for translating (ADAPTO) SysML activity diagrams to TORXAKIS. Our algorithm will consist of two steps. First, we define a grammar that is able to encode all information from SysML activity diagrams which is required for the translation. After this, we need to translate this intermediate representation to a modelling language. We have chosen to translate to the TORXAKIS modelling language, because the result of the translation can directly be used by an MBT tool without having to perform an additional translation step, unlike LTSs and STSs *and* because TORXAKIS has a lower entry barrier than Axini because Vanderlande already has some knowledge regarding using TORXAKIS and TORXAKIS is freely available. Furthermore, we have chosen to use the second option for solving the presence of timer constructs in ADAPTO activity diagrams: by configuring TORXAKIS to wait long enough for outputs that may only occur after a set timeout. This answers **SQ1**.

Now that we have established our translation source and target, we will adapt preceding work to make it suitable for our use case: MBT with TORXAKIS. We do this by deriving an updated formalization for MBT with TORXAKIS from [18] and [23] after we have proposed an enhancement to the translation procedure from [13]. This is done to gain insight in the translation procedure and to provide Vanderlande with a more maintainable translation procedure if they decide to continue using the procedure. These two translation procedures will be used in the case study in chapter 4.

3.3 Activity Diagram Translation Enhancement

In this section, we propose an enhancement to the SysML activity diagram to TORXAKIS translation procedure from [13]. We do this to gain insight in the translation procedure which could be beneficial for our subsequent translation methods and to provide Vanderlande with a more maintainable translation procedure if they decide to continue using this translation procedure.

In the procedure, **Strings** are used for the communication channels. While this does offer flexibility for the messages that are communicated between TORXAKIS and the adapter, this flexibility comes at a cost. Since we can send anything we want to the adapter, it is also possible to send illegal instructions. To make our communication type-safe, we have decided to declare the type **TYPEDEF Transition** and use this type for our channels. This way it has to be explicitly stated which messages are allowed to be sent in the communication with the adapter. This can be done by adding a constructor to the **Transition** type. The updated channel definition can be seen in listing 3.1.

```

1 CHANDEF Channels ::= In :: Transition
2                               ; Out :: Transition
3 ENNDEF

```

Listing 3.1: Channel definition. Over the channels only messages with the **Transition** type can be communicated

The introduction of the **Transition** type also implies that the connection definition needs to be adapted. Since we would like to continue to ensure type-safety, we need to encode the outgoing messages and decode the incoming messages. Since the transitions may contain arguments, it is preferred to structure our data. TORXAKIS has XML encoding and decoding

built in with the respective functions `toXml` and `fromXml` which we will use. The adapted connection definition is laid out in listing 3.2.

```

1 CNECTDEF Sut
2   ::=
3     CLIENTSOCK
4
5     CHAN OUT In          HOST "localhost" PORT 7890
6     ENCODE      In ? s   -> ! toXml(s)
7
8     CHAN IN  Out        HOST "localhost" PORT 7890
9     DECODE    Out ! s   <- ? fromXml(s)
10 ENDDDEF

```

Listing 3.2: Connection definition. Using port 7890, messages

Now that the channels communicate `Transitions` instead of `Strings`, we need to make some adjustments to the translation procedure.

When translating actions in the activity diagram, for each action we need to add a constructor to the `Transition` type. Since the translation procedure does not consider data flow between actions, these actions do not require arguments which allows us to use the name of the action with all whitespaces replaced with underscores to comply to the grammar of TORXAKIS.

Listing 3.3 shows what the `Transition` type looks like after translating the actions *Activity1* and *Activity2*.

```

1 TYPEDEF Transition
2   ::=
3     Activity1
4     | Activity2
5 ENDDDEF

```

Listing 3.3: Type safe transitions with the `Transition` type.

3.4 Translation Algorithm

We define an algorithm that systematically can translate a SysML activity diagram to the TORXAKIS modelling language. We do this by first transforming the graph of the activity diagram into an *intermediate representation*. This intermediate representation is a formal language based on the grammars 2.12 and 2.16 from the Activity Calculus (AC)[18] and the New Activity Calculus (NuAC) [23] respectively where the SysML activity diagram has been formalized. We combine these grammars to suit our needs. The choices we made for our *Adapted Grammar* are:

- We want to model the semantics of the activity diagram in the TORXAKIS modelling language. This means that the grammar does not have to support tokens, because the state does not have to be kept track of; we only want to define the structure of the activity diagram in the grammar. TORXAKIS keeps track of the state. Therefore, we only need to consider the unmarked grammar from figure 2.12a.
- We will add the Send Signal, Receive Signal and Call Behaviour activity diagram constructs considered in [23] to our grammar to make our solution more complete and versatile and to make it compatible with the activity diagrams of ADAPTO.

- We omit the probabilistic decision term, because an MBT internally decides with what probability each path can be chosen.
- We remove empty term ϵ , because we do not allow empty activity diagrams.
- The grammar does not support Decision or Fork nodes with more than two outgoing edges. This does not limit the terms we can create, because a term (fork or decision) with n outgoing edges is semantically equivalent to chaining this term $n - 1$ times. An example of this can be seen in figure 3.1. This can be proven with the **FRK-1** rule from table 2.3 which describes that a token incoming on a fork propagates a token to each outgoing edge.
- The grammar does not support multiple incoming edges to an action. This notation is sometimes used as shorthand for a merge node, so we can fix this issue by adding a merge node when an action has multiple incoming edges. This operation is shown in figure 3.2.

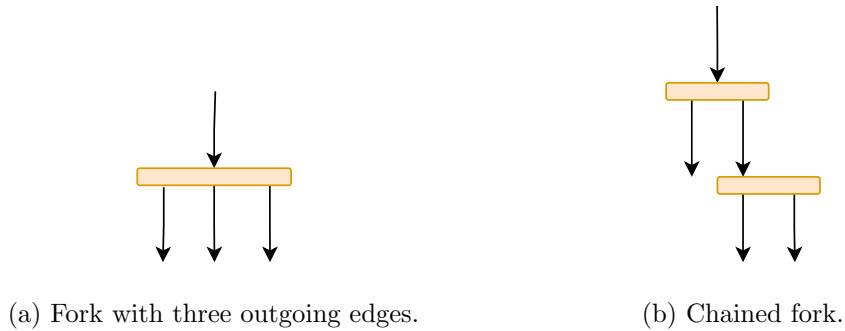


Figure 3.1: The fork with three outgoing edges is semantically equivalent to the chained fork.

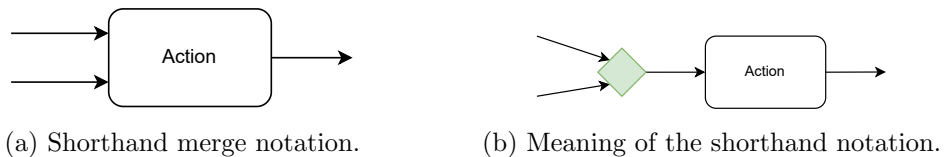


Figure 3.2: The adapted grammar uses the extended notation when a merge is written in the shorthand notation.

The resulting adapted grammar can be found in figure 3.3. The terms *Merge*, *Join*, *Fork* and *Decision* are allowed to be abbreviated to *M*, *J*, *F* and *D* respectively for space constraints.

\mathcal{A}	$::=$	$a.t \mapsto \mathcal{N}$
\mathcal{N}	$::=$	$l : \otimes$
		$l : \odot$
		$l : Merge(\mathcal{N})$
		$l : Join(\mathcal{N})$
		$l : Fork(\mathcal{N}, \mathcal{N})$
		$l : Decision(\mathcal{N}, \mathcal{N})$
		$l : \mathcal{B} \mapsto \mathcal{N}$
		l
\mathcal{B}	$::=$	a
		$!a$
		$?a$
		$\uparrow a.\mathcal{A}$

Figure 3.3: Adapted Grammar

Table 3.1 shows the SysML activity diagram constructs and the corresponding Adapted Grammar term.

Activity Diagram construct	Adapted Grammar term
	$a.\iota \mapsto \mathcal{N}$
	$l : \otimes$
	$l : \odot$
	$l : Merge(\mathcal{N})$
	$l : Join(\mathcal{N})$
	$l : Fork(\mathcal{N}, \mathcal{N})$
	$l : Decision(\mathcal{N}, \mathcal{N})$
	$l : \mathcal{B} \mapsto \mathcal{N}$
	a
	$!a$
	$?a$
	$\uparrow a.\mathcal{A}$

Table 3.1: Relation between Activity Diagram constructs and Adapted Grammar terms.

We define a translation function T that takes an adapted grammar term and translates it to the TORXAKIS modelling language. In these translation steps, the function T is used recursively. The translations are based on the derivation rules from table 2.3 that encapsulates the semantics of a SysML activity diagram.

<i>Term</i>	TORXAKIS <i>Syntax</i> ($T(\text{Term})$)	<i>Description</i>
$a.l \mapsto \mathcal{N}$	PROCDEF a [In , Out :: Transition]() EXIT $\hookrightarrow ::= T(\mathcal{N})$ ENDDDEF	For each initial state a new process is created with its name a . This process is called at the <i>call behaviour</i> term or in the model definition if this is the main initial state.
$l : \otimes$	EXIT	The EXIT keyword of TORXAKIS stops the current flow, just as rule FFin in table 2.3 describes.
$l : \odot$	EXIT	Just like rule FFin , the EXIT keyword of TORXAKIS stops all flows in the current diagram, just as rule AFin in table 2.3 describes.
$l : M(\mathcal{N})$	l [In , Out]() ; PROCDEF l [In , Out :: \hookrightarrow Transition]() EXIT ::= $T(\mathcal{N})$ ENDDDEF ; l [\hookrightarrow In , Out]()	For each merge term (M) a new process is created. This process can be referenced when an incoming edge of the merge reaches the node.
$l : J(\mathcal{N})$	EXIT ; ...>>> $T(\mathcal{N})$	When a <i>Join</i> node is reached, stop the current flow. Where the corresponding fork is defined i.e. the fork that divided the flow which is getting merged by this join, continue with the enable operator (>>>). This operator ensures that all incoming edges of the join are finished before continuing with \mathcal{N} as is described in rules JON-1 and JON-2 from table 2.3.
$l : F(\mathcal{N}_1, \mathcal{N}_2)$	$((T(\mathcal{N}_1)) (T(\mathcal{N}_2)))$	The TORXAKIS parallel operator $ $ allows for the terms on the left-hand side and right-hand side to execute in parallel, just as rule FRK-1 in 2.3 for a fork (F) describes.
$l : D(\mathcal{N}_1, \mathcal{N}_2)$	$((T(\mathcal{N}_1)) ## (T(\mathcal{N}_2)))$	The TORXAKIS choice operator ## allows for the execution of either the term on the left-hand side or the term on the right-hand side, just as rules DEC-1 and DEC-2 in 2.3 for a decision (D) describe.
$l : \mathcal{B} \mapsto \mathcal{N}$	$T(\mathcal{B}) \>\> T(\mathcal{N})$ or $T(\mathcal{B}) \>>> T(\mathcal{N})$ if \mathcal{B} is a function call	The $\>\>$ operator chains the actions, described in rules ACT-1 and ACT-2 . The enable operator (>>>) is required for a call behaviour term, because the enable operator continues with \mathcal{N} only when \mathcal{B} finishes. This behaviour is described in the rule BEH-2 from table 2.3.
l	l [In , Out]()	A reached label indicates that it has already been encountered en translated before in the algorithm. Therefore, we can call the process with name l .

a	In ! a or Out ! a	Depending on action a , this is either an input or output.
$!a$	Out ! a	A send action is an output. The TORXAKIS behaviour corresponds with the SND rule from table 2.3.
$?a$	In ! a	A receive action is an input. The TORXAKIS behaviour corresponds with the REC rule from table 2.3.
$\uparrow a.\mathcal{A}$	a [In, Out] () ; $T(\mathcal{A})$	When a call behaviour with name a is reached, it recursively is translated using the function T if this has not been done before. Then this process a is called according to rule BEH-0 from table 2.3.

Table 3.2: Translation of the Adapted Grammar terms in TORXAKIS.

As an example, we will apply the algorithm on the coffee machine activity diagram displayed in figure 2.19.

First, we need to adjust the activity diagram such that our adapted grammar can express it. Our grammar does not support the notion of *groups*. These are the boxes in the diagram labelled with *Boiler*, *Mixer* and *Grinder*. We can omit these groups since their sole purpose is to make the diagram more structured and have no semantic meaning. We also notice the *Action Pins* *Pin1*, *Pin2*, *Pin3* and *Pin4* in the diagram. An Action pin denotes the transfer of data from a source action to a target action. For the behaviour of this activity diagram, this data is not relevant and does not influence the behaviour and semantics. In general, the data that flows between action pins does not need to be translated to a TORXAKIS model, because even when edges have guards, we allow all transitions in the TORXAKIS model. The output of the SUT communicates the taken path to TORXAKIS. Therefore, we can interpret an input pin, an edge and an output pin as a normal edge.

After the preprocessing steps, we can continue. We need to rewrite the activity diagram to the adapted grammar. We do this by starting at the initial node. Then we traverse the diagram. When the flow splits, we create new terms to which can be referred. This process needs to be done manually at the moment, but chapter 5 describes how this algorithm can be implemented and used.

The coffee machine activity diagram from figure 2.19 can be expressed by the adapted grammar term $\mathcal{A}_{\text{coffee}}$ where:

$$\begin{aligned} \mathcal{A}_{\text{coffee}} &= \text{Make Coffee.}\iota \rightsquigarrow l_0 : F(\mathcal{N}_1, \mathcal{N}_2) \\ \mathcal{N}_1 &= l_1 : \text{Boil the water} \rightsquigarrow l_2 : \text{Send water boiled} \rightsquigarrow l_3 : \text{Receive water boiled} \rightsquigarrow l_4 : \text{Join}(\mathcal{N}_3) \\ \mathcal{N}_2 &= l_5 : \text{Grind coffee beans} \rightsquigarrow l_6 : \text{Send beans grinded} \rightsquigarrow l_7 : \text{Receive beans grinded} \rightsquigarrow l_4 \\ \mathcal{N}_3 &= l_8 : \text{Mix} \rightsquigarrow l_9 : \text{Pour in a cup} \rightsquigarrow l_{10} : \odot \end{aligned}$$

Listing 3.4 shows the translation of the term $\mathcal{A}_{\text{coffee}}$ to the TORXAKIS modelling language using the translation function T .

```

1 PROCDEF Make_Coffee[In, Out :: Transition]() EXIT
2   ::=
3     (11[In, Out]() ||| 15[In, Out]())
4     >>> 18[In, Out]()
5 ENDDDEF
6
7 PROCDEF 11[In, Out :: Transition]() EXIT
8   ::=
9     In ! Boil_the_water
10    >-> In ! Send_water_boiled
11    >-> In ! Receive_water_boiled
12    >-> EXIT
13 ENDDDEF
14
15 PROCDEF 15[In, Out :: Transition]() EXIT
16   ::=
17     In ! Grind_coffee_beans
18    >-> In ! Send_beans_grinded
19    >-> In ! Receive_beans_grinded
20    >-> EXIT
21 ENDDDEF

```

```

22
23 PROCDEF 18[In, Out :: Transition]() EXIT
24 ::=
25     In ! Mix
26     >-> In ! Pour_in_a_cup
27     >-> EXIT
28 ENDDDEF

```

Listing 3.4: Translation of the term $\mathcal{A}_{\text{coffee}}$ to the TORXAKIS modelling language using the translation function T

We notice the following things from the result of the translation.

1. The `Transition` type is not defined. As we discussed in 3.3, for type safety we will need to define this type that includes all possible actions. This type definition can be found in listing 3.5.
2. An instance (or constructor) of a type cannot contain spaces in TORXAKIS. Since each action will be a constructor of the `Transition` type, we replace all spaces by underscores.
3. As can be seen on line 3, sub-terms can be refactored in new processes. The reason for doing this is code legibility.
4. To make the model work, we need to add a model definition, channel definition and connection definition. These definitions can be found in listing 3.5.
5. From the model it can be noted that all actions are marked as inputs. Translating each action in a SysML activity diagram is the only part of the algorithm that may need manual guidance. Depending on the testing requirements, an action can be handled differently. Since the activities in the coffee machine activity diagram do not use send signal actions or receive signal action, we cannot determine with certainty whether an action is an input or output.

In the example of the coffee machine, one might say that the *Send water boiled* and *Send beans grinded* actions are outputs, because the *Boiler* and *Grinder* send the boiled water and beans as output respectively. If the components of the coffee machine (*Boiler*, *Grinder* and *Mixer*) are considered as stand-alone units for MBT, then the inputs and outputs would make sense. However, since MBT is a *black-box* testing technique, the internal steps of the machine are not considered and abstracted away with a τ transition. When the coffee machine is considered as a whole, it might be the case that the cup of coffee is the only observable output, which would imply that the TORXAKIS model would only consist of one output. This would result in a trivial TORXAKIS model that only outputs a *Pour in cup* output when the coffee is poured in a cup after the activity diagram is started (which is the only input).

```

1 TYPEDEF Transition
2 ::=
3     Boil_the_water
4     | Send_water_boiled
5     | Receive_water_boiled
6     | Grind_coffee_beans
7     | Send_beans_grinded

```

```

8     | Receive_beans_grinded
9     | Mix
10    | Pour_in_a_cup
11 ENDDDEF
12
13 MODELDEF Model
14 ::=
15     CHAN IN   Input
16     CHAN OUT  Output
17
18     BEHAVIOUR
19         Make_Coffee[Input, Output]()
20 ENDDDEF
21
22 CHANDEF Channels
23 ::=
24     Input  :: Transition;
25     Output :: Transition
26 ENDDDEF
27
28 CNECTDEF Sut
29 ::=
30     CLIENTSOCK
31
32     CHAN OUT Input  HOST "localhost" PORT 7890
33     ENCODE   Input  ? s  -> ! toString(s)
34
35     CHAN IN  Output HOST "localhost" PORT 7890
36     DECODE   Output ! fromString(s) <- ? s
37 ENDDDEF

```

Listing 3.5: Additional TORXAKIS code required to make the model from listing 3.4 run.

3.5 Reductions

The translation algorithm produces a TORXAKIS model that adheres to the behaviour of the activity diagram according to the formalized derivation rules from [18] and [23], because these rules were used to define the translation of the terms in the adapted grammar to TORXAKIS terms. However, the model can be more compacted by applying one of the reductions listed below. Each reduction does not alter the behaviour of the resulting model.

- Superfluous parentheses can be removed. This is the case when there are multiple parentheses, $((A)) = (A)$, or when there are parentheses surrounding a single action.
- Statements including `EXIT` can be reduced if they contain a reducible sub-term. The terms $A \parallel \text{EXIT}$ or $\text{EXIT} \parallel A$ can be rewritten to A , because executing an `EXIT` statement in parallel stops immediately. This is visualized in figure 3.4. These terms can be rewritten to A . The terms $A \gg \text{EXIT}$ and $A \>\text{EXIT}$ can be replaced by A if and only if the `EXIT` statement is the final statement of the process.
- `PROCDEFs` that only contain an `EXIT` statement can be removed. In all places where this process is referenced, replace the reference by `EXIT` and apply the reduction from the previous point. Construct $A \>\text{EXIT}$ can be replaced by A .

- The term $B \#\# (A \#\# B)$ can be reduced to $A \#\# B$, because two consecutive decisions that lead to the same term does not differ from having only one such decision. This is visualized in figure 3.5.

Let say we have just translated an activity diagram in model A . Let us call the function that applies the reductions listed above R . Applying function R n times is noted as R_n . We continue to apply R on A until $R_{n+1}(A) = R_n(A)$; i.e. a fixpoint has been reached and therefore there are no further reductions possible. It is possible that after one application of R , new constructs form that can be further reduced. This process terminates since each reduction strictly reduces the size of the term. This process is also confluent i.e. the order of applying R on a term, because each reduction does not affect the general terms (like A or B). This means that if a reduction could be done on A , but also on a term with A as subterm, the former reduction can still be applied if first the reduction is applied where A is the subterm. Together, this means that each resulting TORXAKIS model can be reduced to its unique normal form.

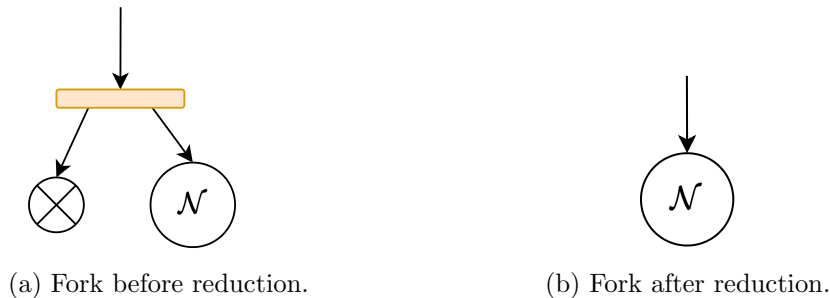


Figure 3.4: A fork can be reduced when one of its outgoing edges point to a FlowFinal node.

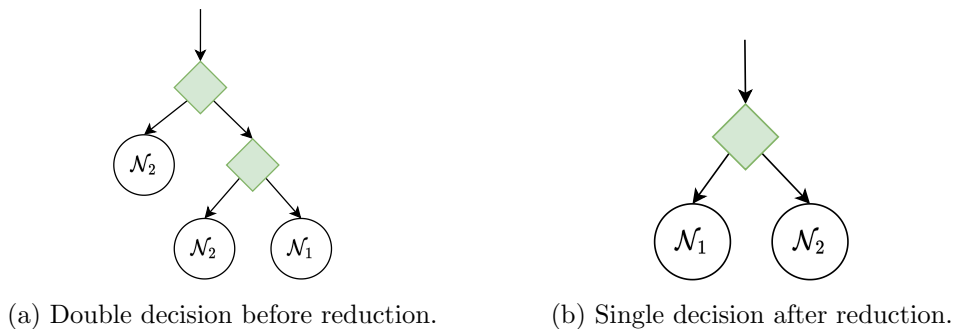


Figure 3.5: A double decision that points twice to the same term can be reduced to one decision.

3.6 Discussion

In the previous section we have defined a formalization for the SysML activity diagram and an algorithm that translates the formal language to the TORXAKIS modelling language. In theory, this algorithm can be applied on any activity diagram. In practice however, there are three points that need to be considered when applying this algorithm.

- All *behavioural* activity diagram constructs are supported by the algorithm. However, constructs like groups and action pins, which we do not consider, are available in activity diagrams. The activity diagram needs to be preprocessed by omitting these constructs. These operations do not influence the semantics of the activity diagram. The SysML timer construct (figure 3.6) is also not supported by the algorithm, because TORXAKIS does not support the notion of time. There is however a workaround for this limitation. We can set the `param_Sim_deltaTime` and `param_Sut_deltaTime` of TORXAKIS to the highest timer value we found during the algorithm. This way, TORXAKIS waits for the next action as long as is specified.
- The most important question a user of this algorithm should ask themselves is *What does this activity diagram represent?* In MBT, the software system is considered a black box. The SUT can be stimulated by providing inputs. Then the outputs are validated against the model. SysML activity diagrams on the other hand are usually used for modelling the *internal* behaviour of the SUT, so the number of inputs and observable outputs might be low. Therefore, it has to be determined carefully what exactly the inputs and outputs are the tester wants to include in the MBT model. If an action is determined to be internal, the τ operation or `ISTEP` can be used.
- The SysML activity diagram does not support regular actions i.e. not considering send signal actions or receive signal actions, to be labelled as inputs or outputs. Since this concept is fundamental for MBT, it is important that the translation algorithm is aided by someone with domain knowledge about the translated model to indicate which activities can be considered inputs, outputs or internal steps.



Figure 3.6: SysML Timer

Chapter 4

Case Study on ADAPTO

In this chapter, we will perform a case study on applying Model-Based Testing (MBT) on a system which is modelled with SysML models. This system will be ADAPTO. We attempt to apply Model-Based Testing on ADAPTO in three different ways. We first attempt to apply the found structure of Gherkin tests in section 2.4.1 on the existing Gherkin tests of ADAPTO, because we want to test if the existing Gherkin tests would provide sufficient information to be a viable option for using with MBT. We will apply the enhanced translation procedure from section 3.3 on the SysML activity diagrams of ADAPTO, because we want to learn how the existing translation procedure works on the activity diagrams of ADAPTO and to see if our proposed enhancement provides any benefit. Finally, we will apply our translation algorithm from section 3.4 on the SysML activity diagrams of ADAPTO to see if the formalization and translation is applicable on the activity diagrams of ADAPTO. For each of these methods, we discuss their usability and limitations.

Note: in the public version of this thesis, sensitive information/diagrams covered under a signed NDA are replaced by public information/diagrams.

4.1 System Under Test

Let us first define the system under test (SUT). The SUT for this case study is Vanderlande's ADAPTO storage system, which is explained in section 2.4.

With MBT, we are interested in the behaviour of the SUT. Hence, we will consider the behaviour diagrams from figure 2.23. As can be seen the only behavioural diagrams used in our SUT are *Use Case Diagrams* (UCDs) and *Activity Diagrams*. The UCDs that are modelled for ADAPTO provide an overview of how ADAPTO can be used. This is not thorough and granular enough for our testing (see section 4.2.2). Furthermore, the UCDs do not cover all parts of the system that Vanderlande is interested in testing. Therefore, we decided to focus our research on activity diagrams.

We will use the same activity diagrams for each of the three methods we will apply on the SUT. We decided to test the core of the ADAPTO system: storing an incoming storage tray (TSU). This flow covers the hardware components shuttles and lifts, the communications between the ADAPTO components (communicating to what level and location the TSU needs

to be moved and instructing the lift and shuttle to execute this) and error handling. The considered activity diagram can be found in figure 2.19. These also include the nested activity diagrams that are referenced from other activity diagrams.

4.2 Adapter

To use our SUT for model based testing in TORXAKIS, we need to use an adapter. An adapter functions as the layer in between the SUT and the TORXAKIS model. Essentially TORXAKIS sends messages to the adapter over the network which get interpreted by the adapter. Then the adapter performs the corresponding action on the SUT. This communication is bidirectional. Figure 2.7 shows this communication in a diagram.

4.2.1 Step Definitions

The step definitions of ADAPTO are considered the atomic actions on ADAPTO when testing the system, because these are the instructions that can be given to ADAPTO by a user controlling ADAPTO. The definitions are grouped by each component of ADAPTO in order to make it clearer to which component the WCF needs to send which instruction. These definitions are an abstraction of underlying sequences of actions that together form a commonly used action. Examples are show in figure 4.3.1. The step definitions of ADAPTO have been developed with automatic testing in mind, which is mentioned in section 2.4.1.

The step definitions include sending instructions to the ADAPTO components and retrieving information about the state of the components. Therefore, they are suitable to be used with TORXAKIS. The step definitions containing instructions for ADAPTO can be matched with TORXAKIS inputs and the step definitions containing state information can be matched with the outputs.

4.2.2 Granularity

When applying model based testing on a system, a decision needs to be made on which granularity of the steps in the system should be used. The granularity is what steps on the SUT are considered atomic. When choosing a low level granularity, for example the bits that make up a message send between ADAPTO components, the model becomes too complex; you lose overview. On the other hand, when choosing a high level granularity, for example only considering if the system is turned on or off, a lot of valuable information might be lost.

Picking a suitable granularity often lies in the middle of these extremes. In our case, we chose to consider the step definitions atomic, because of the following three reasons:

- Step definitions are the instructions that can be given to ADAPTO by a user controlling ADAPTO.
- The step definitions are currently used by the automatic Gherkin tests for ADAPTO and the step definitions are deemed to provide sufficient insight in these Gherkin tests while not being too low level.
- For ADAPTO, the step definitions are already developed and can therefore be used trivially from an adapter.

4.3 First Attempts

To determine the feasibility of using the SysML models created for ADAPTO for MBT, we started with a top-down approach. In our situation this means that we first investigated how SysML models would be translated for MBT. We tried to formalize the SysML activity diagrams without truly understanding how we would like to test our SUT. Therefore, we switched to a bottom-up approach, which means that we first make our goal of the translation clear and then researching how to achieve that goal.

For our bottom-up approach, the first goal was to set up the foundation: being able to communicate from TORXAKIS via an adapter to our SUT. We did this by creating a trivial TORXAKIS model which could turn the SUT on and off. The model can be found in listing 4.1.

```

1 STAUTDEF
2   STATE
3     off , on
4   INIT
5     off
6   TRANS
7     off -> In ! "turnOn"      -> on
8     on  -> Out ! "isTurnedOn" -> on
9     on  -> In ! "turnOff"     -> off
10 ENDDF

```

Listing 4.1: State Automaton Definition that can turn on and off a SUT

Even though this is a very basic model, we can already learn a lot about our SUT. In this model, the messages ‘turnOn’, ‘isTurnedOn’ and ‘turnOff’ are sent to the adapter as plain strings. The adapter interprets these messages and calls the corresponding step definition. Later, as we create more complex models, we use a type safe message system, which is mentioned in section 2.3.3, for the communication.

After the ‘turnOn’ message is sent by TORXAKIS, it can do two possible steps. Either TORXAKIS expects a ‘isTurnedOn’ message *or* it sends a ‘turnOff’ message. In the latter case the SUT should turn off after it has been turned on and the test would succeed. In the other case however, the test fails. This is because TORXAKIS expects the message ‘isTurnedOn’ for which it waits at most 1 second. Since ADAPTO takes approximately 45 seconds to boot on our system, the test fails. To solve this issue, we can adjust some TORXAKIS parameters to allow for a waiting time of one minute. This is done by performing the commands in listing 4.3.

```

1 param param_Sim_deltaTime 60000
2 param param_Sut_deltaTime 60000

```

Using the step definitions, we can retrieve the current state of the SUT in order to check whether it is turned on. This check busy waits until the SUT is in the checked state. The busy waiting stops after a set timeout. If the checked state is not (yet) reached by that time, the assert in the step definition will fail and therefore the adapter will stop running. This also implies that the TORXAKIS test fails, because either the timeout has been reached or the connection between the adapter and TORXAKIS is lost.

Now that we have a basic TORXAKIS model working, we will construct a more interesting model. We will do this by using one of the automatic Gherkin tests as a basis.

4.3.1 Gherkin Tests

The relation between Gherkin tests and TORXAKIS is discussed in section 2.4.1. Now we will create a TORXAKIS model from a Gherkin test.

For this manual translation, we have chosen a Gherkin test that will retrieve a TSU from storage and deposits it in a module outbound -a conveyor belt just outside the ADAPTO system which can be accessed-. This test can be found in listing 4.2.

```

1 Scenario: Make coffee
2   Given Coffee machine is started
3   And cup is placed under the faucet
4   When pressing the start button
5   Then water is boiled
6   And beans are grinded
7   And water and beans are mixed
8   And coffee is poured in the cup

```

Listing 4.2: Make coffee Gherkin test

A Gherkin test is used for automatic testing. Since the goal of a Gherkin test is to create a test in a human-readable way, the steps in such a test need to be translated for a computer to understand. This process is done with regular expressions that match the sentences. This way the arguments are also retrieved. Since the Gherkin test are created in terms of the step definitions, each line in a Gherkin test corresponds with a step definition. We will use this fact too in our TORXAKIS translation.

We can translate each of the lines in the given Gherkin test to TORXAKIS using the relation we found in section 2.4.1. To connect the translations, we assume that clauses are evaluated consecutively instead of allowing parallel evaluations. This is also in line with the way that Gherkin tests for ADAPTO are created. Since we will be translating a linear flow to TORXAKIS, we can use the linear translation procedure from section 2.3.3 in combination with our proposed enhancement in section 3.3 for defining the structure of the resulting TORXAKIS model. This includes defining a **Transition** type for all transitions. When adding a constructor to a type definition (**TYPEDEF**), TORXAKIS generates the following functions. Given the constructor `x { y :: Int }`, the function `isx` is created to determine if an object is of type `x`. The function `y` is also generated, which takes as input an object and returns the integer value associated with the argument of the object. Naturally, this object has to be of type `x`. Since these functions are created, TORXAKIS does not allow for constructors or arguments to have the same identifier, because there will be more than one function with the same name. Since some step definitions have the same argument labels as other step definitions and according to the translating procedure from section 2.3.3, we have to create constructors in our defined **Transition** type, we need to make sure that no duplicate constructor names or argument names occur by adding a postfix to the identifiers that do. The result of the translation can be found in listing 4.3.

```

1 TYPEDEF Transition
2   ::=
3     Start_coffee_machine
4     | Place_cup_under_faucet
5     | Press_start_button
6     | Boil_water
7     | Grind_beans
8     | Mix_water_and_beans
9     | Pour_in_a_cup

```

```

10 | Stop_coffee_machine
11 ENDEF
12
13 STAUTDEF Make_coffee [Inp, Outp :: Transition]()
14 ::=
15   STATE s0, s1, s2, s3, s4, s5, s6, s7
16   INIT s0
17   TRANS
18     s0 -> Inp ! Start_coffee_machine -> s1
19     s1 -> Inp ! Place_cup_under_faucet -> s2
20     s2 -> Inp ! Press_start_button -> s3
21     s3 -> Inp ! Boil_water -> s4
22     s4 -> Inp ! Grind_beans -> s5
23     s5 -> Inp ! Mix_water_and_beans -> s6
24     s6 -> Outp ! Pour_in_a_cup -> s7
25     s7 -> Inp ! Stop_coffee_machine -> s0
26 ENDEF

```

Listing 4.3: Make coffee model

The state machine of this TORXAKIS STAUTDEF can be seen in figure 4.1. In this model, the transitions are labelled according to the Gherkin test clauses. For each trigger, the word ‘Trigger’ is appended to the end of the label. The state labels are made up labels for the states in between transitions. When testing this model on our SUT, we find that TORXAKIS does not receive the outputs from the adapter. This is because our SUT does not send messages about the state of itself automatically after turning the system on and creating a retrieve order. We solved this problem by adding a trigger for each output. When we expect and output, we first send a trigger to our adapter. The adapter checks if the SUT is in the requested state and writes the corresponding message to the output.

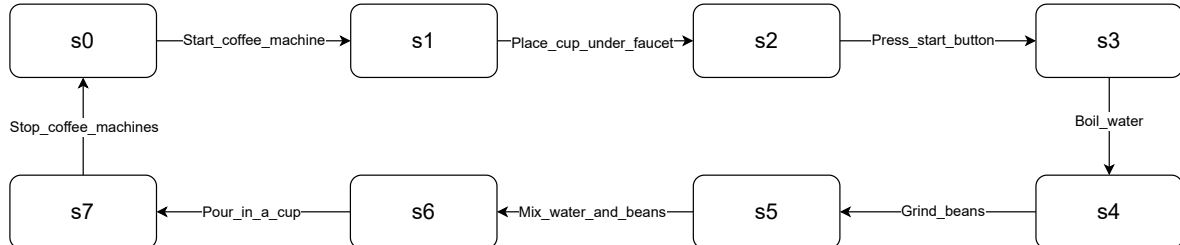


Figure 4.1: State machine for the make coffee Gherkin test

We figured out that a Gherkin test is essentially one path from an initial node to a final node in the underlying SysML model. A SysML activity diagram can therefore be associated with multiple Gherkin tests. Since it is more efficient to translate one activity diagram than multiple Gherkin tests and because the translated TORXAKIS test can fail, we will now move on to translating the SysML activity diagrams to TORXAKIS.

4.3.2 Applying Translation Procedure

Before we apply our translation algorithm from section 3.4 on the activity diagrams of ADAPTO, we will first apply the enhanced translation procedure from section 3.3. The sole purpose of using this translation procedure is to gain knowledge about how Vanderlande has translated activity diagrams before. Because we know that this translation procedure is not formal, we know that we cannot use this procedure reliably on all activity diagrams.

Therefore, we limit ourselves to applying this translation procedure on a subset of the activity diagrams to understand the workings of the procedure. We will apply this procedure on the activity diagram in figure 2.19

To translate this activity diagram, we will use the non-linear translation procedure described in section 2.3.3, because our diagrams contain non-linear nodes like forks and joins. In contrary to what we mentioned in section 4.2.2, we adapt the granularity of this translation procedure such that each action in the activity diagram corresponds to one step in TORXAKIS. This is done to match the original translation procedure and to therefore understand how this procedure works. Since the goal of applying this translation procedure is to show how control flow would be translated to TORXAKIS, we decided to use the `String` type for the communications.

In section 4.2.2 we determined that step definitions are the atomic actions that need to be considered for testing ADAPTO. We need to find a way to apply this granularity while translating the activity diagrams. In the next section, we will show how we achieve this by preprocessing the activity diagrams. After the preprocessing, we will apply the translation algorithm from section 3.4 on the preprocessed activity diagrams. This translation algorithm is formal, while the translation procedure we used in this section is not.

4.4 Preprocessing Activity Diagrams

To ensure that the translated TORXAKIS model uses the granularity specified in section 4.2.2, we preprocess the activity diagrams to simplify the application of the translation algorithm afterwards. We use the following preprocessing steps:

4.4.1 Preprocessing Steps

Since we will consider the ADAPTO step definitions as atomic steps, we need to solve the following two problems.

1. Determine if an activity in a SysML activity diagram corresponds to a step definition from ADAPTO.
2. Find a procedure that removes an action that does not correspond to a step definition from the activity diagram while not influencing the behaviour of the activity diagram.

There is no correlation between the activity diagrams and the step definitions according to Vanderlande. Therefore, we need to find a way to determine for a given activity whether it corresponds to a step definition. The following steps are taken to solve issues 1 and 2 listed above.

1. Create a list of all the names of the step definitions of ADAPTO. This can be done manually by going through the source code and copying each method name *or* for example by using a regular expression on the source code to extract all the names of the step definitions.
2. While doing the procedure from section 2.3.3, when an activity is reached, first check if this activity is a step definition. The activity name is a sentence in English, so any

method that can check whether this sentence matches a step definition would probably need a manual check. Now there is a match or not.

- In case of a **match**, continue the procedure normally.
- In case of **no match**, we have two options.
 - We can translate the activity to an internal (τ) step in TORXAKIS.
 - We can remove the activity from the diagram. We do this by performing a **delete** operation. This operation first takes the input flow arrow set I and the output flow arrow set O . Each set contains edges stating the source activity and the target activity as a tuple. These two sets and the activity itself are removed from the activity diagram. The edges in the set $\{(s_I, t_O) | (s_I, t_I) \in I, (s_O, t_O) \in O\}$ are added to the activity diagram. Duplicate edges i.e. edges with the same source and target are removed.

The first option is easier to implement, but the second option reduces the state space of the resulting TORXAKIS model. This state space reduction can be significant if a lot of activities need to be deleted, as is the case for the ADAPTO SysML models.

Figure 4.2 shows the effect of changing an action in an internal action and how the *delete* operation affects an action.

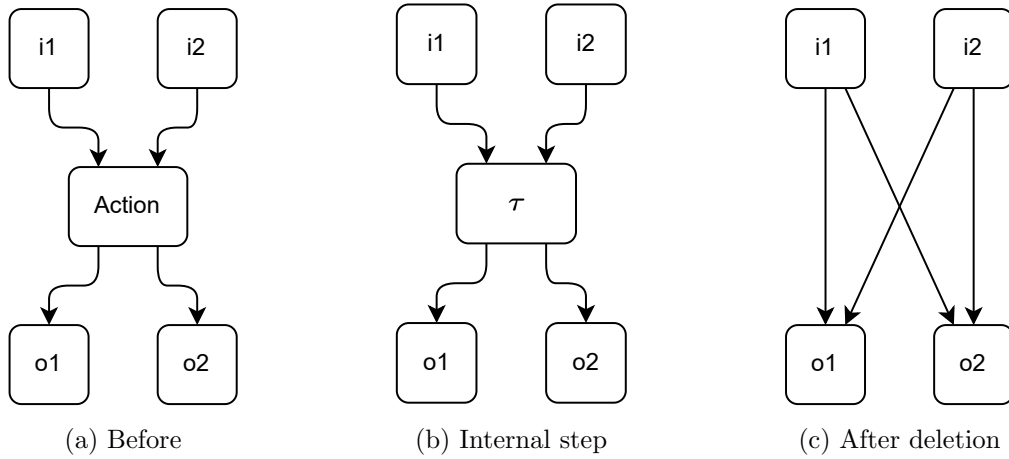


Figure 4.2: The effect of changing an action in an internal action or applying the *delete* operation

We want to ensure that applying the delete operation does not affect the test verdicts of an MBT tool. Therefore, we will argue the following properties of the delete operation.

- Let us call the model that results from the translation algorithm of the activity diagram before the delete operation s_{before} and the model that results from the translation algorithm of the activity diagram after the delete operation s_{after} . We have $traces(s_{before}) = traces(s_{after})$, because a trace is a τ -abstracted sequence of observable actions. This means that unobservable τ actions are not included in a trace. These actions are exactly the actions we remove using the delete operation, because only the internal actions that

do not correspond with an ADAPTO step definition is deleted, hence $traces(s_{before}) = traces(s_{after})$.

We cannot conclude properties about the delete operation regarding preserving Straces or *ioco*, because we cannot say in what cases we can observe quiescence in the translated activity diagram, because the semantics regarding the behaviour of choosing a next action in SysML activity diagrams is not formally defined.

Figure 4.3 shows an example where applying the *delete* operation may result in different suspension traces. In figure 4.3a, after the τ action, we can only perform input *c*, hence the system is quiescent. In figure 4.3b, we can choose between output *b* or input *c*, hence the system is not quiescent. Therefore, the out-sets are not equivalent and we cannot show *ioco*. Since choosing the next action in an activity diagram is not formally defined, we cannot say if after action *a* in figure 4.3b which action is performed. Since the next step in this state is not formally defined, we conclude that we cannot conclude whether the *delete* operation preserves suspension traces or *ioco*.

- The model before the delete operation is *not trace equivalent* to the model after the delete operation. This is because the LTS equivalent to the activity diagram before applying the delete operation has a strace that includes the to be the deleted action. Since this action will be deleted, the LTS equivalent to the activity diagram after applying the delete operation does not have a trace that includes the deleted action.

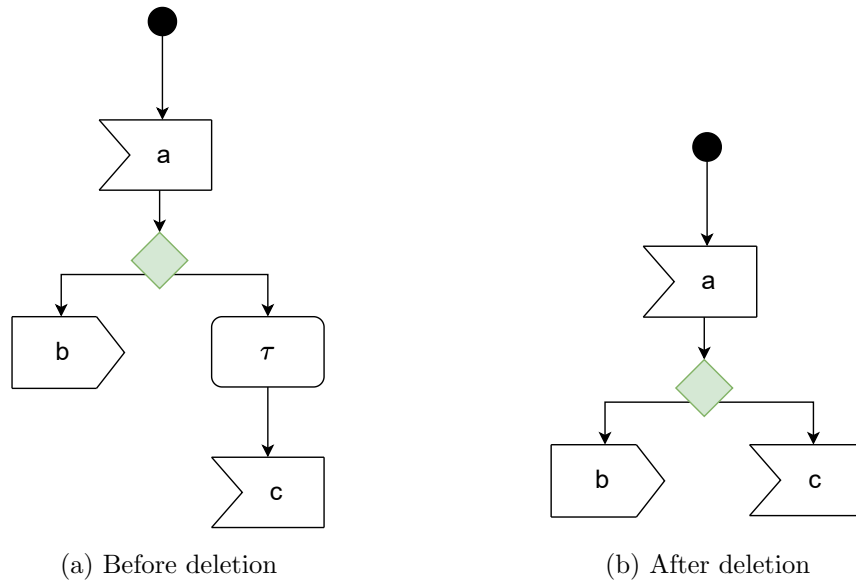


Figure 4.3: When applying the *delete* operation, the suspension traces may change, depending on the semantics of SysML activity diagrams.

4.5 Application of the Translation Algorithm

In this section we will apply the adjustments from section 4.4 on the activity diagram from figure 2.19. After this, we will apply the translation algorithm from section 3.4 on the pre-processed activity diagrams. We will use the resulting TORXAKIS model for MBT on the SUT.

4.5.1 Preprocessing

As we determined in section 4.4, there are two possible options to reduce the activity diagrams to only use step definitions as actions. We figured that the majority (> 80%) of the actions in the activity diagrams of ADAPTO are not step definitions. This is because in the ADAPTO activity diagrams, internal steps as well as the communication between ADAPTO components is modelled, while these are not observable from the outside. In our case study, we also came across activity diagrams that do not contain actions that correspond to a step definition. These activity diagrams were either not completed or they model only internal actions. In this case, the *delete* operation removes these diagrams completely and where these models are referenced in call behaviours, can be deleted as well. Since a large part of the actions in activity diagrams do not correspond to a step definition, converting the actions in the activity diagrams that do not correspond to a step definition into an internal action would make the preprocessed model unnecessarily large while being unclear to read because of all the internal steps scattered throughout the activity diagram. Therefore, we chose to use the *delete* operation. When applying the first step of the translation algorithm, which is encoding the activity diagrams in the Adapted Grammar, we simultaneously performed the delete operation on the actions that do not correspond with a step definition. We determined that an action needed to be deleted by manually checking if a step definition existed for this action. We could not reliably do this process automatically, because the actions are described in natural language, while the step definitions are functions in the source code. In the next section, we show the results of applying the translation algorithm as well as the results of testing using the resulting model.

4.6 Results

In this section, we first show the results of applying the translation algorithm (section 3.4) on the preprocessed activity diagram from figure 2.19. After this, we use the resulting model for MBT and show the results.

4.6.1 Encoding Activity Diagrams

Figure 4.4 shows the result of encoding the activity diagram from figure 2.19 which describe the behaviour when storing a TSU into the Adapted Grammar while deleting actions that do not correspond to a step definition. This activity diagram can be expressed by the adapted grammar term \mathcal{A}_{coffee} where:

$$\begin{aligned} \mathcal{A}_{coffee} &= \iota \mapsto l_1 : \text{Fork}(\mathcal{N}_1, \mathcal{N}_2) \\ \mathcal{N}_1 &= l_2 : \text{Boil the water} \mapsto l_3 : \text{Send water boiled} \mapsto l_4 : \text{Receive water boiled} \mapsto l_5 : 2.\text{Join}(l_6) \\ \mathcal{N}_2 &= l_7 : \text{Grind coffee beans} \mapsto l_8 : \text{Send beans grinded} \mapsto l_9 : \text{Receive beans grinded} \mapsto l_{10} : 2.\text{Join}(l_6) \\ \mathcal{N}_3 &= l_6 : \text{Mix} \mapsto l_{11} : \text{Pour in a cup} \mapsto l_{12} : \odot \end{aligned}$$

Figure 4.4: Adapted Grammar term of the preprocessed coffee machine activity diagram.

\perp indicates that a term is undefined. This happens when in the provided activity diagram a nested activity diagram is referenced through a call behaviour, but is not (yet) defined in

the SysML model. This happens, because the SysML models are not finished and still being worked on. A workaround for this is to replace each \perp with the empty term \otimes .

4.6.2 Encoding Activity Diagrams

Listing 4.4 shows the TORXAKIS model when translating the adapted grammar term from figure 4.4 to the TORXAKIS modelling language. Note that this listing shows the translation after applying the reductions from section 3.5.

```

1 PROCDEF Make_Coffee[In, Out :: Transition]() EXIT
2   ::=
3     (parallel_1[In, Out]() ||| parallel_2[In, Out]())
4     >>> continuation[In, Out]()
5 ENDDDEF
6
7 PROCDEF parallel_1[In, Out :: Transition]() EXIT
8   ::=
9     In ! "Boil the water"
10    >-> In ! "Send water boiled"
11    >-> In ! "Receive water boiled"
12    >-> EXIT
13 ENDDDEF
14
15 PROCDEF parallel_2[In, Out :: Transition]() EXIT
16   ::=
17     In ! "Grind coffee beans"
18    >-> In ! "Send beans grinded"
19    >-> In ! "Receive beans grinded"
20    >-> EXIT
21 ENDDDEF
22
23 PROCDEF continuation[In, Out :: Transition]() EXIT
24   ::=
25     In ! "Mix"
26    >-> In ! "Pour in a cup"
27    >-> EXIT
28 ENDDDEF
29
30 CHANDEF Channels ::= In :: String
31                   ; Out :: String
32 ENDDDEF
33
34 MODELDEF Model
35   ::=
36     CHAN IN In
37     CHAN OUT Out
38
39     BEHAVIOUR Make_Coffee [In,Out] ( )
40 ENDDDEF
41
42 CNECTDEF Sut
43   ::=
44     CLIENTSOCK
45
46     CHAN OUT In HOST "localhost" PORT 7890
47     ENCODE In ? qop -> ! qop
48

```

```

49      CHAN IN    Out      HOST "localhost" PORT 7890
50      DECODE    Out ! s   <-    ? s
51 ENDDDEF

```

Listing 4.4: Resulting TORXAKIS model translating the adapted grammar term from figure 4.4.

We observed the following in the resulting translation:

- First, we noticed that all actions in the resulting model are outputs. That is because the activity diagrams model the behaviour of ADAPTO after a store order is created. This order is the input of the system. Furthermore, the preconditions in the Gherkin tests of ADAPTO, i.e. the **when** clauses, are implicitly assumed in the activity diagrams, but are not listed in the activity diagram. Therefore, we have to make sure that the SUT is configured according to the assumptions of the SysML model to make applying MBT make sense. In the case of ADAPTO, this is starting the system using a level captive topology. This means that shuttles are not allowed to move from one level to another, the number of shuttles is equal or greater than the number of levels and that the shuttles are divided over the levels equally.
- As we expected, the actions that ended up in the model (the step definitions) closely resemble the Gherkin tests that tests the store order functionality. In essence, the resulting model is the result of merging all Gherkin tests into one, which is what we wanted to achieve, because from this model the Gherkin tests should be derivable, which they are.
- The step definitions that correspond with the actions in the resulting model may have arguments. These arguments contain information about the current order (e.g. location of the pickup point). These arguments cannot be modelled in the current TORXAKIS model. We decided to abstract away over the arguments of a specific order. Instead, when we create a store order (in the adapter for example), we keep track of all the relevant arguments and use them when calling a step definition when the corresponding TORXAKIS action is retrieved in the adapter. This would be relatively simple to add by communicating these arguments from ADAPTO through the adapter to TORXAKIS, and by using these arguments for the transitions in TORXAKIS.

4.6.3 Using the Resulting Model for MBT

We attempted to use the TORXAKIS model from listing 4.4 for MBT on our SUT. We could not properly test the resulting model, because several SysML activity diagrams were incomplete or not yet defined since the modelling process is still a work in progress. However, we attempted to test the SUT based on the resulting model and found three faults in the resulting model which are all caused by faults in the activity diagram used for translation.

1. On line 83, there is an infinite loop since process 172 is unconditionally invoked from itself.
2. The process `Store_TSU_on_level` is invoked on line 55, after which it is called a second time on line 9. Since this process moves the TSU in question, invoking it twice would result in faulty behaviour of the SUT even though this double invoking is modelled in the activity diagram where the resulting model is based on.

3. The state of the order changes in parallel, because in the activity diagram where the resulting model is based on, forks are used, while the state changes in reality happen sequentially on the SUT. This may result in the state change `Order_deposited` (line 39) to occur before the state change `Order_picked_up` (line 36). This is faulty since an order must be picked up first before it is deposited at the intended location.

We asked the modeller of the SysML models of ADAPTO of Vanderlande for a reaction on these findings. She told that at the moment it is not possible to confirm or deny if the activity diagrams are correct, because the diagrams are a work in progress. We suspect that these faults are because the SysML activity diagrams are not modelled after the real implementation of ADAPTO. Therefore, we cannot conclude that there are faults in ADAPTO.

4.7 Conclusion

In this section, we summarize and conclude the findings of our case study: attempting to use MBT for ADAPTO using Gherkin tests and SysML activity diagrams.

We found that determining the granularity of the action in the activity diagrams of ADAPTO was an important first step for our case study, because we needed to have a clear understanding of the goal of testing ADAPTO with MBT. We figured that considering the step definitions as atomic steps for MBT was the best option, because these were already used for the automatic Gherkin tests to test ADAPTO. Since the Gherkin tests use the step definitions, our first translation attempt was to derive a state automaton from a Gherkin for TORXAKIS. We concluded that this was possible, but not really feasible since a model should be generated for each Gherkin test to test the SUT extensively. Since the activity diagrams describe all flows for a particular task in ADAPTO, we shifted to using activity diagrams instead for the translation. First, we applied a translation procedure which informally describes the required steps for a successful translation. Finally, we decided to apply the translation algorithm from section 3.4 on the activity diagrams, because this is a formal and complete solution that has the highest probability of success. We found that the algorithm could be applied automatically, provided its implementation, short of two actions itemized below.

4.7.1 Domain Knowledge

We established during the case study that manual input will still be required when creating models in the TORXAKIS modelling language from the SysML models of the SUT. In the case of our case study, these manual inputs are:

- Determining whether an action in a SysML activity diagram needs to be considered an internal action or an action that needs to be modelled.
- For the actions that need to be modelled, the action needs to be associated with a step definition.

To do these manual operations, the ADAPTO step definitions are required to be known. In addition to this, understanding how ADAPTO works helps with understanding how one should give the manual input. Deciding which granularity of the atomic actions is best in the case of this specific SUT requires understanding of the SUT in question. For each system

which strives to use MBT and already has models describing its behaviour, such as SysML models, the requirements and best solutions might vary. Therefore, we can conclude that domain knowledge about the SysML models is required to be able to derive an MBT model from SysML models. To perform the manual operations, one needs to be familiar with the step definitions and requires therefore domain knowledge. The general structure and the granularity of the atomic operations on the SUT are however the minimum required knowledge. This answers **SQ2**: How much domain knowledge is required to effectively translate SysML models of real world system to a modelling language for MBT?

The translation algorithm that is used in this case study to translate the SysML activity diagrams considered in this case study can be applied to other activity diagrams to then use the translations for MBT. However, manually applying this translation algorithm to each activity diagram is a time-consuming and error-prone task. Therefore, we will provide a guide on implementing this translation algorithm so that it can be used automatically. The manual parts of the translation algorithm cannot be implemented of course, but we think it is still worth it to implement this algorithm for using MBT from SysML activity diagrams, because the manual task of linking actions in activity diagrams to step definitions only has to be done once, after which this linking table can be re-used. We are not sure that such an implementation would provide added academic value, because the implementation will be focused on ADAPTO specifically. Therefore, the implementation is not part of this thesis.

Chapter 5

Translation Algorithm Implementation Guide

In this chapter we will provide a step-by-step approach that explains how the translation algorithm from chapter 3 can be implemented. This algorithm translates a SysML activity diagram to the TORXAKIS modelling language. The implementation of the algorithm is not part of this thesis, because we found that the relation between actions in activity diagrams to TORXAKIS in- and outputs are not generalizable as we discussed in section 3.6. Therefore, we determined that including such an implementation would not provide sufficient academic value.

This approach focuses on implementing a translation algorithm specifically for ADAPTO and should be a convenient entry point when this implementation will be realized in the future. This implementation is specific to ADAPTO because of the chosen granularity for the translation. In the case of ADAPTO, we need to only use the actions of the activity diagrams that correspond to step definitions. Apart from this specialization, this implementation guide is applicable to other activity diagrams.

5.1 Step-by-step approach

The translation algorithm consists of two steps: encoding a SysML activity diagram in the adapted grammar and translating the adapted grammar to the TORXAKIS modelling language. The way we designed the adapted grammar makes it that the first step, encoding activity diagrams in the adapted grammar, should not require manual input after it is implemented. The second step however, may require manual input when using the implemented translation algorithm.

5.1.1 Encoding Activity Diagrams

First, we need to encode the activity diagram in the Adapted Grammar. For this, we are using the underlying structure for a SysML activity diagram. The visual diagram is derived from the underlying XML source code. An example of such an XML file can be seen in appendix C. This code describes the activity diagram from figure 2.19.

In theory, the programming language used for this implementation does not matter, however, we recommend using a statically typed language that supports sum types, since this allows us to encode the adapted grammar in a type-safe and elegant way. Swift and Rust are examples of languages that support this.

To transform this XML to the Adapted Grammar (figure 3.3), the following steps are required:

1. First, we create the datastructures required to encode the adapted grammar. This could be done by creating an enumeration type for each constructor in the grammar: \mathcal{N} , \mathcal{A} and \mathcal{B} . Each enumeration should contain a case for each possible term that type can be. So for example, the enumeration that describes \mathcal{N} contains 8 cases. The cases that contain a recursive term should have an argument with the type of that term. All cases with a label should have an argument that describes this label. This could be a String or an Integer for example. The cases that have an action name should have an argument of type String for this name.
2. The XML format of SysML is described in [12]. To encode an XML file in our datastructures, we need to traverse the abstract syntax tree (AST) of the code. We recommend using an XML parsing library if this functionality is not built-in in the programming language that is used. Now we should be able to parse an XML file and recursively traverse through the AST. We learn from appendix C that an activity diagram is an `ownedBehaviour` with an attribute in the tag that says `xmi:type='uml:Activity'`, so we can start our process when this tag is reached.
3. The children of this tag are the elements in the activity diagram. In XML, the order of the children is not specified. However, we can use the identifiers of the elements to determine which nodes are connected to other nodes.
4. In the traversal of the AST, we should keep track of all nodes and edges. Keep in mind that we have to recursively traverse the `group` construct. For each node we visit, we add the identifier and the type to a list of all constructs. For edges, we store the identifiers of the source and target nodes. Keep in mind that we omit action pins in the adapted grammar. When an action pin is visited, we can store the identifier and correspond it to the action the pin is attached to.
5. Since most terms require a label as an identifier, we need this label to be unique. Therefore, we keep track of a global counter that is used for the label. If this counter is used in a label, we increment this counter.
6. Now we have a list of all relevant constructs. We start by finding the initial node. If there are multiple initial nodes, we implement the idea illustrated in figure 3.1. For each of the edges, we check if the source is the initial node. If so, we can create our first term $a.t \mapsto \mathcal{N}$ with a the name of the activity diagram. \mathcal{N} will be the term that the initial node is connected to. This process of looking up edges continues with the node that is connected to the initial node. When a fork or decision node is reached, the recursive process is applied on both sub-terms. When on all paths a *FlowFinal* or *ActivityFinal* node is reached, we finish the procedure. We now have a term in the Adapted Grammar that describes the activity diagram.

5.1.2 Reduction

After the translation, we would like to reduce the resulting model as much as possible according to the reductions listed in section 3.5. Even though the mentioned reductions are applied after the translation to TORXAKIS, the better choice is to apply these reductions on the adapted grammar (or intermediate representation in the code) when implementing the translation algorithm. The reductions can be implemented as follows:

- Superfluous parentheses in the adapted grammar term of the encoded activity diagram can be recognized by checking if the parentheses are needed when a closing parenthesis is added to the translation. This usually occurs when translating a fork or decision.
- Reductions regarding EXIT statements can be recursively applied on the root term of the encoded activity diagram. It can be checked if a sub-term is a FlowFinal or ActivityFinal node in the implementation.
- If an \mathcal{A} term only has an EXIT statement as the body, it can simply be removed. It should be recursively checked if this term is used anywhere. Each occurrence should be replaced by a FlowFinal node, after which the reduction of the previous point should be applied.
- To check if a double decision has two identical sub-terms, terms should be equatable. The implementation of this is programming language dependent.

These reductions should be applied until the term does not change anymore after the next reduction. This can be tracked by a Boolean variable that is initialized with `false` and is set to `true` when a reduction is applied. After the reduction, this variable can be checked. If it is `true`, do another reduction step. Otherwise, stop. Set the variable back to `false`.

5.1.3 Translating to TorXakis

Now that we have encoded an activity diagram in the adapted grammar, we can translate this to TORXAKIS. We perform the following steps to achieve this:

1. Since we have defined enumerations for each constructor in the grammar, we can define a function for each enumeration that translates each case in the enumeration to the TORXAKIS code that represents this term. We can use a String as the return type for this function. If we reach a case that has another term that needs to be translated in its arguments, we recursively call the function that translates this term. We use the result of this translation in the result of the translation of the entire term.
2. This approach works for all but two terms in the adapted grammar: The *Join* and the *Action*. We will explain how to deal with these two cases.
 - **Join:** When we reach a join, we stop the current flow with the EXIT keyword. Then, we need to continue where the parallelization has started using the enable operator (`>>>`). We use a stack to keep track of the latest fork that we visited while traversing the AST. When we visit a *Fork*, we push the translation together with an identifier on the stack. We also use this identifier as a placeholder for where the translated fork must be inserted. When we visit a *Join*, we pop the top value

of the stack and add the translation of the continuation of the *Join* to this String with the enable operator. Then we add this entire String to another map with the identifier as the key. After everything is translated, we add all remaining values of the stack to the map (this happens when not all forks are accompanied by a join). Finally, we replace the identifiers placeholder of the forks with the values in the map.

- **Action:** We have discussed in section 3.6 that the translation of actions require manual input. When an action is visited while traversing the AST, the implementation can ask if the action should be translated to an input, output or internal step. There is another method possible that does not require human interaction. In the ADAPTO behavioural model, the step definitions are considered atomic. With other words, the step definitions define the granularity of the model. Using this fact, we can create a list of these step actions including whether they can be considered an input or output. When an action in the activity diagram is visited, we check if this activity is in the list. If it is, we create an input or output, depending on what the step definition is. If not, we create an internal step. Since the activities in the activity diagrams are written in plain text and do not necessarily correspond to a step definition, this method might not be feasible.
3. For each translated action, we need to add a constructor to the `Transition` type we discussed in section 3.3. We also need to add support for each translation action in the adapter. We do this by calling the corresponding step definition when the adapter receives a message containing the action.
 4. To complete the translation, we need to include the boilerplate for the channel definition, connection definition and the model definition, which is laid out in section 2.3.3.

This answers **SQ3**: Which steps are required to implement the formalization algorithm to be used on a real world system?

5.2 Conclusion

The step-by-step guide shows how the translation algorithm from section 3.4 can be implemented and adapted to suit ADAPTO. Even though the translation algorithm does not support data flow, we deem that implementing the translation algorithm is achievable and a viable option for using MBT with SysML activity diagrams, because the lack of data flow only limits the randomization of inputs in TORXAKIS, but not the actual behaviour when hard coding the predetermined arguments. To make the application of the implemented translation algorithm easier, we can optimize the manual steps. The labelling of actions in activity diagrams can be streamlined by using an identifier in each label that corresponds to a step definition for example. This makes the manual step only a one time effort.

Chapter 6

Conclusion & Discussion

In this chapter, we conclude the research in this thesis by answering the research question. We interpret the conclusion in the discussion. Finally, we summarize a recommendation for Vanderlande concerning using Model-Based Testing (MBT) for ADAPTO and perhaps for other systems in their company.

Let us recall the research question: *To what extent can SysML models be used for Model-Based Testing?*

We found that the behavioural diagrams of SysML were contenders to be used as a translation source for MBT, because the model that is used for MBT to test if an implementation adheres to it describes the behaviour of the System Under Test (SUT). Preceding academic work has been done for the *FormaSig* project[28] on using the SysML state machine diagrams in combination with the SysML structural diagrams to derive formal mCRL2 models which can be used for formal verification and MBT. We found that using SysML activity diagrams for the translation to TORXAKIS was better suited in our situation, because the only behavioural SysML diagrams that are substantially used for ADAPTO are activity diagrams. This is explained more in detail in the answer on *SQ1: Which formalization language is most suitable for SysML models in the case of MBT?* in section 3.2. We defined a translation algorithm based on the formalization of SysML activity diagrams in other preceding work. We therefore showed that it is possible to use SysML models for Model-Based Testing to the extent of the limitations of the translation algorithm by translating SysML activity diagrams to the TORXAKIS modelling language. The translation algorithm can mostly be used on SysML activity diagrams in other industries as well, because the only ADAPTO specific steps from the case study are preprocessing the activity diagrams such that they adhere to our predefined granularity. When all actions of an activity diagram already correspond to inputs and outputs of the modelled system, or if a custom preprocessing step is defined, the translation algorithm can be applied on those activity diagrams. The translation algorithm has the limitation that manual input is required by someone with domain knowledge about ADAPTO.

The fact that domain knowledge is required is a limitation, because the translation algorithm cannot be effectively applied on a SysML activity diagram without a domain expert. This is because the granularity of the tested actions needs to be determined. It may be the case that not all actions modelled in an activity diagram need to be used for MBT or need to be considered as internal steps. Furthermore, the actions that will be used for MBT need to

be categorized as an input or output, because this information is not encoded in an activity diagram. This categorization requires domain knowledge. Each action in the activity diagram does not correspond to multiple inputs, outputs or internal actions, because the activity diagrams of ADAPTO are modelled using the step definitions, which are the inputs or outputs, supplemented with other actions which can be considered as internal actions. The need for domain knowledge is further explained in the answer on *SQ2* in section 4.7.1.

Domain knowledge is also required for the manual input that is required for the translation algorithm. For each action in an activity diagram, it needs to be determined if said action is an internal action or need to be used for MBT. In the latter case, the action needs to be categorized as in input or output, because this information is needed for MBT. We did find that such an action is usually an output, because an activity diagram describes the action flow for a certain procedure, which can be considered the input. It can however not be assumed that an action is always an output. The fact that the translation algorithm requires manual input means that it cannot be fully implemented. We explain how to implement the translation algorithm and how to deal with its limitations in the answer of *SQ3* in chapter 5.

6.1 Discussion

Now that we have concluded that using SysML activity diagrams for MBT with TORXAKIS is possible, albeit with limitations, we will discuss if this process is desirable. The goal of developing this translation algorithm is to minimize manual labour to create behavioural models for MBT. We found that the translation algorithm requires manual input by someone with domain knowledge. In some circumstances, it could be that this amount of work by a domain expert is similar to the work it would require to develop a behavioural model without the use of SysML models. For example, it might be that using the translation algorithm is a short-term temporary solution, or that for a certain company the SysML activity diagrams are developed very precise and are therefore well suited for using the translation algorithm; when the granularity of the actions in the activity diagram correspond to the inputs and outputs for MBT for example. If a behavioural model is developed manually by a domain expert, it could be that the resulting model is ‘better’ than the model generated from the translation algorithm. Here ‘better’ can be defined as shorter, more compact, uncluttered, better performant and more precise. In chapter 7 we mention that research can be done to compare a model generated by the translation algorithm with a model developed by a domain expert, as well as additional research ideas for future work.

Furthermore, we would like to point out that depending on the software development lifecycle activity diagrams are useful for translation to MBT. To clarify, when a SysML activity diagram is modeled after a current implementation, the activity diagram acts more as an implementation instead of a specification. This means that the activity diagram describes how the implementation behaves instead of specifying how the implementation is allowed to behave. On the contrary, activity diagrams can be used and are more likely to be used as a specifications over implementations when the activity diagrams are modeled as a specification for the software before it is implemented. This observation implies that the usefulness of using SysML activity diagrams for MBT depends on the function of SysML models in a specific software development lifecycle. When activity diagrams are modelled independently

of an existing implementation there is a higher probability of finding mistakes or subtleties compared to when the activity diagrams are modeled dependently on the implementation.

6.2 Recommendation

This section provides a recommendation for Vanderlande on using MBT in their software development lifecycle.

Foremost, we recommend that Vanderlande uses MBT for testing their software, because MBT has numerous advantages over the traditional testing techniques that are currently used in the company: it solves the problem of an exponential growing number of test cases as the systems get more complex and it ensures more reliable software.

- The work required to apply MBT to a system does not scale exponentially with the complexity of the SUT. This fact was the premise of this thesis. Instead, only the model that describes the behaviour of the system needs to be maintained. With complex systems, the time it costs to create these models outweigh the time it costs to keep up with the testing work of the currently used testing techniques. This saves money in the medium to long term.
- Since a behavioural model for MBT describes the behaviour of the SUT, test cases can be derived from this model. Since these test cases are generated automatically, they can cover cases that could be overlooked when test cases are created manually. This implies that using MBT is less error-prone and this will result in more reliable software, which is important for the clients of Vanderlande.

The only work it costs to use MBT is creating a behaviour model. In this thesis, we explored the feasibility of using the already existing SysML models as a basis for automatically generating these models. We found that we can use the activity diagrams to generate a TORXAKIS model. However, we found that using our algorithm would still require manual input. Specifically, it has to be decided which actions in an activity diagram need to be considered internal steps, inputs or outputs for the MBT tool. Furthermore, when the SysML models are updated, the manual inputs need to be updated as well.

We would recommend implementing this translation algorithm of section 3.4 using the guide from chapter 5. To minimize the manual work required when using this translation algorithm, we recommend labeling the actions in activity diagrams consistently. For example, by using an identifier in the label which maps to a step definition. This way, a table which maps identifiers to step definitions only needs to be created once.

To ensure that the TORXAKIS models yielded from the translation algorithm comply to the ADAPTO implementation, we would recommend hiring or educating a modeller with knowledge about MBT. The benefit for doing this is that, provided this modeller has domain knowledge about the systems that need to be modelled, the modeller exactly knows how the system needs to be modelled for MBT. Furthermore, it will be easier to keep this model updated.

Finally, we would recommend using TORXAKIS as the MBT tool, since the translation algorithm in this thesis is based on TORXAKIS and TORXAKIS is free under the BSD3 license. If Vanderlande prefers to use maintained software with active support, we would recommend

using Axini as the MBT tool. The fact that Axini supports the modelling of time while TORXAKIS does not should not be a decisive factor when choosing an MBT tool, since this problem is solved by setting the timeout of TORXAKIS, which is mentioned in section 4.3.

Chapter 7

Future Work

In this chapter, we present six research possibilities for future work. We explain each subject and elaborate why research on the topic would have academic value. These research ideas are formed while performing the research in this thesis. This could either be because a certain question could not be answered yet, or because an interesting idea was thought of that is outside the scope of this thesis.

- The first suggestion is to implement the translation algorithm from SysML activity diagrams to TORXAKIS mentioned in section 3.4. The fact that part of the algorithm requires manual input should be considered. Chapter 5 can be used as a reference for the implementation. However, since this chapter focuses on the implementation of the ADAPTO system specifically, we do not have to filter the actions. This implies that we assume the granularity of the activity diagram: namely that each action correspond to an input or output for Model-Based Testing (MBT). As a result, the implementation would be more generic, which has more academic value.
- We have defined a translation algorithm from SysML activity diagrams to TORXAKIS which abstracts over data flow between actions in activity diagrams. Even though we briefly mentioned how this data flow could be considered in the translation algorithm, further research can be done to actually formalizing this data flow. This research could also include a comparison between the translation without or with modelling data flow to see what added value the latter algorithm provides.
- We have shown that it is possible to partially derive behavioural models for MBT from SysML activity diagrams. Provided that the activity diagrams correctly describe the behaviour of the system according to the semantic derivation rules from figure 2.3, we can apply MBT. Further research can be done on using the output of an MBT tool to extract the information required to figure out mistakes in the software. This is not trivial, because the model used for MBT is derived from the activity diagrams, which describe the behaviour of the software implementation. So there are two steps required from source code to model, which need to be traversed in the opposite direction from the output of the MBT tool to fixing the errors in the source code.
- The translation algorithm (section 3.4) could be adapted to work with Axini. To do this, the Adapted Grammar could be altered to suit the translation to the Axini modelling language better. For example, the modelling of time could be allowed by the grammar.

The resulting algorithm could be compared to the algorithm mentioned in this thesis to find which MBT tool is better suited under what circumstances for using SysML activity diagrams as a basis for creating the behavioural models. Note that this is not required for ADAPTO, but using Axini might be better suited for other systems modelled with SysML activity diagrams compared to TORXAKIS.

- Since the translation algorithm requires manual input by someone with domain knowledge about the to be tested system, which is also the case by manually developing a behavioural model, research can be done to the properties of a behavioural model from both instances. Criteria that can be used to determine which model is better can be: which model is shorter, more compact, independent, uncluttered, requires less time to develop, better performant or more precise.
- The highest bar for using MBT for a system is developing the behavioural model. Research could be done to the possibilities to lower the bar for using MBT. This could either be done by using already developed parts of a software product (e.g. automatic/gherkin tests) and generate a model from it *or* by developing a tool in which a model can be defined, after which code is generated which adheres to the model.
- Since the semantics of SysML behavioural diagrams are not formally defined (i.e. they are semi-formal), research could be done on developing a systems modelling tool that has the benefits of SysML (easy to model systems, simple to understand due to visual diagrams) while having a formally defined semantics that could easily (perhaps automatically) be used for MBT. This research could be performed bottom-up, i.e. start with SysML and improve or formalize it such that using MBT on a SysML model becomes trivial *or* top-down, i.e. start with an MBT tool and develop a tool that allows the modeller to view and edit the model visually. Such a systems modelling tool would lower the bar for using MBT for the industry, which in turn helps to reduce testing cost and yield more reliable software.

Bibliography

- [1] Takahiro Ando, Hirokazu Yatsu, Weiqiang Kong, Kenji Hisazumi, and Akira Fukuda. Formalization and model checking of sysml state machine diagrams by csp. In *International Conference on Computational Science and Its Applications*, pages 114–127. Springer, 2013.
- [2] Axel Belinfante. Jtorx: A tool for on-line model-driven test derivation and execution. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 266–270. Springer, 2010.
- [3] Axel Belinfante. Jtorx: exploring model-based testing. 2014.
- [4] Machiel van der Bijl, Arend Rensink, and Jan Tretmans. Compositional testing with ioco. In *International Workshop on Formal Approaches to Software Testing*, pages 86–100. Springer, 2003.
- [5] Mark Bouwman, Bas Luttik, and Djurre van der Wal. A formalisation of sysml state machines in mcrl2. In Kirstin Peters and Tim A. C. Willemse, editors, *Formal Techniques for Distributed Objects, Components, and Systems*, pages 42–59, Cham, 2021. Springer International Publishing.
- [6] Vincent Bruijn. Model-based testing with graph grammars. Master’s thesis, University of Twente, 2013.
- [7] Hans-Erik Eriksson, Magnus Penker, Brian Lyons, and David Fado. *UML 2 toolkit*. John Wiley & Sons, 2003.
- [8] Lars Frantzen, Jan Tretmans, and Tim AC Willemse. A symbolic framework for model-based testing. In *Formal approaches to software testing and runtime verification*, pages 40–54. Springer, 2006.
- [9] Jean-Marie Gauthier. Test generation for rtes from sysml models: Context, motivations and research proposal. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, pages 503–504, 2013.
- [10] Jan Friso Groote, Jeroen Keiren, Aad Mathijssen, Bas Ploeger, Frank Stappers, Carst Tankink, Yaroslav Usenko, Muck van Weerdenburg, Wieger Wesselink, Tim Willemse, et al. The mcrl2 toolset. In *Proceedings of the International Workshop on Advanced Software Development Tools and Techniques (WASDeTT 2008)*, page 53, 2008.

-
- [11] Jan Friso Groote, Aad Mathijssen, Michel Reniers, Yaroslav Usenko, and Muck Van Weerdenburg. The formal specification language mcrl2. In *Dagstuhl Seminar Proceedings*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2007.
- [12] Object Modelling Group. Detailed description of steps for translating sysml models to torxakis. Omg.org, 2022.
- [13] Kyra Hameleers. Detailed description of steps for translating sysml models to torxakis. Vanderlande, 2022.
- [14] Matthew Hause et al. The sysml modelling language. In *Fifteenth European Systems Engineering Conference*, volume 9, pages 1–12, 2006.
- [15] Jon Holt and Simon Perry. *SysML for systems engineering*, volume 7. IET, 2008.
- [16] IEEE. Ieee standard for voltage endurance testing of form-wound coils and bars for hydrogenerators. *IEEE Std 1553-2002*, pages 1–12, 2003.
- [17] ISO/IEC. International standard for software and systems engineering—software testing—part 3:test documentation - redline. *ISO/IEC/IEEE 29119-3:2021(E) - Redline*, pages 1–274, 2021.
- [18] Yosr Jarraya, Mourad Debbabi, and Jamal Bentahar. On the meaning of sysml activity diagrams. In *2009 16th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems*, pages 95–105, 2009.
- [19] Tianyao Li, Shigeru Tsubota, and Koji Hirono. Gherkin syntax extension for parameterization of network switch configurations in test specification. In *2017 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 12–14, 2017.
- [20] Johan Lilius and Ivn Porres Paltor. The semantics of uml state machines. 1999.
- [21] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.
- [22] University of Twente. TORX – test tool information. <http://fmt.cs.utwente.nl/tools/torx>.
- [23] Samir Ouchani, Otmane Ait Mohamed, and Mourad Debbabi. A formal verification framework for sysml activity diagrams. *Expert Systems with Applications*, 41(6):2713–2728, 2014.
- [24] Strategic Planning. The economic impacts of inadequate infrastructure for software testing. *National Institute of Standards and Technology*, page 1, 2002.
- [25] GJ Tretmans and Hendrik Brinksma. Torx: Automated model-based testing. In *First European Conference on Model-Driven Software Engineering*, pages 31–43, 2003.
- [26] GJ Tretmans and Pierre van de Laar. Model-based testing with torxakis: the mysteries of dropbox revisited. 2019.

- [27] Jan Tretmans. Model based testing with labelled transition systems. In *Formal methods and testing*, pages 1–38. Springer, 2008.
- [28] Djurre van der Wal. Formal methods in railway signalling infrastructure standardisation processes. *Leveraging Applications of Formal Methods, Verification and Validation*, page 500.

Appendix A

Derivation Rules Activity Calculus

INIT-1	$\overline{\iota \succ \mathcal{N}} \longrightarrow_1 \bar{\iota} \succ \mathcal{N}$	ACT-1	$\overline{\overline{l:a^k} \succ \mathcal{M}}^n \longrightarrow_1 \overline{\overline{l:a^{k+1}} \succ \mathcal{M}}^{n-1} \quad \forall n > 0$
INIT-2	$\bar{\iota} \succ \mathcal{N} \longrightarrow_1 \iota \succ \overline{\mathcal{N}}$	ACT-2	$\overline{\overline{l:a^k} \succ \mathcal{M}}^n \xrightarrow{a}_1 \overline{\overline{l:a^{k-1}} \succ \overline{\mathcal{M}}}^n \quad \forall k > 0$
INIT-3	$\frac{\mathcal{M} \xrightarrow{\alpha}_q \mathcal{M}'}{\iota \succ \mathcal{M} \xrightarrow{\alpha}_q \iota \succ \mathcal{M}'}$	ACT-3	$\frac{\mathcal{M} \xrightarrow{\alpha}_q \mathcal{M}'}{\overline{\overline{l:a^k} \succ \mathcal{M}}^n \xrightarrow{\alpha}_q \overline{\overline{l:a^k} \succ \mathcal{M}'}^n}$

Figure A.1: Rules for initial

Figure A.2: Rules for action prefixing

		MERG-1	$\frac{\mathcal{B}[\bar{l}: \text{Merge}(\mathcal{M})^n, \bar{l}^k] \longrightarrow_1}{\mathcal{B}[\bar{l}: \text{Merge}(\mathcal{M})^{n+1}, \bar{l}^{k-1}] \quad \forall k \geq 1}$
		MERG-2	$\overline{l: \text{Merge}(\mathcal{M})^n} \longrightarrow_1 \overline{l: \text{Merge}(\overline{\mathcal{M}})^{n-1}} \quad \forall n \geq 1$
FLOW-FINAL	$\overline{l: \otimes}^n \longrightarrow_1 \overline{l: \otimes}^{n-1} \quad \forall n > 0$	MERG-3	$\frac{\mathcal{M} \xrightarrow{\alpha}_q \mathcal{M}'}{\overline{l: \text{Merge}(\mathcal{M})^n} \xrightarrow{\alpha}_q \overline{l: \text{Merge}(\mathcal{M}')^n}}$
FINAL	$\mathcal{B}[\bar{l}: \odot] \longrightarrow_1 \mathcal{B} $		

Figure A.3: Rules for finals

Figure A.4: Merge rules

FORK-1	$\overline{l: \text{Fork}(\mathcal{M}_1, \mathcal{M}_2)^n} \longrightarrow_1 \overline{l: \text{Fork}(\overline{\mathcal{M}}_1, \overline{\mathcal{M}}_2)^{n-1}} \quad \forall n > 0$
FORK-2	$\frac{\mathcal{M}_1 \xrightarrow{\alpha}_q \mathcal{M}'_1}{\overline{l: \text{Fork}(\mathcal{M}_1, \mathcal{M}_2)^n} \xrightarrow{\alpha}_q \overline{l: \text{Fork}(\mathcal{M}'_1, \mathcal{M}_2)^n}}$ $\overline{l: \text{Fork}(\mathcal{M}_2, \mathcal{M}_1)^n} \xrightarrow{\alpha}_q \overline{l: \text{Fork}(\mathcal{M}_2, \mathcal{M}'_1)^n}$

Figure A.5: Fork rules

$$\begin{array}{l}
 \text{PDEC-1} \quad \overline{l: \text{Decision}_p(\langle g \rangle \mathcal{M}_1, \langle \neg g \rangle \mathcal{M}_2)^n} \longrightarrow_p \overline{l: \text{Decision}_p(\langle tt \rangle \overline{\mathcal{M}}_1, \langle ff \rangle \overline{\mathcal{M}}_2)^{n-1}} \quad \forall n > 0 \\
 \text{PDEC-2} \quad \overline{l: \text{Decision}_p(\langle g \rangle \mathcal{M}_1, \langle \neg g \rangle \mathcal{M}_2)^n} \longrightarrow_{1-p} \overline{l: \text{Decision}_p(\langle ff \rangle \mathcal{M}_1, \langle tt \rangle \overline{\mathcal{M}}_2)^{n-1}} \quad \forall n > 0 \\
 \text{PDEC-3} \quad \frac{\mathcal{M}_1 \xrightarrow{q} \mathcal{M}'_1}{\overline{l: \text{Decision}_p(\langle g \rangle \mathcal{M}_1, \langle \neg g \rangle \mathcal{M}_2)^n} \xrightarrow{q} \overline{l: \text{Decision}_p(\langle g \rangle \mathcal{M}'_1, \langle \neg g \rangle \mathcal{M}_2)^n}} \\
 \overline{l: \text{Decision}_p(\langle g \rangle \mathcal{M}_2, \langle \neg g \rangle \mathcal{M}_1)^n} \xrightarrow{q} \overline{l: \text{Decision}_p(\langle g \rangle \mathcal{M}_2, \langle \neg g \rangle \mathcal{M}'_1)^n}
 \end{array}$$

Figure A.6: Probabilistic decision rules

$$\begin{array}{l}
 \text{DEC-1} \quad \overline{l: \text{Decision}(\langle g \rangle \mathcal{M}_1, \langle \neg g \rangle \mathcal{M}_2)^n} \longrightarrow_1 \overline{l: \text{Decision}(\langle tt \rangle \overline{\mathcal{M}}_1, \langle ff \rangle \overline{\mathcal{M}}_2)^{n-1}} \quad \forall n > 0 \\
 \text{DEC-2} \quad \overline{l: \text{Decision}(\langle g \rangle \mathcal{M}_1, \langle \neg g \rangle \mathcal{M}_2)^n} \longrightarrow_1 \overline{l: \text{Decision}(\langle ff \rangle \mathcal{M}_1, \langle tt \rangle \overline{\mathcal{M}}_2)^{n-1}} \quad \forall n > 0 \\
 \text{DEC-3} \quad \frac{\mathcal{M}_1 \xrightarrow{q} \mathcal{M}'_1}{\overline{l: \text{Decision}(\langle g \rangle \mathcal{M}_1, \langle \neg g \rangle \mathcal{M}_2)^n} \xrightarrow{q} \overline{l: \text{Decision}(\langle g \rangle \mathcal{M}'_1, \langle \neg g \rangle \mathcal{M}_2)^n}} \\
 \overline{l: \text{Decision}(\langle g \rangle \mathcal{M}_2, \langle \neg g \rangle \mathcal{M}_1)^n} \xrightarrow{q} \overline{l: \text{Decision}(\langle g \rangle \mathcal{M}_2, \langle \neg g \rangle \mathcal{M}'_1)^n}
 \end{array}$$

Figure A.7: Non-deterministic decision rules

$$\begin{array}{l}
 \text{JOIN-1} \quad \mathcal{B}[\overline{l: x.\text{Join}(\mathcal{M})^n}, \overline{l^{k_x} \{x-1\}}] \longrightarrow_1 \mathcal{B}[\overline{l: x.\text{Join}(\overline{\mathcal{M}})}, \overline{l \{x-1\}}] \quad x > 1, n \geq 1, k_x \geq 1 \\
 \text{JOIN-2} \quad \overline{l: 1.\text{Join}(\mathcal{M})^n} \longrightarrow_1 \overline{l: 1.\text{Join}(\overline{\mathcal{M}})^{n-1}} \quad n \geq 1 \\
 \text{JOIN-3} \quad \frac{\mathcal{M} \xrightarrow{q} \mathcal{M}'}{\overline{l: x.\text{Join}(\mathcal{M})^n} \xrightarrow{q} \overline{l: x.\text{Join}(\mathcal{M}')^n}}
 \end{array}$$

Figure A.8: Join rules

Appendix B

Adapter

```
1 using System;
2 using System.Threading;
3 using System.Diagnostics;
4 using System.IO;
5 using System.Net;
6 using System.Net.Sockets;
7 using System.Linq;
8 using System.Collections.Generic;
9 using System.Xml;
10
11 namespace TorXakisAdapter
12 {
13     class Program
14     {
15         static void Main(string[] args)
16         {
17             bool logging = true;
18             try
19             {
20                 TcpListener server = new TcpListener(IPAddress.Parse("127.0.0.1"),
21                 ↪ 7890);
22                 server.Start();
23
24                 Console.WriteLine("Waiting for TorXakis");
25
26                 // Connect with TorXakis
27                 TcpClient client = server.AcceptTcpClient();
28
29                 // Create Streams
30                 BufferedStream stream = new BufferedStream(client.GetStream());
31                 StreamReader reader = new StreamReader(stream);
32                 StreamWriter writer = new StreamWriter(stream);
33
34                 // Initialize coffee machine
35                 CoffeeMachine coffeeMachine = new CoffeeMachine();
36
37                 Console.WriteLine("Connected");
38
39                 string line;
40                 while ((line = reader.ReadLine()) != null)
41                 {
42                     XmlDocument xmlDocument = new XmlDocument();
43                     xmlDocument.LoadXml(line);
44                     XmlNode node = xmlDocument.ChildNodes.Item(0).ChildNodes.Item(0);
45                     XmlNodeList arguments = node.ChildNodes;
46                     if (logging)
```

```

47         Console.WriteLine("Input: " + node.Name);
48     }
49
50     int argumentIndex = 0;
51     int i()
52     {
53         return Int32.Parse(arguments.Item(argumentIndex++).InnerText);
54     }
55
56     string s()
57     {
58         return arguments.Item(argumentIndex++).InnerText;
59     }
60
61
62     switch (node.Name)
63     {
64         case "TurnOn":
65             coffeeMachine.TurnOn();
66             break;
67         case "PressButton":
68             coffeeMachine.StartMakingCoffee();
69             break;
70         case "TurnOff":
71             coffeeMachine.TurnOff();
72             break;
73     }
74 }
75
76 scenarioPlayerInteractor.StopEnvironment();
77
78 void write(string command, (string, object)[] arguments)
79 {
80     string argumentString = string.Join(string.Empty, arguments.ToList
↪ ().Select(argument => $"<{argument.Item1}>{argument.Item2}</{argument.Item1}>"))
↪ ;
81     string message = $"<TorXakisMsg><{command}>{argumentString}</{
↪ command}></TorXakisMsg>";
82     if (logging)
83     {
84         Console.WriteLine("Output: " + command);
85     }
86     writer.WriteLine(message);
87     writer.Flush();
88 }
89 }
90 catch (Exception e)
91 {
92     Console.WriteLine(e.ToString());
93 }
94 }
95 }
96 }

```

Appendix C

Coffee Machine XML Code

```
1 <ownedBehavior xmi:type="uml:Activity" xmi:id="
  ↳ EAID_AC000001_2081_4d99_89A6_3F7905DD104B" name="EA_Activity1" visibility="
  ↳ public">
2 <node xmi:type="uml:ActivityFinalNode" xmi:id="
  ↳ EAID_2CFF1FBC_9154_4588_B91F_8F4736149393" name="ActivityFinal" visibility="
  ↳ public">
3 <incoming xmi:idref="EAID_7CEB6951_6AD3_4c83_915B_A8D0B352938E" />
4 </node>
5 <node xmi:type="uml:InitialNode" xmi:id="EAID_5DF25D88_4B8E_4712_BEEB_27723BB6222F
  ↳ " name="ActivityInitial" visibility="public">
6 <outgoing xmi:idref="EAID_DAADBB4F_627B_4539_96BE_B193526CF680" />
7 </node>
8 <edge xmi:type="uml:ControlFlow" xmi:id="EAID_DAADBB4F_627B_4539_96BE_B193526CF680
  ↳ " visibility="public" source="EAID_5DF25D88_4B8E_4712_BEEB_27723BB6222F" target="
  ↳ EAID_1FFB060A_62D9_4d63_B9A2_35FA046A8290" />
9 <group xmi:type="uml:ActivityPartition" xmi:id="
  ↳ EAID_5D58F9F8_68CA_47dc_B122_D105B60EEAE8" name="Boiler" visibility="public">
10 <node xmi:idref="EAID_F0A626D3_531F_4169_8CAA_7F682FA03A73" />
11 <containedNode xmi:type="uml:Action" xmi:id="
  ↳ EAID_F0A626D3_531F_4169_8CAA_7F682FA03A73" name="Send water boiled" visibility="
  ↳ public">
12 <incoming xmi:idref="EAID_7843B214_FBF5_4676_B65D_D5248ED5BA0B" />
13 <output xmi:type="uml:OutputPin" xmi:id="
  ↳ EAID_E772B898_11EF_49d6_B1A8_9A09C1DB85F3" name="Pin1" visibility="public"
  ↳ ordering="FIFO">
14 <outgoing xmi:idref="EAID_EDEF6CAC_6040_4524_B5C6_0050CE2228C2" />
15 <type xmi:idref="EAID_FF0CBADA_0C16_4153_80ED_4EC9063FB387" />
16 </output>
17 </containedNode>
18 <containedEdge xmi:type="uml:ObjectFlow" xmi:id="
  ↳ EAID_EDEF6CAC_6040_4524_B5C6_0050CE2228C2" visibility="public" source="
  ↳ EAID_E772B898_11EF_49d6_B1A8_9A09C1DB85F3" target="
  ↳ EAID_DD3A08BF_BF6E_4c5f_BA6C_4E5D776A1367" />
19 <node xmi:idref="EAID_29E9F1B6_C365_4987_BBDE_83D5E952210D" />
20 <containedNode xmi:type="uml:Action" xmi:id="
  ↳ EAID_29E9F1B6_C365_4987_BBDE_83D5E952210D" name="Boil the water" visibility="
  ↳ public">
21 <incoming xmi:idref="EAID_6C439DF4_852A_4c6b_BA75_8DEB4D440430" />
22 <outgoing xmi:idref="EAID_C4C938BA_DA9A_45da_8E8F_D7527A31E063" />
23 <outgoing xmi:idref="EAID_7843B214_FBF5_4676_B65D_D5248ED5BA0B" />
24 </containedNode>
25 <containedEdge xmi:type="uml:ControlFlow" xmi:id="
  ↳ EAID_7843B214_FBF5_4676_B65D_D5248ED5BA0B" visibility="public" source="
  ↳ EAID_29E9F1B6_C365_4987_BBDE_83D5E952210D" target="
  ↳ EAID_F0A626D3_531F_4169_8CAA_7F682FA03A73" />
26 <containedEdge xmi:type="uml:ControlFlow" xmi:id="
  ↳ EAID_C4C938BA_DA9A_45da_8E8F_D7527A31E063" visibility="public" source="
  ↳ EAID_29E9F1B6_C365_4987_BBDE_83D5E952210D" target="
```



```

27     ↪ EAID_157D29CD_D250_4689_AFE7_16BCDF4E77C4" />
28     <node xmi:idref="EAID_157D29CD_D250_4689_AFE7_16BCDF4E77C4" />
29     <containedNode xmi:type="uml:SendSignalAction" xmi:id="
30     ↪ EAID_157D29CD_D250_4689_AFE7_16BCDF4E77C4" name="Send [water boiled]" visibility
31     ↪ ="public">
32         <incoming xmi:idref="EAID_C4C938BA_DA9A_45da_8E8F_D7527A31E063" />
33         <outgoing xmi:idref="EAID_BCCC01C1_4122_4069_BD6B_9128DF85ECDB" />
34         <target xmi:type="uml:InputPin" xmi:id="
35     ↪ EAID_BFCC5845_518D_4000_8BDE_0865FE8AC1EF" name="target" visibility="public"
36     ↪ ordering="FIFO" />
37     </containedNode>
38     <containedEdge xmi:type="uml:ControlFlow" xmi:id="
39     ↪ EAID_BCCC01C1_4122_4069_BD6B_9128DF85ECDB" visibility="public" source="
40     ↪ EAID_157D29CD_D250_4689_AFE7_16BCDF4E77C4" target="
41     ↪ EAID_CAEA0C64_7D17_4934_8B6C_510380F7D70A" />
42 </group>
43 <node xmi:type="uml:Action" xmi:id="EAID_E63E0237_2D40_4731_9740_20CBC5226FCF"
44     ↪ name="Grind coffee beans" visibility="public">
45     <incoming xmi:idref="EAID_CFEFFD05_99EF_4fb2_BC75_5F529556F7F7" />
46     <outgoing xmi:idref="EAID_BA289373_ACAA_44b9_8F57_77A96BD91BD3" />
47     <outgoing xmi:idref="EAID_2B613922_6B8B_4085_B29C_53DF353645C2" />
48 </node>
49 <edge xmi:type="uml:ControlFlow" xmi:id="EAID_2B613922_6B8B_4085_B29C_53DF353645C2
50     ↪ " visibility="public" source="EAID_E63E0237_2D40_4731_9740_20CBC5226FCF" target="
51     ↪ "EAID_E95C690C_5B2B_4df9_9605_35F211AEE719" />
52 <edge xmi:type="uml:ControlFlow" xmi:id="EAID_BA289373_ACAA_44b9_8F57_77A96BD91BD3
53     ↪ " visibility="public" source="EAID_E63E0237_2D40_4731_9740_20CBC5226FCF" target="
54     ↪ "EAID_B682FF92_AF78_4ce9_9C2C_5E29104025B1" />
55 <group xmi:type="uml:ActivityPartition" xmi:id="
56     ↪ EAID_7560CB75_62A5_45e1_AEB4_D6C8DAE238E9" name="Grinder" visibility="public" />
57 <group xmi:type="uml:ActivityPartition" xmi:id="
58     ↪ EAID_3ECC5A02_69A3_46b6_A93D_A6D5E8D835C2" name="Grinder" visibility="public">
59     <node xmi:idref="EAID_E95C690C_5B2B_4df9_9605_35F211AEE719" />
60     <containedNode xmi:type="uml:Action" xmi:id="
61     ↪ EAID_E95C690C_5B2B_4df9_9605_35F211AEE719" name="Send beans grinded" visibility="
62     ↪ "public">
63         <incoming xmi:idref="EAID_2B613922_6B8B_4085_B29C_53DF353645C2" />
64         <output xmi:type="uml:OutputPin" xmi:id="
65     ↪ EAID_817EBB61_DDF1_4c02_BE66_CDC34F52E281" name="Pin3" visibility="public"
66     ↪ ordering="FIFO">
67             <outgoing xmi:idref="EAID_1D5F6B83_FBD8_4bf2_8D2E_8E80098DE595" />
68             <type xmi:idref="EAID_6E96564F_BFC6_4d0d_A96E_5E548DD936E1" />
69         </output>
70     </containedNode>
71     <containedEdge xmi:type="uml:ObjectFlow" xmi:id="
72     ↪ EAID_1D5F6B83_FBD8_4bf2_8D2E_8E80098DE595" visibility="public" source="
73     ↪ EAID_817EBB61_DDF1_4c02_BE66_CDC34F52E281" target="
74     ↪ EAID_BCAE5E2F_6EEF_4793_B331_68F9653027C1" />
75     <node xmi:idref="EAID_B682FF92_AF78_4ce9_9C2C_5E29104025B1" />
76     <containedNode xmi:type="uml:SendSignalAction" xmi:id="
77     ↪ EAID_B682FF92_AF78_4ce9_9C2C_5E29104025B1" name="Send [Coffee grinded]"
78     ↪ visibility="public">
79         <incoming xmi:idref="EAID_BA289373_ACAA_44b9_8F57_77A96BD91BD3" />
80         <outgoing xmi:idref="EAID_FEF369CB_1AF4_4538_8A51_4903228D6B76" />
81         <target xmi:type="uml:InputPin" xmi:id="
82     ↪ EAID_48D0FFFFB_7042_472f_BF3F_7D86C5BC83CB" name="target" visibility="public"
83     ↪ ordering="FIFO" />
84     </containedNode>
85     <containedEdge xmi:type="uml:ControlFlow" xmi:id="
86     ↪ EAID_FEF369CB_1AF4_4538_8A51_4903228D6B76" visibility="public" source="
87     ↪ EAID_B682FF92_AF78_4ce9_9C2C_5E29104025B1" target="
88     ↪ EAID_AAE99A46_3EE1_451e_B54E_38F56F792B2A" />
89 </group>
90 <group xmi:type="uml:ActivityPartition" xmi:id="
91     ↪ EAID_BE6A6304_FDB7_41b4_BE1B_92B15B1FF7EB" name="Mixer" visibility="public">
92     <node xmi:idref="EAID_F062FA5B_12EF_4188_B50C_D013075CCEAB" />

```

```

63     <containedNode xmi:type="uml:Action" xmi:id="
↳ EAID_F062FA5B_12EF_4188_B50C_D013075CCEAB" name="Pour in a cup" visibility="
↳ public">
64         <incoming xmi:idref="EAID_4118333C_30B2_4ab5_8AB0_1B9D885787D8" />
65         <outgoing xmi:idref="EAID_7CEB6951_6AD3_4c83_915B_A8D0B352938E" />
66     </containedNode>
67     <containedEdge xmi:type="uml:ControlFlow" xmi:id="
↳ EAID_7CEB6951_6AD3_4c83_915B_A8D0B352938E" visibility="public" source="
↳ EAID_F062FA5B_12EF_4188_B50C_D013075CCEAB" target="
↳ EAID_2CFF1FBC_9154_4588_B91F_8F4736149393" />
68     <node xmi:idref="EAID_9614F308_ABA4_47a3_8938_80F181247AD8" />
69     <containedNode xmi:type="uml:Action" xmi:id="
↳ EAID_9614F308_ABA4_47a3_8938_80F181247AD8" name="Mix" visibility="public">
70         <incoming xmi:idref="EAID_6A361CDD_E0D8_456e_B69B_CDE3F0EF618E" />
71         <outgoing xmi:idref="EAID_4118333C_30B2_4ab5_8AB0_1B9D885787D8" />
72     </containedNode>
73     <containedEdge xmi:type="uml:ControlFlow" xmi:id="
↳ EAID_4118333C_30B2_4ab5_8AB0_1B9D885787D8" visibility="public" source="
↳ EAID_9614F308_ABA4_47a3_8938_80F181247AD8" target="
↳ EAID_F062FA5B_12EF_4188_B50C_D013075CCEAB" />
74     <node xmi:idref="EAID_DE10536F_4A53_479a_B28A_8EEA6E264C45" />
75     <containedNode xmi:type="uml:Action" xmi:id="
↳ EAID_DE10536F_4A53_479a_B28A_8EEA6E264C45" name="Receive beans grinded"
↳ visibility="public">
76         <outgoing xmi:idref="EAID_B7849965_3508_425d_9AA4_F0805DEEA6C9" />
77         <input xmi:type="uml:InputPin" xmi:id="
↳ EAID_BCAE5E2F_6EEF_4793_B331_68F9653027C1" name="Pin4" visibility="public"
↳ ordering="FIFO">
78             <incoming xmi:idref="EAID_1D5F6B83_FBD8_4bf2_8D2E_8E80098DE595" />
79             <type xmi:idref="EAID_6E96564F_BFC6_4d0d_A96E_5E548DD936E1" />
80         </input>
81     </containedNode>
82     <containedEdge xmi:type="uml:ControlFlow" xmi:id="
↳ EAID_B7849965_3508_425d_9AA4_F0805DEEA6C9" visibility="public" source="
↳ EAID_DE10536F_4A53_479a_B28A_8EEA6E264C45" target="
↳ EAID_583178B1_9A49_450c_B66E_BDF1A12E7CE5" />
83     <node xmi:idref="EAID_0572336B_A904_47c2_846E_F08704657D44" />
84     <containedNode xmi:type="uml:Action" xmi:id="
↳ EAID_0572336B_A904_47c2_846E_F08704657D44" name="Receive water boiled"
↳ visibility="public">
85         <outgoing xmi:idref="EAID_43CD027C_751F_46fd_AE54_D848961B167F" />
86         <input xmi:type="uml:InputPin" xmi:id="
↳ EAID_DD3A08BF_BF6E_4c5f_BA6C_4E5D776A1367" name="Pin2" visibility="public"
↳ ordering="FIFO">
87             <incoming xmi:idref="EAID_EDEF6CAC_6040_4524_B5C6_0050CE2228C2" />
88             <type xmi:idref="EAID_FF0CBADA_0C16_4153_80ED_4EC9063FB387" />
89         </input>
90     </containedNode>
91     <containedEdge xmi:type="uml:ControlFlow" xmi:id="
↳ EAID_43CD027C_751F_46fd_AE54_D848961B167F" visibility="public" source="
↳ EAID_0572336B_A904_47c2_846E_F08704657D44" target="
↳ EAID_583178B1_9A49_450c_B66E_BDF1A12E7CE5" />
92     <node xmi:idref="EAID_AAE99A46_3EE1_451e_B54E_38F56F792B2A" />
93     <containedNode xmi:type="uml:AcceptEventAction" xmi:id="
↳ EAID_AAE99A46_3EE1_451e_B54E_38F56F792B2A" name="Receive [Coffee grinded]"
↳ visibility="public">
94         <incoming xmi:idref="EAID_FEF369CB_1AF4_4538_8A51_4903228D6B76" />
95         <outgoing xmi:idref="EAID_653F0797_F98C_42d7_A132_3D742BAF03AE" />
96     </containedNode>
97     <containedEdge xmi:type="uml:ControlFlow" xmi:id="
↳ EAID_653F0797_F98C_42d7_A132_3D742BAF03AE" visibility="public" source="
↳ EAID_AAE99A46_3EE1_451e_B54E_38F56F792B2A" target="
↳ EAID_583178B1_9A49_450c_B66E_BDF1A12E7CE5" />
98     <node xmi:idref="EAID_CAEA0C64_7D17_4934_8B6C_510380F7D70A" />
99     <containedNode xmi:type="uml:AcceptEventAction" xmi:id="
↳ EAID_CAEA0C64_7D17_4934_8B6C_510380F7D70A" name="Receive [water boiled]"

```

```

100     ↪ visibility="public">
101         <incoming xmi:idref="EAID_BCCC01C1_4122_4069_BD6B_9128DF85ECDB" />
102         <outgoing xmi:idref="EAID_BEE40766_BAE6_4fdc_AFB6_52DB6ABEA996" />
103     </containedNode>
104     <containedEdge xmi:type="uml:ControlFlow" xmi:id="
105     ↪ EAID_BEE40766_BAE6_4fdc_AFB6_52DB6ABEA996" visibility="public" source="
106     ↪ EAID_CAEA0C64_7D17_4934_8B6C_510380F7D70A" target="
107     ↪ EAID_583178B1_9A49_450c_B66E_BDF1A12E7CE5" />
108 </group>
109 <node xmi:type="uml:ForkNode" xmi:id="EAID_1FFB060A_62D9_4d63_B9A2_35FA046A8290"
110 ↪ visibility="public">
111     <incoming xmi:idref="EAID_DAADB4F_627B_4539_96BE_B193526CF680" />
112     <outgoing xmi:idref="EAID_CFEFFD05_99EF_4fb2_BC75_5F529556F7F7" />
113     <outgoing xmi:idref="EAID_6C439DF4_852A_4c6b_BA75_8DEB4D440430" />
114 </node>
115 <edge xmi:type="uml:ControlFlow" xmi:id="EAID_6C439DF4_852A_4c6b_BA75_8DEB4D440430
116 ↪ " visibility="public" source="EAID_1FFB060A_62D9_4d63_B9A2_35FA046A8290" target="
117 ↪ "EAID_29E9F1B6_C365_4987_BBDE_83D5E952210D" />
118 <edge xmi:type="uml:ControlFlow" xmi:id="EAID_CFEFFD05_99EF_4fb2_BC75_5F529556F7F7
119 ↪ " visibility="public" source="EAID_1FFB060A_62D9_4d63_B9A2_35FA046A8290" target="
120 ↪ "EAID_E63E0237_2D40_4731_9740_20CBC5226FCF" />
121 <node xmi:type="uml:ForkNode" xmi:id="EAID_583178B1_9A49_450c_B66E_BDF1A12E7CE5"
122 ↪ visibility="public">
123     <incoming xmi:idref="EAID_BEE40766_BAE6_4fdc_AFB6_52DB6ABEA996" />
124     <incoming xmi:idref="EAID_B7849965_3508_425d_9AA4_F0805DEEA6C9" />
125     <incoming xmi:idref="EAID_653F0797_F98C_42d7_A132_3D742BAF03AE" />
126     <incoming xmi:idref="EAID_43CD027C_751F_46fd_AE54_D848961B167F" />
127     <outgoing xmi:idref="EAID_6A361CDD_E0D8_456e_B69B_CDE3F0EF618E" />
128 </node>
129 <edge xmi:type="uml:ControlFlow" xmi:id="EAID_6A361CDD_E0D8_456e_B69B_CDE3F0EF618E
130 ↪ " visibility="public" source="EAID_583178B1_9A49_450c_B66E_BDF1A12E7CE5" target="
131 ↪ "EAID_9614F308_ABA4_47a3_8938_80F181247AD8" />
132 </ownedBehavior>

```