Model checking test models

Author: Kevin de Berk
Supervisors: Prof. dr. Wan Fokkink, dr. ir. Machiel van der Bijl

February 14, 2014

Abstract

This thesis is about model checking testing models. These testing models are used during the automated testing of software systems and encode symbolic transition systems that are necessarily open. Open models need to be closed before they can be model checked by LTSmin. The open models are closed by the automatic generation and insertion of an environment process. They are then translated to DVE and model checked. The model closing algorithm does preserve the verifiable properties for the models that it can close, but cannot deal with models that have an infinitely large underlying labeled transition system. Future research is suggested to deal with this issue.

Keywords: model checking, closing open models, symbolic transition systems, labeled transition systems, Kripke structures, LTL, LTSmin, Ruby, automated testing, software development

Contents

List of Figures

List of Tables

List of Listings

List of Definitions

Chapter 1

Introduction

Axini is a young company that has specialized itself in a novel approach to the problem of software testing. This approach is called *model-based testing.* Software testing is an intrinsic part of software development where one checks whether the developed software actually conforms to its requirements. This task is necessary to ensure that the final software product can be accepted by the project *stakeholders.* A practical approach to software testing is to actively search for erroneous behavior [43]; one could try entering a suspicious input sequence and verify whether the output is still correct. Generally speaking, as the number of these attempts increases, so will the *confidence* in the correctness of the software increase [58]. The software product can be delivered to the stakeholders once this confidence has become sufficiently strong.

However, *"program testing can be used to show the presence of bugs, but never to show their absence"* [23]; something other than testing is necessary to be completely sure that the program is free of errors. To obtain such a definite answer, one might have to resort to formal methods and *model checking* in particular. This project is about enabling model checking for Axini, but not for verifying the correctness of the software. Rather, this project will allow Axini to model check the models that they use in their approach to software testing.

The rest of this chapter introduces the important concepts that were previously briefly mentioned, and finally describe how model checking can be a valuable tool for Axini in testing software correctness. Furthermore, this chapter contains a list of the project goals and finally describes the remaining contents of this thesis.

1.1 Axini and Model-based Testing

TestManager is an application that is created and maintained by Axini and that enables the automatic testing of software systems. A system that is being tested by TestManager is referred to as a *System Under Test* (SUT). TestManager's approach to automated testing is through black-box model-based functionality testing and a general overview of this approach is shown in figure 1.1. A user makes the *implementation* (the SUT) and a *specification* (the test model) available to TestManager. TestManager then derives a *test suite* from the specification and executes the test cases in the suite on the implementation. The result is a *verdict* on how well the implementation conforms to the specification [54, 52]. If one of the generated test cases failed, then the behavioral trace that led to the violation is shown to the user. He can then reproduce the offending behavior.

Model-based testing is a specific form of automated testing. Normally, automated testing only refers to the automatic execution of tests and sometimes also to the automatic analysis of test results [52]. In these cases, the test scenarios have to be manually written in a low-level tool-specific language. However, model-based testing not only includes both automatic test execution and automatic result analysis, but also the automatic generation of executable test cases. This approach has many benefits over other forms of software testing. The manual construction and maintenance of the specification requires less effort than the manual construction and maintenance of the individual test cases. Model-based testing also allows a much larger coverage of the SUT by being able to produce and test a significantly higher number of test cases than what could possibly be done manually.

For TestManager, a specification or *testing model* is a functional description of the requirements written in a language called Axini Modeling Language (AML), but is more often referred to

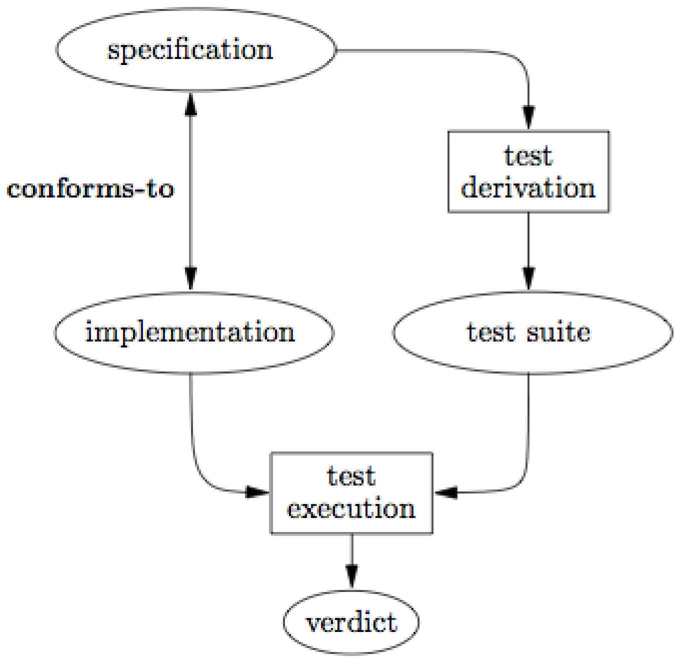


Figure 1.1: Overview of model-based testing, taken from [54]

as STS. This description lists the observable actions that the SUT can make along with the control structures that dictate which actions can be emitted in response to which stimuli, in what order and under what conditions. Appendix A gives a short overview of this language and this thesis uses this modelling language to define some example models.

A major issue with model-based testing is the problem of correctness. If the specification isn't correct with respect to the requirements, then any model-based test case based upon this specification may declare valid behavior to be incorrect, or worse, may fail to recognize invalid behavior as such when that behavior is emitted by the SUT. If the system requirements could be expressed formally, then a formal method may show that the test model is correct with respect to these formalized requirements. Model checking may be such a method.

1.2 Model checking

Model checking is a technique that can check whether a *Kripke structure*, or something that can be reduced to such a structure, is a model for a formula that is expressed in a temporal logic [15, 47]. A *model checker* translates a system that is normally specified in a high-level formalism to a Kripke structure; accepts a property specified in a temporal logic and then checks if the structure is a model of the property, or rather, if the property holds within the structure. It also produces a *counterexample* if it turned out that the structure wasn't a model for the property. The properties that can be checked can be grouped into two categories: *safety properties* and *liveness properties.* Safety properties specify that *"nothing bad will ever happen"*, e.g. the system will not halt and will never acquire an undesired property. Liveness properties are used to show that *"something good will happen eventually"*, e.g. the system will always accept input and will always respond to input [3].

An important application of model checking is the verification of communication protocols [25, 15]. The approach itself was first proposed by Clarke and Emerson while they were researching the synthesis of valid communication protocols from temporal logic formulas [16]. They first proposed the generation of a *synchronization skeleton* which could be easily extended to an actual implementation of the communication protocol that would be correct with respect to the original properties. However, during this research, they realized that the problem of checking models was much more important than the problem of synthesizing correct programs. Independently from Clarke and Emerson, Sifakis et al. had also invented model checking and presented their model checker CESAR in 1982 [47]. In 2007, Clarke, Emerson and Sifakis received the Turing Award for their invention of model checking.

Model checking is not the only formal method that can be used to verify whether a system is correct. Before model checking was invented, systems often had to be verified by the manual construction of proofs, using e.g. the Floyd-Hoare framework that allowed proving *total correctness* and *termination* of functional programs [25]. Applying these techniques to real-world programs was very difficult as these programs weren't necessarily functional, or didn't need to terminate at all [15]. In 1977, Pnuelli suggested a formalism to be used during manual proof construction that could unify the verification of sequential and concurrent programs [46]. His formalism allowed reasoning on both *invariants* (properties that didn't change through the program execution) and *eventuality* (statements that would become true during every possible execution). Model checking accepted this formalism as a method for expression temporal properties. Manually proving program correctness is still being done, and has been partially automated by *term rewriting systems* and *proof checkers*. However, these approaches still require significant user intervention and human ingenuity [20].

While model checking may be a fully automatic approach, it is effectively a *brute-force* solution to the problem of program verification. The most telling symptom of this is the combinatorial state space explosion that will occur whenever the model checker tries to check certain non-trivial systems. The phenomenon arises because the size of the underlying Kripke structure grows exponentially to the size of the system [17]. For non-trivial systems, the Kripke structure may even be too large to be stored in computer memory. However, the state space does not need to be stored *explicitly*, where each state is stored as an individual entity. States can also be stored *implicitly* or *symbolically*, where the entire state space is stored as e.g. a binary or multi-valued decision diagram [11, 13]. In 2001, Ciardo et al. reported that they were able to store the entire state space of a model with approximately 10^{627} states within 390KB of memory [14].

Currently, several model checkers are freely available as open source. SPIN is one of the most popular and powerful model checkers and has been successfully applied to several significant industrial test cases[1] [34]. Some model checkers are designed to only check for specific types of systems, e.g. Uppaal is designed to verify systems that can be modelled as a collection of timed automata [5] and Mocha can verify systems that are *reactive* or *open* [2]. Some of these are overtly academical in nature; they are rarely updated, not fully optimized and only seem to serve as case studies.

This project uses the model checker LTSmin [41, 9]. LTSmin is a model checker that has been specifically designed to be modular in such a fashion that it can accept multiple specification languages and offer each of these languages more or less the same state space optimizations and property verification algorithms. Most of these algorithms are also designed to be high-performance and can distribute their workload over multiple machines. LTSmin supports some of the popular model specification languages such as Promela, which was originally designed for SPIN; Uppaal, DVE and mCRL2. This feature gives the user the freedom to switch over to, or from, LTSmin without having to rewrite his models to a different specification language.

1.3 Model checking test models

This project is about enabling Axini to model check its `STS` models. Axini can then formally determine whether its model satisfy the system requirements by converting those requirements to formulas of temporal logic and having a model checker check whether the underlying Kripke structure is a model for these formulas. As this process is relatively fast and can be fully automated, model checking can be used during the development of the model to verify the correctness with respect to the SUT requirements. This may decrease the effort that is needed to construct a correct model. Model checking can therefor be useful addition to the existing Axini workflow.

Research Question 1. *Can we model check* `STS` *models?*

Unfortunately, there is no model checker that can accept models written in the `STS` language. A solution to this problem is not trivial. Most model checkers only accept *closed* models for checking, but the models used by TestManager are *open* and necessarily so. Existing model checkers that do support open models, such as Mocha, support too few features to be of good use. To bridge this gap and allow Axini to model check their models, this project had to research and construct a method for closing open models.

Research Question 2. *How can an open model be properly closed?*

[1]http://spinroot.com/spin/success.html

During the course of this project, a prototype was developed that will allow some `STS` models to be model checked. This model uses code that was developed by Axini to parse the `STS` models, and closes them by inserting a finite process that represents the maximum environment. This new environment only sends the messages that are needed to retain the behavior of the original model and is in this sense a minimal representation. The model checking is done by LTSmin and the closed `STS` models are translated to DVE so that LTSmin can access the model. Not all models can be closed. Models that use datatypes other than integers are currently not supported, but future support for such models is discussed later within this thesis. Models that have an infinitely large underlying Kripke structure are also not supported. Again, future work to deal with such models is discussed later on. While these limitations diminish the outcome of this project, we would like to highlight that the prototype is designed to produce only correct results for the models that fall within the scope. That is: the applied model normalizations, model closing and DVE translation algorithms are designed to retain the LTL-expressible model properties so that no false-negative or false-positive results can be generated. Nonetheless, the prototype is currently not capable of handling most models that are developed and used by Axini.

1.4 Document structure

The next three chapters describe the context: chapter 2 introduces some theoretical constructs that are useful for understanding the chapters that follow it, while 3 and 4 discusses the two separate domains that this project intends to connect: Axini's approach to automated testing, and model checking. The prototype is examined in chapter 5. This chapter starts with an explanation of the important decisions that were made, followed by an in-depth discussion of the methods of model minimizing, model closing and DVE translation. The following chapter, 6 contains an elaboration of the test results as well as a listing of all products that were delivered for this project. An evaluation of the research goals and the prototype, along with its limitations and the future work necessary for the success of this prototype, is given in chapter 7. Finally, this thesis is wrapped up with the conclusion in chapter 8.

Added to this document are three appendices. Appendix A gives an overview of the `STS` language in which many of the example models in this thesis are specified. Appendix B lists several models for which the model checker produced correct results, as is discussed in chapter 6. Lastly, appendix C is a suggested task assignment for the next intern or Axini employee who will continue this project.

Chapter 2

Background

This chapter gives a background for the concepts that are discussed later on in this thesis. The focus is placed on the concepts that are related to model checking. We discuss transition systems, temporal logics and verifiable properties. In the final section 2.4, we will give some examples of Kripke structures and discuss some of their LTL-expressible properties.

2.1 Models

The models that are considered in this chapter are mathematical constructs that can be expressed as graphs. In this section, we review three types of models: *a*) *symbolic transition systems*, which the STS models are evaluated to; *b*) *labeled transition systems*, which label edges with events; and *c*) *Kripke structures*, which label vertices with atomic propositions.

Model checking algorithms are normally specified for Kripke structures. To model check an STS model, it will have to be transformed to such a structure first. *Interpreting* a symbolic transition system to a labeled transition system is an important step in that transformation, as Kripke structures and labeled transition systems are relatively interchangeable [44].

Definition 1 (Kripke Structure). *A Kripke structure over a set of atomic propositions AP is a tuple $K = \langle S,\ R,\ L,\ I\rangle$ where S is the set of states, $R \subseteq S^2$ is the set of transitions, $I \subseteq S$ is the nonempty set of initial states, and $L : S \to 2^{AP}$ labels each state by a set of atomic propositions [17].*

Figure 2.1 contains a visual representation of the Kripke structure $\langle\{a,b,c\},\{(a,b),(b,c),(c,c)\},\{a \mapsto \{p\},b \mapsto \{p,q\},c \mapsto \{q\}\},\{a\}\rangle$, with $AP = \{p,q\}$. A formal verification of the formulas that are constructed from the elements in AP involves an inspection of the labeled states and their relationships with each other. Model checking is such a technique and is discussed extensively in chapter 4.

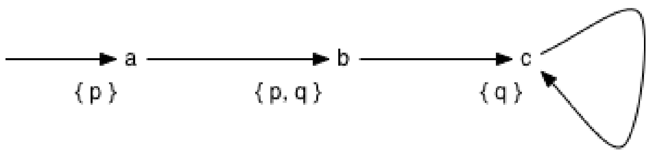


Figure 2.1: A small Kripke structure.

We do not discuss the transformation of a labeled transition system to a Kripke structure. For a discussion on that subject, see [28, 29]. However, we do wish to mention that, as will be described later on in this section, the interpretation of a symbolic transition system generates a labeled transition system with states that contain valuations for the variables of the interpreted *sts*. The Kripke labeling function L can use this valuation to determine for each state the relevant atomic propositions.

The remainder of this section gives some important definitions that are applicable to Kripke structures but also to the other two types of models. After that, we introduce symbolic and labeled transition systems, discuss the decoding of an STS model to a symbolic transition system, and the interpretation of the symbolic transition system to a labeled transition system.

States are connected to each other through transitions, which are directed edges. Transitions can connect at most two distinct states to each other. The state from which a transition departs and the state that it arrives at are its *origin* and *destination* respectively. A state can have a non-negative number of transitions that depart from it. This number is the *outdegree* of that state.

Following is the definition of a path through a Kripke structure, it has been adapted from [3].

Definition 2 (Path in a Kripke structure). *A finite path fragment $\hat{\pi}$ through a Kripke structure $K = \langle S,\ R,\ L,\ I\rangle$ is a finite state sequence $s_0, s_1, ..., s_n$ such that for all $0 \leq i < n$, $(s_i,\ s_{i+1}) \in R$. An infinite path fragment π is an infinite state sequence $s_0, s_1, ...$ such that $(s_i,\ s_{i+1}) \in R$ for all $i \geq 0$. An initial path fragment $\pi = s_0, s_1, ...$ is a path fragment with $s_0 \in I$. A maximal path fragment is said to be either an infinite path fragment, or a path fragment that ends in a state with outdegree 0, i.e. a terminal state. Finally, a path is a maximum initial path fragment.*

A Kripke structure contains a *cycle* if there is a finite path fragment $s_0, ..., s_k$ with $k > 0$ and $s_0 = s_k$. Furthermore, if $\hat{\pi} = s_0, s_1, ..., s_k$ is a finite path fragment, then s_k can be reached from all states in $\{s_0, s_1, ..., s_{k-1}\}$. More specifically, a state s is said to be *reachable* when it can be reached from a state $s_0 \in I$. For a path π we will use $|\pi|$ to denote the length of the path, with $|\pi| = n$ for some $n \in \mathbb{N}_{>0}$ if the path is of finite length, and $|\pi| = \infty$ otherwise.

2.1.1 STS to Symbolic Transition System

Symbolic transition systems were introduced by Frantzen et al. as a more powerful abstraction of labeled transition systems [28]. The labeled transition system (def. 4) is extended with data variables, and data-dependent control flow in the form of guards that can be placed on transitions.

Definition 3 (Symbolic Transition System). *A Symbolic Transition System is a tuple $\langle L, l_0, \mathcal{V}, \iota, \mathcal{I}, \Lambda, \rightarrow\rangle$, where: [28]*

- *L is a countable set of locations and $l_0 \in L$ is the initial location.*
- *$\mathcal{V}$ is a countable set of location variables.*
- *$\iota \in \mathfrak{T}(\varnothing)^{\mathcal{V}}$ is an initialization of the location variables.*
- *$\mathcal{I}$ is a set of interaction variables, disjoint from $\mathcal{V}$.*
- *Λ is a finite set of gates. The unobservable gate is denoted τ ($\tau \notin \Lambda$); we write Λ_τ for $\Lambda \cup \{\tau\}$. The arity of a gate $\lambda \in \Lambda_\tau$, denoted* **arity**(λ)*, is a natural number. The type of a gate $\lambda \in \Lambda_\tau$, denoted* **type**(λ)*, is a tuple of length* **arity**(λ) *of distinct interaction variables. We fix* **arity**$(\tau) = 0$*, i.e. the unobservable gate has no interaction variables.*
- *$\rightarrow\ \subseteq L \times \Lambda_\tau \times \mathfrak{F}(\mathcal{V} \cup \mathcal{I}) \times \mathfrak{T}(\mathcal{V} \cup \mathcal{I})^{\mathcal{V}} \times L$ is the switch relation. We write $l \xrightarrow{\lambda,\varphi,\rho} l'$ instead of $(l,\ \lambda,\ \varphi,\ \rho,\ l') \in \rightarrow$, where φ is referred to as the constraint (acting as guard) and ρ as the update mapping. We require* $\mathbf{free}(\varphi) \cup \mathbf{var}(\rho) \subseteq \mathcal{V} \cup \mathbf{type}(\lambda)$.

For a set X of variables, the set of terms constructed from the elements in X is denoted $\mathfrak{T}(X)$. The elements $t \in \mathfrak{T}(\varnothing)$ are *ground terms*. For sets of variables X, Y $\mathfrak{T}(Y)^X$ denotes the set of term mappings where each term mapping $\rho \in \mathfrak{T}(Y)^X$ assigns to every element $x \in X$ a term $t \in \mathfrak{T}(Y)$, and for every $x \notin X$ the term x. Here, $\mathfrak{T}(\varnothing)^{\mathcal{V}}$ is the set of all term mappings that each assign to all elements in $\mathcal{V}$ a ground term, and $\mathfrak{T}(\mathcal{V} \cup \mathcal{I})^{\mathcal{V}}$ the set of all term mappings that each assign to all elements in $\mathcal{V}$ a term constructed from the variables in $\mathcal{V} \cup \mathcal{I}$. Furthermore, $\mathfrak{F}(\mathcal{V} \cup \mathcal{I})$ is the set of all first order formulas constructed from the location and interaction variables. Finally, $\mathbf{var}(\varphi)$ and $\mathbf{free}(\varphi)$ denote for a first order formula φ the set of variables and the set of free variables that appear in φ respectively.

The previous definition of paths (def. 2) was given in the context of a Kripke structure. This definition can be applied to Symbolic Transition Systems if we make the following adaptations. For a symbolic transition system, a possibly infinite path fragment $\pi = l_0, l_1, ...$ consists of a sequence of locations where for every non-terminal location l_i in such a sequence, there exists a switch $l_i \xrightarrow{\lambda,\varphi,\rho} l_{i+1}$ for which the finite path fragment $l_0, ..., l_i$ enables a valuation of the values in $\mathcal{V} \cup \mathcal{I}$ to satisfy φ. To clarify that last point: if none of the path fragments that reach l_i can assign to the variables in $\mathcal{V} \cup \mathcal{I}$ the values that satisfy φ, then the switch relation can never be taken and paths that contain the locations l_i and l_{i+1} consecutively can therefor not exist.

Informally speaking, a *closed system* is a system whose behavior is completely determined by the state of the system [36]. If a closed system interacts with another system, then this external system cannot influence the behavior of the closed system: all decisions that influence the behavior are made by the closed process and not by the environment. A definition that is more strict and more relevant for this thesis, is that a closed system does not communicate with anything that is not contained within the model.

The distinction between an open and a closed system is an important one; most publicly available model checkers can only model check closed systems. Figure 2.2 shows a closed model that consists of two separate transition systems, a coffee/tea dispenser and the user. The drink dispenser first waits for a user to insert a coin and then, depending on the following decision made by the user, prepares and dispenses either coffee or tea. This system is closed because there is no pathway through the model in which something that is external to either the dispenser or the user, could communicate with one of the two *processes*. If one of these processes were to be omitted, then the model would be open. An open model can be closed by *internalizing* the relevant aspects of its environment, as has been done by adding the user process in this specific case.

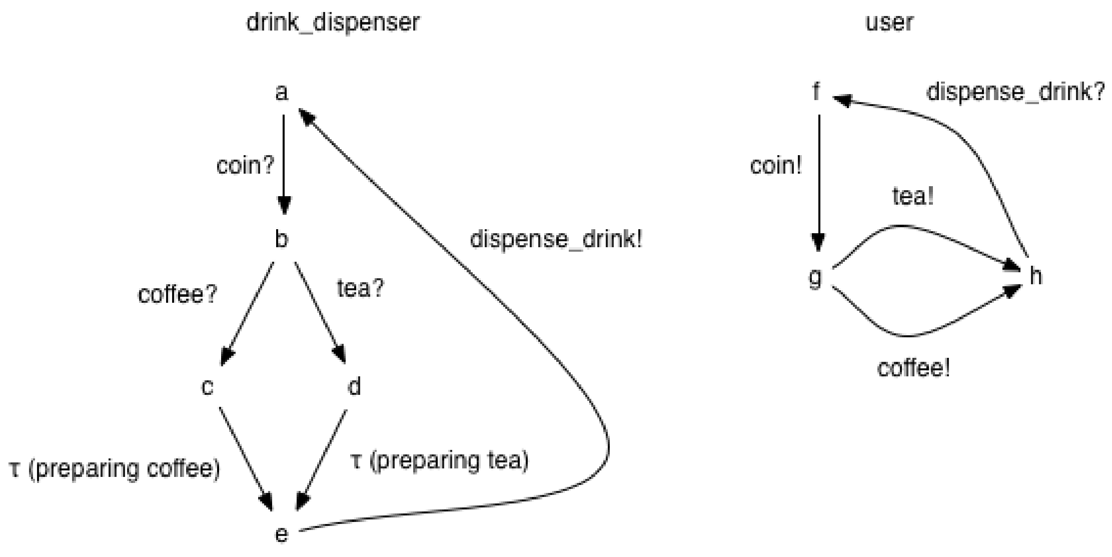


Figure 2.2: A closed model for a coffee/tea dispenser.

The decoding of an STS model to its corresponding symbolic transition system is mostly straightforward. The following list shows how the important STS structures map to their symbolic transition system counterparts.

- An STS model translates to a set of symbolic transition systems where each `sts` construct encodes a single transition system. As individual systems act as processes, they are referred to as such during the remainder of this thesis.
- The set of variables $\mathcal{V}$ is encoded by the `var` keyword.
- The set of gates Λ are defined by the `response` and `stimulus` keywords. To differentiate between the gates of a process that transmits message `a` and those of a process that receives it, the gates are renamed to `a!` and `a?` respectively. If the messages are valued, then the interaction variables are placed into $\mathcal{I}$.
- The initial location l_0 is the origin of the transition that is encoded by the first behavioral expression that follows the `stimulus` and `response` declarations. This location is named if the `state` keyword appears on the line preceding the behavioral expression. The location elements are referred to as states within the remaining chapters of this thesis.
- The elements of the set of locations L are automatically generated to connect the behavioral expressions. Each unnamed location is identified by a string that is generated from its sequence number. If the location is the nth unnamed location that is generated, then its identifier is `_n`.
- A `constraint` expression with the first order logic formula $\varphi \in \mathfrak{F}(\mathcal{V})$ that connects the locations l and l' is decoded to $l \xrightarrow{\tau,\varphi,\rho} l'$, where ρ is a term mapping that maps every location variable $v \in \mathcal{V}$ to the term v.

Listing 2.1: STS model that models a simple slot-machine

```
external 'a_channel'

sts('process_a') {
  var 'count', :integer, 0

  channel('a_channel') {
    stimulus 'coin'
    respone 'tray', {'_amount' => :integer}
  }

  state 'start'
    receive 'coin', { :update => 'count = count + 1;' }
    choice {
      o { send 'tray', :constraint => '_amount == count',
                       :update => 'count = 0;' }
      o { send 'tray', :constraint => '_amount == 0' }
    }
    goto 'start'
}
```

- An `update` expression contains a non-empty set of term mappings $\rho \in \mathfrak{T}(\mathcal{V})^X$ for a set of variables $X \subseteq \mathcal{V}$. If the expression connects the two locations l and l', then it is decoded to $l \xrightarrow{\tau,\top,\rho} l'$, where $\top$ denotes a tautology.
- Finally, the `receive` and `send` constructs map to the transition $l \xrightarrow{\lambda,\varphi,\rho} l'$ where $\lambda \in \Lambda$ is an observable gate (i.e. $\lambda \neq \tau$), $\varphi \in \mathfrak{F}(\mathcal{V} \cup \mathcal{I})$ is the guard, and $\rho \in \mathfrak{T}(\mathcal{V} \cup \mathcal{I})^{\mathcal{V}}$ is the update statement. Furthermore, for every interaction variable $i \in \textbf{type}(\lambda)$, $i \in \textbf{var}(\varphi)$, that is: the guard constraints the value of every interaction variable.

A simple example of an STS model is shown in listing 2.1. It models a slot-machine that can communicate with a user through two gates. The user inserts a coin by sending `coin`, and then the machine signals the outcome of its internal decision through the `tray` response. Its *sts* representation is ⟨{`start`, `_0`, `_1`, `_2`}, `start`, {`count`}, {`count` ↦ 0}, {`_amount`}, {`coin`, `tray`}, →⟩ where the → relation is shown in figure 2.3. In the strict sense, this system is open because it communicates with an entity outside the model, but according to the more general definition, this model is closed because an external agent cannot alter the behavior of this model; only the jackpot decides whether it moves all of its stored coins to the tray.

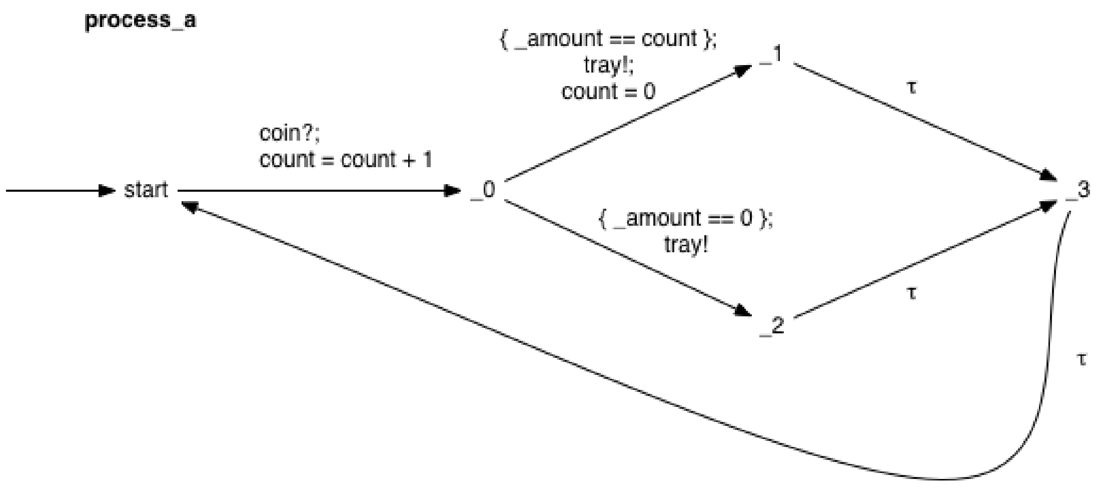


Figure 2.3: State graph for the model in listing 2.1.

2.1.2 Symbolic Transition System to Labeled Transition System

The conversion of a symbolic transition system to a labeled transition system is called an *interpretation* of the symbolic system. We use the definitions from the paper by Frantzen et al. [28].

Definition 4 (Labeled Transition System)**.** *A Labeled Transition System is a tuple $\mathcal{L} = \langle S, s_0, \Sigma, \rightarrow\rangle$, where:*

- *S is a (possibly infinite) set of states.*
- *$s_0 \in S$ is the initial state.*
- *Σ is a possibly infinite set of action labels. The special action $\tau \notin \Sigma$ denotes an unobservable action. In contrast, all other actions are observable. We write Σ_τ to denote the set $\Sigma \cup \{\tau\}$.*
- *$\rightarrow \subseteq S \times \Sigma_\tau \times S$ is the transition relation. When $(s, \mu, s') \in \rightarrow$ we write $s \xrightarrow{\mu} s'$.*

Definition 5 (STS Interpretation)**.** *Let $\mathcal{S} = \langle L, l, \mathcal{V}, \iota, \mathcal{I}, \Lambda, \rightarrow\rangle$ be an STS. The interpretation of $\mathcal{S}$ is given by the LTS $[\![\mathcal{S}]\!] = \langle S,\ s_0,\ \Sigma,\ \rightarrow\rangle$, where:*

- *$S = L \times \mathfrak{U}^{\mathcal{V}}$ is the set of states.*
- *$s_0 = (l_0,\ \textbf{eval} \circ \iota) \in S$ is the initial state.*
- *$\Sigma = \bigcup_{\lambda \in \Lambda_\tau} (\{\lambda\} \times \mathfrak{U}^{\textbf{arity}(\lambda)})$ is the set of actions.*
- *$\rightarrow \subseteq S \times \Sigma \times S$ is the transition relation.*

The transition relation $\rightarrow$ in $[\![\mathcal{S}]\!]$ is generated according to a specific deduction rule. For this rule, see the paper by Frantzen et al. [28]. To demonstrate how the interpretation of an STS could result in an LTS with an infinite number of states, we use the slot-machine example that was introduced in section 2.1.1.

Within an interpretation of an STS, a state is a pairing of a location with a valuation for all the variables in $\mathcal{V}$. This valuation needs to have been possible for that location. The slot-machine model only has a single variable in $\mathcal{V}$, `count`, and due to the nature of the model, `count` can store all possible non-negative integers. Therefor, the general form of a state $s \in S$ in $[\![\mathcal{S}]\!]$ is $(l,\ n)$ where $l \in L$ and $n \in \mathbb{N}$. The set of actions Σ consists of the elements (τ), (`coin`) and (`tray`, n) for all $n \in \mathbb{N}_{>0}$.

The initial section of the state graph of this model is shown in figure 2.4. As can be seen, $[\![\mathcal{S}]\!]$ has an infinite number of states because the `coin?` transition can always be taken from any state with location `start` and this transition will reach a state with a value for `count` that will be higher than it was in all previous states.

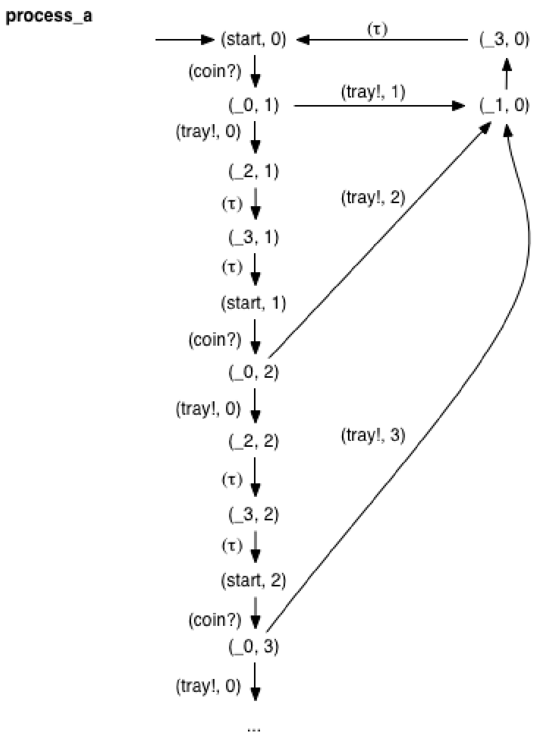


Figure 2.4: State graph for the LTS model $[\![\mathcal{S}]\!]$.

2.2 Temporal logic

Verifiable properties for Kripke structures are often expressed within a temporal logic. Such logics enable the specification of formulas over the set of atomic propositions AP and can be used to express properties over the reachable states and over the traversable paths that exist within a Kripke structure. They are called *temporal* because they incorporate a concept of time; as the traversal of a path can be interpreted as the passing of time, states along such a path can signify the way in which a system changes as time progresses. A proposition that is expressed in a temporal logic can therefor state something about the future of a system, in addition to stating propositions about individual states.

There are two important distinct ways to look at time: there is either only one possible future, or there can be multiple distinct futures from any given state [27]. These are often referred to as *linear* and *branching time* respectively. In his paper *"Sometime" is Sometimes "Not Never"*, Lamport has tried to show that these two paradigms give rise to two fundamentally different expressive powers by using a small temporal logic to define a certain proposition for a specific model and subsequently showing that depending on the perspective, the proposition will either fail or not for that specific model [42].

This troublesome proposition declared that a certain atomic predicate $p \in AP$ is not always not false, or $\varphi = \neg\Box\neg p$. A linear interpretation of this proposition states that for every possible future there is a state at which p will be not be false, denoted in Lamport's logic as $\rightsquigarrow p$ or *"p will eventually become true"*. However, from a branching perspective there are two separate circumstances in which this property holds: *a*) either p will eventually become true in all possible paths, or *b*) there exists at least one path in which p will become true. For a model of the latter type that additionally has a single path in which p won't become true, φ will fail from a linear perspective, but not from a branching perspective. Conclusively, the expressive powers of linear and branching time are different.

It is now commonly suggested that linear logic and branching logic refer to different elements of a Kripke structure; propositions of the former can be used to express properties about paths, while those of the latter are for expressing properties about individual states. In a response to Lamport's paper, Emerson and Halpern defined the temporal logic CTL* which subsumes both linear and branching logics [27, 18]. Within CTL* there are *state formulas* and *path formulas* and those of the former type can be used to express branching properties, e.g. that from the perspective of a specific state, there is a future in which something happens, and an alternative future in which it doesn't.

In the remainder of this section, we focus on the linear temporal logic LTL. This logic was initially proposed by Pnuelli in [46]. The reason why we focus on LTL and not on e.g. CTL, is because LTL is more expressive than CTL in the sense that its semantics are more intuitive and its formulas more easy to specify [58]. Furthermore, the selected model checker LTSmin seems to have a stronger support for properties that are expressed in LTL and less for those expressed in CTL. In the next section we show how LTL can be used to express some important properties. The following definition has been adapted from previous work by Vardi [57].

Before we define LTL, we first introduce the following notation. For a path π (def. 2) within a Kripke structure $K = \langle S,\ R,\ L,\ I\rangle$, π_i with $0 \leq i < |\pi|$ indicates the ith state along the path with π_0 always being an initial state i.e. $\pi_0 \in I$. Consequently, $L(\pi_i)$ denotes the set of atomic propositions that are true at the ith state of the path.

Definition 6 (LTL formula)**.** *A linear time propositional logic formula is built from the set AP of atomic propositions and is interpreted over a path π. The set of LTL formulas that holds for π and its states is generated according to the following rules:*

- $\pi_i \vDash p$ *for* $p \in AP$ *iff* $p \in L(\pi_i)$.
- $\pi_i \vDash \xi \wedge \phi$ *iff* $\pi_i \vDash \xi$ *and* $\pi_i \vDash \phi$.
- $\pi_i \vDash \xi \vee \phi$ *iff* $\pi_i \vDash \xi$ *or* $\pi_i \vDash \phi$.
- $\pi_i \vDash \neg\varphi$ *iff not* $\pi_i \vDash \varphi$.
- $\pi_i \vDash X\varphi$ *iff* $\pi_{i+1} \vDash \varphi$.
- $\pi_i \vDash \xi U \phi$ *iff for some* $i \leq j < |\pi|$, $\pi_j \vDash \phi$ *and for all* k, $i \leq k < j$, *we have* $\pi_k \vDash \xi$.
- $\pi_i \vDash \xi R \phi$ *iff* $\neg(\neg\xi U \neg\phi)$ *[19].*

- $\pi_i \vDash F\varphi$ *iff* $\top U\varphi$, *where* $\top$ *denotes a tautology. This is the eventuality: "*φ *will eventually hold."*
- $\pi_i \vDash G\varphi$ *iff* $\neg F\neg\varphi$, *or "*φ *will hold henceforth".*
- $\pi \vDash \varphi$ *iff* $\pi_0 \vDash \varphi$.

2.3 Properties

Formulas can be used to express interesting properties of models. There are two important properties that can be expressed with LTL:

- *Safety properties*, *"no bad things will happen"*. There are two important subcategories: *invariants* and *freedom from deadlocks* [38]. A property is invariant if it holds for every reachable state and a model is free from deadlocks if every reachable state has at least one successor. *Mutual exclusion* is a common example of an invariant and can be expressed as $G\neg(p \wedge q)$ if p were to mean that some process A is in its critical region, and q that some other process B is in his. In general, a safety property is expressed in LTL as $G\varphi$ if φ represents for a single state the absence of a "bad" property. Deadlock-freedom is normally not expressed in LTL, and model checkers can use specialized algorithms to detect deadlocks. However, $GX\top$ could be a possible formula to express this, where $\top$ would denote a tautology.
- *Liveness properties*, *"something good will always eventually happen"*. These properties are different from safety properties because they are not verified over individual states, but over all possible futures that are reachable from such states. Liveness properties can be used to express various forms of *fairness* (e.g. an inactive process cannot remain inactive forever) and *livelock-freedom* (i.e. infinite runs must have progress infinitely often [38]). These formulas are often expressed in LTL with the general form $GF\varphi$, where φ is a property that denotes progress. For example, $G(\textbf{request} \rightarrow F\textbf{response})$ could state that whenever a system receives a request, it will always eventually emit a response.

This categorization is useful because it outlines the different approaches that are required to verify these properties [3, 39]. For example, to verify a safety property, it is sufficient to examine each reachable state individually. However, to verify a liveness property, it it sometimes necessary to examine all states that are reachable from that state. For a labeled transition system with an infinite number of states, this verification might be undecidable. See chapter 4 for a more in-depth discussion about verifying these properties.

There are two types of deadlocks that are particularly relevant to this thesis. They are *explicit* and *implicit deadlocks* and refer to those that are part of the *symbolic transition system* and those that emerge during the interpretation of such a symbolic transition system respectively. A symbolic transition system $\mathcal{S} = \langle L, l_0, \mathcal{V}, \iota, \mathcal{I}, \Lambda, \rightarrow, \rangle$ has an explicit deadlock when there is a location $l \in L$ with an outdegree of 0. It has an implicit deadlock if the interpretation $[\![\mathcal{S}]\!] = \langle S, s_0, \Sigma, \rightarrow\rangle$ contains a state $s \in S$ with $s = \langle l, \mathfrak{u}_1, \mathfrak{u}_2, ...\mathfrak{u}_n\rangle$ that has an outdegree of 0, but the corresponding location $l \in L$ has an outdegree that is non-zero.

2.4 Examples

The Kripke structure shown in figure 2.1 contains three states $\{a, b, c\}$ and a single infinite path $\pi = a, b, c, c, \ldots$. If the absence of both p and q would be a "bad" property, then $G\neg(\neg p \wedge \neg q)$, or rather: $G(p \vee q)$, would be the corresponding safety property and $\pi \vDash G(p \vee q)$ because all three reachable states have either p, or q, or both. However, if $p \wedge q$ would indicate that two processes A and B are in their critical region, then $G\neg(p \wedge q)$ would express the property of mutual exclusion. In that case, $\pi \nvDash G\neg(p \wedge q)$ because $b \vDash p \wedge q$.

Another interesting Kripke structure is shown in figure 2.5. This structure has two paths $\pi' = a, b, b, \ldots$ and $\pi'' = a, c, c, \ldots$ which agree on some properties, but not on others. For example, they both agree on $G(p \vee q)$, but not on Gp or Gq. However, even though p is true for all states that are on path π', it is important to mention that $\pi' \nvDash Gp$. This is because from a linear perspective, there is only one future, and so all futures that are possible from, e.g. the state a, should agree on Gp, which they don't. The final interesting Kripke structure is a finite one and is shown in figure 2.6. Its only path $\pi = a, b, c$ halts in a terminal state, i.e. c has an outdegree of 0. This would mean that this model has a deadlock, and so $\pi \nvDash GX\top$.

Furthermore, for all three models, we can define the eventuality Fq because either q is true in the initial state, or q will become true in all future paths that are reachable from that initial state. However, while the models in figures 2.1 and 2.6 agree on GFq, the one in figure 2.5 does not because for state b in that model, $b \nvDash Fp$.

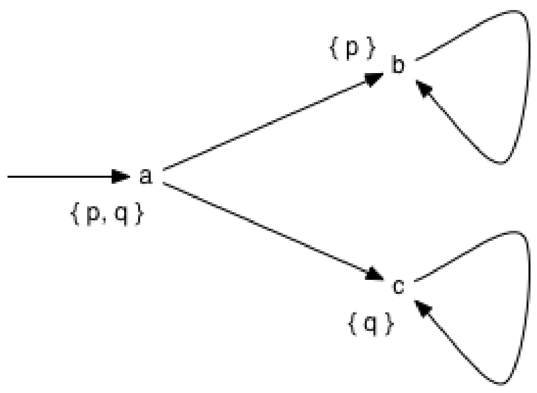


Figure 2.5: A Kripke structure with two separate paths

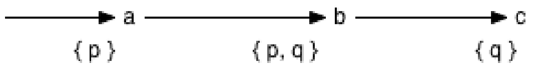


Figure 2.6: A Kripke structure with a single finite path

Chapter 3

Axini's TestManager

TestManager is the name of a testing framework that is used to verify the public behavior of a *system under test* according to its behavioral requirements. To allow TestManager to recognize incorrect behavior, a user supplies an `STS` model that defines the correct behavior. During the test, TestManager interacts with the SUT by sending it messages and interpreting the responses. If the received response was not conform to the `STS` model then TestManager may conclude that the implemented system is in error.

Figure 3.1 shows the relation between the SUT, TestManager and the `STS` model. The `Adapter` is a component that converts the actual communication of the SUT to and from the abstract behavioral *traces* that are specified within the `STS` model.

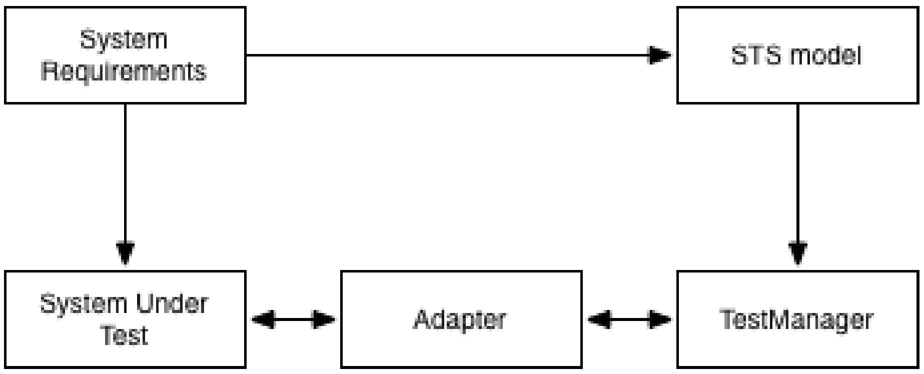


Figure 3.1: An overview of the relation between the SUT, TestManager and the `STS` model

The `STS` model does not need to be a representation of how the SUT is implemented; the model only needs to be a formalization of the correct behavior. It also does not need to account for all the behavior that is specified by the requirements, or emitted by the implemented system; the behavior that is generated by the `STS` model can be an *underspecification* of the requirements. This can be done to focus testing on a specific subset of the required behavior.

TestManager communicates with the SUT through messages that are specified within the `STS` model as those that are transmitted over *external channels*. Models whose behavior depend on the messages they receive over external channels are referred to as *open* or *reactive models*.

The next section of this chapter contains a brief description of TestManager's algorithm. It is important to mention that TestManager only tests the system to a certain degree. It cannot test all behavior of systems that are of non-trivial complexity. Section 3.2 contains a more in-depth discussion of this issue.

3.1 TestManager algorithm

Underneath an `STS` model is a finite directed graph that represents a *symbolic transition system*. This *sts* is assumed to have multiple states and multiple transitions linking them. Furthermore, it is assumed that there is a single initial state and that each non-initial state within the *sts* is, either directly or indirectly, reachable from that initial state. It is also assumed that the *sts* contains at least a single state with an outdegree > 1. Models without such states certainly are valid, but they are not very interesting to test because they only have a single path through the graph, and thus only a single behavioral sequence.

TestManager starts from the initial state and explores the graph while interacting with the SUT. During this exploration, TestManager maintains a set of states. The future behavior of the SUT is predicted by a path that leaves one of these states. Normally, TestManager only keeps track of a single state and by examining the transitions that lead away from this single state, it can determine whether it has to wait for a response from the SUT, or send it a message. However, when there are two diverging transitions that lead away from this single state and both specify the same interaction, then after this interaction has taken place, TestManager must expect either of both possible futures. Listing 3.1 shows an STS fragment in which the behavioral sequence $[\alpha!, \beta?]$ can be followed by either $\gamma!$ or $\epsilon!$. It won't know what the SUT will actually send, as this is apparently the outcome of an internal decision made by the SUT to which TestManager is not privy.

Listing 3.1: STS fragment that accounts for an internal decision made by the SUT.

```
send 'α'
choice {
  o { receive 'β'
      send 'γ'
      receive 'δ' }
  o { receive 'β'
      send 'ϵ'
      receive 'ζ' }
  o { receive 'η' }
}
```

During the exploration of the graph, TestManager interacts with the SUT by either sending it a message when it knows that the SUT expects one, or expecting a message from the SUT when it knows that the SUT is no longer waiting for input but is in the process of generating a response. These interactive cues are determined according to the STS model. When TestManager receives a response from the SUT, it will try to find all transitions that leave from a state within the set that match the response. All members of the set are then replaced by those states that the predicting transitions led to. If there were no transitions that matched the response, then this set becomes empty and TestManager must signal an error.

When a transition that is leaving a state in the set signifies that the SUT is waiting for a response, then TestManager is inclined to explore the graph along that transition. If there are multiple of such transitions, then TestManager has to try a single one. If the SUT is waiting for a valued message, then TestManager has to determine which value to generate. If there is a receive transition that places a specific constraint on this value and TestManager has decided that it wants to follow this transition, then it must use a *constraint solver* to find a value that matches the imposed constraint. If the SUT rejects the message sent by TestManager then the states from which the transitions depart that specify the message must be removed from the set of states. Apparently, such states did not predict the future behavior of the SUT. Again, if the set becomes empty, then TestManager must conclude that the SUT is in error.

The constraint solver used by TestManager is a finite domain solver implemented in GNU Prolog[1]. The constraint expressions on the guarded receive transitions are transformed to binary trees and then passed on to Axini's TreeSolver, which uses the GNU Prolog constraint solver to determine values for the open variables so that the expression becomes true. If TestManager has determined that an STS model can traverse a guarded receive transition from its current state, then the model will take that transition when TestManager sends the SUT a valued message of which the values satisfy the constraint on that guarded transition and when the SUT accepted the message.

For example, if a receive transition states that the SUT is waiting for the message `a` with the constraint "`_value1 + _value2 > 10 && _value1 >= 5 && _value2 >= 5`", then TestManager converts this expression to the tree shown in figure 3.2 and passes this tree to the TreeSolver. TreeSolver will then use the finite domain solver that is implemented in GNU Prolog to calculate a solution for the open variables `_value1` and `_value2`, e.g. the values 5 and 6 respectively. TestManager will then generate a message `a` with those values and send it to the SUT. If the SUT accepts this message, then this transition is traversed.

[1]http://www.gprolog.org/

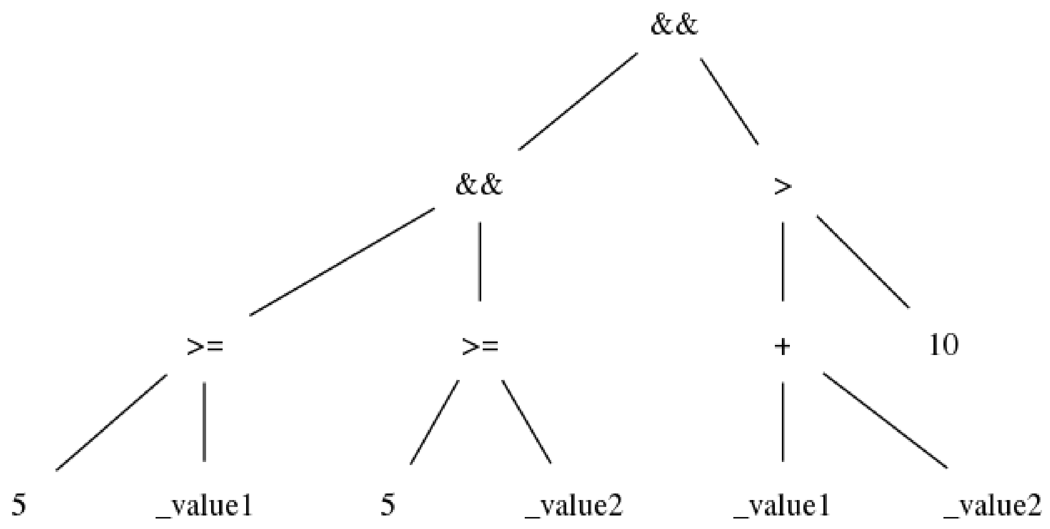


Figure 3.2: Binary tree generated for a constraint expression

Listing 3.2 shows the relevant pseudocode for TESTMANAGER algorithm when given an $STS = \langle L, l_0, \mathcal{V}, \iota, \mathcal{I}, \Lambda, \rightarrow \rangle$ (def. 3). This version has been simplified in a number of ways and the most important simplifications are that this version ignores τ transitions and both state and message variables. It is only designed to give an indication of how the set of states is maintained during an exploration and how a fault is eventually detected.

3.2 Effectiveness

The main issue that limits the effectiveness of TESTMANAGER's approach to software testing is the restricted coverage. During each iteration, in which TESTMANAGER tests a single behavioral trace and finally resets the SUT to an initial state, a single path of the STS model is tested. As the number of distinct paths grows exponentially to the size of the model, the time needed to completely test the SUT grows exponentially as well, as each of these paths need to be checked individually. Consequentially, non-trivial systems cannot be completely tested within a reasonable time frame.

If TESTMANAGER wasn't capable of finding a bug within the available time, then TESTMANAGER may only conclude that if there was a bug, it doesn't visibly affect the tested behavior. To ensure that as much different behavioral aspects are tested, and so to maximize the chances of finding a bug, TESTMANAGER tries to spread the testing over divergent sections of the model.

Nevertheless, model checking can not directly help with this limitation as model checking will only be applied to the STS model, and not to the SUT. However, model checking can be used to reduce the effort that is needed to develop and maintain the models, freeing more resources for software testing.

Listing 3.2: Pseudocode for a simplified version of `TestManager`

```
states = { $s_0$ }

while True
  # cannot predict future behavior of the SUT.
  return False if $\mathsf{states} == \varnothing$

  # if the STS can transmit a message
  if $\exists s \in \mathsf{states}, s' \in L, \lambda \in \Lambda :\ s \xrightarrow{\lambda!} s'$
    # show initiative and send a message.
    m = A message $\lambda \in \Lambda$ so that $\exists s \in \mathsf{states}, s' \in L :\ s \xrightarrow{\lambda!} s'$

    send_message(m)
    if SUT rejected m
      # remove all states from which m could have been sent
      states = states - { $s \in \mathsf{states} \mid \exists s' \in L :\ s \xrightarrow{\mathrm{m}!} s'$ }
    else
      states = { $s' \in L \mid \exists s \in \mathsf{states} :\ s \xrightarrow{\mathrm{m}!} s'$ }
    end
  elsif $\exists s \in \mathsf{states}, s' \in L, \lambda \in \Lambda :\ s \xrightarrow{\lambda?} s'$
    m = A message $\lambda \in \Lambda$ that is sent by the SUT
    states = { $s' \in L \mid \exists s \in \mathsf{states} :\ s \xrightarrow{\mathrm{m}?} s'$ }
  else
    # no communication possible. We're done.
    return True
  end
end
```

Chapter 4

Model checking

This chapter contains an overview of the tasks that a model checker performs when it is verifying a property over a model. A strong focus has been placed on the model checker LTSmin, as this is the most relevant model checker for this project.

Normally, a model checker takes a closed model, which is expressed in some modelling language such as Promela, converts it to its state space, which can be represented by a Kripke structure, and then tries to verify the state space with respect to some desired property. The verification process is often implemented as a graph traversal through the structure that starts at the initial state, and inspects every state until either all states have been inspected, or a violation to the property has been found. If the model checker has detected a violation, then it presents the user with a *counterexample*, which is a finite path through the structure that starts at the initial state and ends at the state at which the violation was detected. If the model checker has not detected any violation of the property, then the property holds within the checked model. Due to the formal nature of the model checker, and assuming that the model checker itself doesn't contain any bugs, this result is conclusive: if the model checker didn't find any violation, then the model doesn't contain a violation.

Most of the properties verifiable that can be verified within such a structure can be expressed in a *temporal logic* such as LTL. To verify such a property, a model checker often contains a verification algorithm that uses a Büchi automaton to find a property violation. This approach is used by LTSmin and is discussed in section 4.2.1. Not all properties need to be expressed in LTL in order for them to be verified. Deadlocks, for example, can be easily found without using such an automaton and this approach is also discussed later on in this chapter.

To check a model, the model checker has to perform the following tasks:

- *State space generation*, which is the act of converting a model, that is expressed in a higher formalism, to a state space.
- *Property verification*, the act of checking whether the property holds within the generated state space.

These two tasks do not need to be executed in sequence. Most industrial model checkers can execute them simultaneously, that is: they can verify the properties while they generate the state space. This approach is commonly known as *on-the-fly* verification and has many benefits over the sequential approach. Section 4.2 will focus on on-the-fly verification, as this is one of the most important features within LTSmin. This discussion will be preceded by an overview of several important state space generation techniques, which is given in section 4.1.

4.1 State space generation

State space generation, or *reachability analysis*, is the task of determining which distinct states are reachable from the initial state of the model. This task is necessary, because when the model checker is given a model that is expressed in a higher modelling language, it has no direct knowledge of all states that are reachable. It must determine the set of reachable states by successively following untraversed transitions in order to find new unexplored states.

For some models, this state space may be infinitely large. Figure 2.4 shows such a model. The task of generating the entire state space for such a model will never halt, and if the verification is not done on-the-fly but sequentially, then no part of the model will ever be verified. For this section, we will assume that the state space is only finitely large.

The generated state space is often simply a collection of the reachable states. After the transitions have been traversed, they are no longer needed and can be omitted. The result is a graph without the edges, or rather, a collection of the states. A state space that allows the verification of temporal relations between states, e.g. the verification of *liveness properties*, can be a combination of the reachable states with the Büchi automaton that represents the negation of the desired property (see section 4.2.1). The relevant temporal relations are then incorporated into the states, making the edges again redundant.

Even when the state space is finite, it might still be too large to be stored within computer memory. This is a consequence of the dreaded *state explosion*. A popular countermeasure is to store the state space *symbolically* or *implicitly* [17]. This is opposed to the *explicit* representation of states, in which each state is represented and stored individually. A symbolic representation groups states that are similar to each order, in such a fashion that they can be stored with sublinear memory requirements [13]. States within such symbolic approaches are often represented as paths within a *decision diagram*, which is a layered acyclic directed graph. This symbolic representation is the subject of discussion in section 4.1.1.

Another possible alternative that can counteract the state explosion, but also deal with an infinite state space is *bounded model checking*. This approach tries to detect countermeasures that have a certain predetermined length k or less [6]. All states that are verified by this approach, lie at a distance of $< k$ from an initial state. The states that lie beyond this boundary are not verified and so this approach will fail to detect a counterexample if its length is larger than k.

4.1.1 Decision diagrams

A *decision diagram* is a layered directed acyclic graph in which the nodes act as decision points. At each node that has an outdegree > 1, a decision is made about the transition through which the state is departed. The non-terminal nodes are each labeled with a single variable, and this decision pertains solely to the value of that variable. If there are K different variables, then the diagram will have $K + 1$ different layers where each layer will contain at least one node. The top layer normally has a single node. Each node that is located within a non-terminal layer will have ≥ 1 outgoing transitions to nodes that are placed at lower layers. We assume that all nodes that are labeled with the same variable are placed within the same layer. This type of decision diagram is an *ordered decision diagram* as every path from the initial node to a leaf will have the same order of variables.

Figure 4.1(b) shows an example of a binary decision diagram. It has four layers for the three variables $\{p, q, r\}$ and these variables are assumed to hold boolean values. The decisions are made on whether the stored value is *true* or *false*. If the variable stores the former value, then the node is departed through the right transition, if it is the latter, then the node is departed through the left transition. Figure 4.1(c) shows the minimized form of this diagram, in which all ineffective decision points have been removed.

A decision diagram can represent the reachable state space within the total state space with its paths. Each distinct path that departs from the top node and arrives at a terminal node that is labeled with the boolean value `1` represents a node within the reachable state space. Likewise, each state that is within the reachable state space is represented by a single path that terminates at a node that is labeled with the value `1`.

Using a decision diagram to represent a Kripke structure, the variables can be elements of the set of atomic proposition AP [11]. Figure 4.1(a) shows a small Kripke structure $K = \langle S, R, L, I\rangle$ over a set of atomic propositions $AP = \{p, q, r\}$ where each state $s \in S$ is conform the logic formula $p \wedge (q \vee r)$. Figure 4.1(b) shows the full binary decision diagram in which each non-terminal node has two departing transitions and is labeled with a single atomic proposition. The bottom layer contains three nodes that are labeled with `1`. The three paths that end at these nodes are $\{p, \neg q, r\}$, $\{p, q, \neg r\}$ and $\{p, q, r\}$ which together represent all the states that are shown in 4.1(a). Finally, figure 4.1(c) shows a minimized binary decision diagram in which all of the redundant nodes and transitions have been omitted. For example, all of the paths that depart the top node through the left transition end in a terminal node with the boolean value `0`. None of the nodes that follow this transition affect this outcome and can therefor be safely omitted.

A node within a decision diagram can also have an outdegree that is larger than two. The variable that labels this node is then associated with a value domain that holds more than two values. Such diagrams are called a *multi-valued decision diagrams* and are often smaller than the

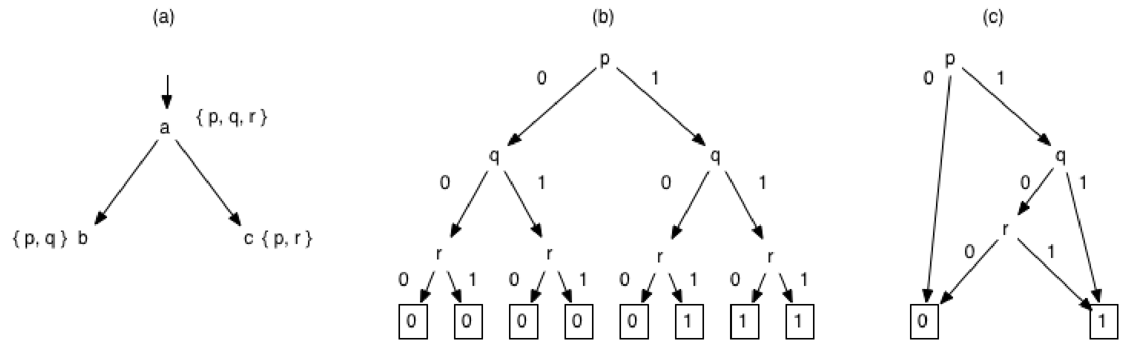


Figure 4.1: A Kripke structure (a) along with a full BDD (b) and a minimized BDD (c).

corresponding binary diagrams [14].

There are two important issues that have to be considered when decision diagrams are used to represent the reachable state space. These issues could have such a detrimental effect on the generation process, that it sometimes could be better to use an explicit representation instead [40].

The first issue pertains to the order in which the variables appear along paths in the diagram. This order is important because it determines the size of the resulting diagram. For example, the minimized diagram that is shown in 4.1(c) has the order p, q, r. If this order was q, p, r or r, p, q then the resulting diagram would have four states instead of three. For models that are larger and more complex, the difference between the minimal BDD and the largest could be drastic.

The other important issue is the *peak size* of the diagram during its generation. Most of the diagram-generating algorithms maintain the diagram while exploring the state space. New nodes and edges are added as new states are discovered. At some points during this process, certain segments can be recognized as redundant, and so the maintained diagram can be reduced to a smaller form. However, the order, or *strategy*, in which the state space is explored may determine how often such moments appear, and with the wrong strategy, the intermediate diagram could become too large to be stored within memory. There are several strategies, such as *saturation*, that are designed to reduce this effect [14, 13].

LTSmin supports both explicit state space representation, binary decision diagrams and multi-valued decision diagrams. It also uses the saturation strategy to keep the diagram size as small as possible during symbolic state space generation [48, 56].

4.2 Property verification

The verification of a property is normally done over the state space of a model. Originally, the state space was generated in its entirety before verification took place, but a more modern and popular approach is to perform both generation and verification simultaneously. The main benefits of such an *on-the-fly* approach are that counterexamples can be detected in less time and with smaller memory requirements.

Within LTSmin, on-the-fly verification is done with a successor generator. This is a function that takes a single state and then returns all its successor states. LTSmin handles models that are written in DVE or Promela by converting them to successor generator functions. Such a function generates with each execution only a small portion of the state space. The explored state space has to be maintained to ensure that states that were already explored are recognized as such so that the process will eventually terminate. These already-discovered states are stored either explicitly or symbolically.

As was mentioned in section 2.3, most verifiable properties are divided into two classes and this categorization reflects how these properties can be verified. Informally, verifying safety properties is about finding “bad” states and verifying liveness properties is about finding “bad” cycles. If no such “bad” things are found, then the property holds within the checked model.

Safety properties are verified by examining each state individually after it has been generated. A deadlock has been detected when the successor generating function returns an empty set. Other safety properties, that are possibly expressed in LTL, can be verified by obtaining the atomic propositions that hold for that state and evaluate the safety property in relation to those propositions.

It is more difficult to verify liveness properties and an popular approach to the verification of such properties will be discussed within the remainder of this chapter.

4.2.1 Büchi automata

A popular approach to the verification of liveness properties is to use a Büchi automaton, which is an *automata-theoretic* approach that was first proposed by Vardi [57, 58] and has been implemented in the LTSmin and SPIN model checkers [39, 31]. A Büchi automaton is a non-deterministic finite automaton that consists of several states of which some are *designated.* The LTL formula that represents the desired property is negated, and converted to such an automaton. The result of this conversion is combined with the model to produce a new Büchi automaton. The remaining task for the model checker is to solve the *emptiness problem*, which is the determination of whether there is an *accepting cycle* within this new automaton. If the model does contain such a cycle, then there is at least a single infinite path through the model for which the liveness property does not hold. Conclusively, the property does not hold for the entire model.

The following definitions have been adapted from [21, 30].

Definition 7 (Büchi automaton)**.** *A Büchi automaton is a tuple* $\mathcal{A} = \langle \Gamma, \rho, \Gamma_0, F \rangle$*, where:*

- Γ *is a set of states,*
- $\rho \subseteq \Gamma \times \Gamma$ *is a non-deterministic transition function,*
- $\Gamma_0 \subseteq \Gamma$ *is a set of starting states, and*
- $F \subseteq \Gamma$ *is a set of designated states.*

A *Labeled Büchi Automaton* is a tuple $\mathfrak{A} = \langle \mathcal{A}, \mathcal{D}, \mathcal{L} \rangle$ where $\mathcal{A}$ is a Büchi automaton, $\mathcal{D}$ is some finite domain and $\mathcal{L} : \Gamma \to 2^{\mathcal{D}}$. An *execution* is an infinite sequence of states $\gamma = \gamma_0, \gamma_1, \ldots$ such that $\gamma_0 \in \Gamma_0$ and for each $i \geq 0$, $(\gamma_i, \gamma_{i+1}) \in \rho$. An *accepting execution* γ is an execution such that there is at least a single designated state $f \in F$ that appears infinitely often within γ. An infinite word $w \in \mathcal{D}^\omega$ for which $w = w_0, w_1, \ldots$ is *accepted* iff there is an accepting execution $\gamma = \gamma_0, \gamma_1, \ldots$ such that $w_i \in \mathcal{L}(\gamma_i)$ for all $i \geq 0$. Furthermore, $\mathcal{L}(\mathfrak{A})$ denotes the set of all words $w \in \mathcal{D}^\omega$ that are accepted by $\mathfrak{A}$.

The intersection of a Kripke structure $K = \langle S, R, L, I \rangle$ (def. 1) over AP with a labeled Büchi automaton $\mathfrak{A} = \langle \mathcal{A}, \mathcal{D}, \mathcal{L} \rangle$, denoted $K \times \mathfrak{A}$ is a Büchi automaton with:

- the domain $\mathcal{D} = 2^{AP}$,
- the state set $S \times \Gamma$,
- transition function $\tau \subseteq (S \times \Gamma) \times (S \times \Gamma)$, where $((s_1, \gamma_1), (s_2, \gamma_2)) \in \tau$ iff $(s_1, s_2) \in R$ and $(\gamma_1, \gamma_2) \in \rho$ and $L(s_2) \in \mathcal{L}(\gamma_2)$, and
- the set of designated states $S \times F$.

Given an LTL formula φ and a Kripke structure K, then the task of the model checker is to generate a labeled Büchi automata from formula $\neg\varphi$, denoted $\mathfrak{A}_{\neg\varphi}$ and see whether the intersection $K \times \mathfrak{A}_{\neg\varphi}$ accepts any words, that is: whether $\mathcal{L}(K \times \mathfrak{A}_{\neg\varphi}) \neq \emptyset$. This is the *nonemptiness problem* and a model checker can check whether this new automaton is non-empty by using an exhaustive search of the state space of this new automaton. If the model checker detects a cycle in which a designated state appears infinitely often, then $\mathcal{L}(K \times \mathfrak{A}_{\neg\varphi})$ is nonempty, and the property φ doesn't hold within K.

Figure 4.2 shows two Kripke structures (a and b) and a Büchi automaton (c). They are relevant to the liveness property $\varphi = G(\texttt{request} \to XF\texttt{response})$ of which an informal translation states that whenever a request is made, a response will appear within the strict future. This property holds for 4.2(a), but not for 4.2(b). To verify this property in both models, one converts the negation of the LTL formula φ to the Büchi automaton $\mathfrak{A}_{\neg\varphi}$. This automaton is shown in 4.2(c) and has been generated from the normalized form of the negation $\neg\varphi = \texttt{true}\ U\ (q \land X(\texttt{false}\ R\ \neg p))$ where for simplicity, `response` has been replaced with p and `request` with q.

The generation of the automaton $\mathcal{A}_{\neg\varphi}$ was done with the method that was originally proposed by Gerth et al. [30] and adapted by Clarke et al. [19]. For the representation shown in 4.2(c), the states are each labeled with the subformulas of $\neg\varphi$ that must hold at that point. Underneath the dividing lines, the representation shows the values of $\mathcal{L}(d)$, $\mathcal{L}(e)$ and $\mathcal{L}(f)$.

The method in [30] also describes how designated states are selected. Using the normalized version of $\neg\varphi$ = $\texttt{true}\ U\ (q \wedge X(\texttt{false}\ R\ \neg p))$, the designated states are determined to be those that either do not contain $\neg\varphi = F(q \wedge XG\neg p)$ or those that contain $q \wedge X(\texttt{false}\ R\ \neg p) = q \wedge XG\neg p$. Within this image, they are enclosed within boxes to differentiate them from non-designated states. The state `e` is a designated state because it contains $q \wedge XG\neg p$ and `f` because it doesn't contain $F(q \wedge XG\neg p)$.

Figure 4.3 shows two Büchi automata. These are the result of the intersections of 4.2(a) with 4.2(c) and 4.2(b) with 4.2(c) respectively. In both automata, the grayed-out states are unreachable because their components contradict each other on some of the propositions from AP. For example, `(b,e)` is unreachable because $L(b) \notin \mathcal{L}(e)$. In 4.3(a), `(a,f)` is unreachable but it does not contain such a contradiction. However, this state has become reachable in 4.3(b) and furthermore, because of the added edge between `(b',f)` and `(a',f)`, this automaton has an accepting cycle and is thus nonempty. This would indicate that the property φ does not hold for the model 4.2(b). The model checker won't find any such cycle in 4.3(a), and will therefor conclude that φ holds for 4.2(a).

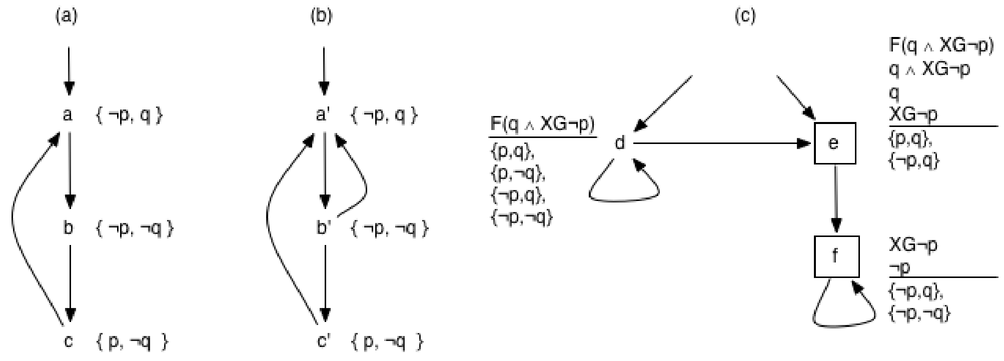


Figure 4.2: Two interesting models for a liveness property (a, b) and the Büchi automaton for that property (c).

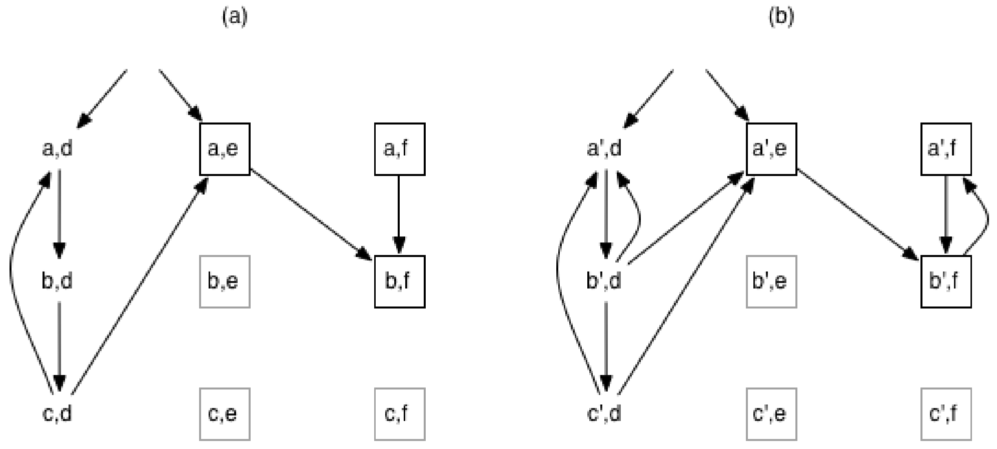


Figure 4.3: The two models from 4.2 each intersected with the Büchi automaton.

LTSmin uses an on-the-fly depth-first-search approach to solve the nonemptiness problem [38, 39]. Such an approach is not easy to parallelize, because depth first search is inherently sequential. However, the implemented algorithm has been designed for use with multiple-cores through *swarm verification*, in which multiple workers each try to verify the entire state space, but are randomly guided to different regions of the state space. Furthermore, whenever any of the workers have determined that a certain region doesn't contain any counterexamples, they communicate this finding to their peers. This decreases the chances that multiple workers are working on the the same region simultaneously, as they check whether a region has already been verified before entering it. The result is that as the number of workers grow, the entire state space will be covered in less time as each worker tries to find and verify his own region of this state space.

Chapter 5

Software Implementation

This chapter describes the software system, or *prototype*, that was developed during this project. The developed software allows Axini to *model check* some of the `STS` models that they use for the automated testing of interactive systems. When this product is incorporated into their toolbox, it will help Axini with determining whether the `STS` model itself adheres to the systems requirements. As a consequence, this will allow Axini to

- prematurely detect errors before the model is used by TestManager,
- develop complex models in less time,
- ensure that the SUT is tested more thoroughly by reducing the effort needed for developing and maintaining the `STS` models, and
- possibly test if the system requirements are consistent with each other, by checking if there is any model that is correct with respect to all these requirements.

Currently, the software system can only check for *deadlocks*. This allows it to ensure that the model will not halt indefinitely. Such a fault can occur when the model is incomplete, i.e. certain transitions are missing that are necessary for the model to continue; or when constraints within the model prohibit the model from continuing from a reachable state (i.e. an *implicit deadlock*). Figure 6.2 shows a state space that is generated from an `STS` model with a deadlock that occurs when a constraint prohibits the model from leaving a state. It is estimated that support for *liveness*, *safety* and other LTL-expressible properties can be added without significant amount of effort.

The current implementation closes the open `STS` models and translates them to the DVE language. The resulting DVE model is then inserted into the LTSmin model checker. Closing the model has proven to be a difficult task, and currently only certain simple models can be checked. Section 5.8 describes which models fall within the scope and a discussion of possible methods to increase this scope is given in chapter 7 and appendix C.

What follows in this chapter is a description of the implementation that is preceded by an enumeration of the challenges that had to be faced during the design and construction and the decisions that were made to meet or avoid these challenges. Chapter 6 lists some of the models that can be checked, along with the output of the prototype. For a lists of technical issues, one could check appendix C.

5.1 Challenges

An obvious issue that prevents `STS` models from being model checked, is that there is no available model checker that can read models that are written in the `STS` language. A possible solution is to modify an existing model checker to accept `STS` models, but there are some important issues that make this solution non-trivial.

- ***`STS` is an extension of the Ruby*[1] *language.*** Ruby class declarations, method calls, Hash operations, etc. are informally part of the `STS` language. TestManager loads the model by using the Ruby interpreter to parse the model file and interpret the `STS` specific keywords as Ruby method calls. One could restrict the `STS` language by excluding non-`STS` specific

[1]https://www.ruby-lang.org

constructs, but this hinders the adoption of model checking within the Axini workflow as many complex `STS` models contain Ruby method definitions and such constructs make it easier to manually create and modify `STS` models.

- *`STS` supports open models.* As was previously mentioned in chapter 3, the models inserted into TestManager must be open as TestManager can only communicate with the messages that are defined on the external channels. Most model checkers do not support open models. Some do but they are often too academical for practical purposes.
- *`STS` supports the specification of message transmissions with unspecified values.* `STS` models can specify the transmission of valued messages without specifying the exact value. Rather, it is possible to specify a value domain from which the values can be selected. Such value domains can be complex and consist of interactions between multiple variables. TestManager uses a constraint solver to select the appropriate values.
- *`STS` supports a wide variety of datatypes.* `STS` allows operations on floats, strings, datetime objects, arrays, sets and dictionaries while most model checkers only support integers and arrays of integers. To address this issue, it's not sufficient to map these datatypes to integers. Operations performed on variables with these datatypes must also be correctly translated.

Finally, these issues need to be resolved in such a way that they still allow for proper model checking. Resolving any of the aforementioned issues in an incorrect way, might cause the software system to claim that the model is correct with respect to certain properties when it isn't (i.e. false positives) or that the model is incorrect with respect to properties when it is (i.e. false negatives).

5.2 Architectural Decisions

To be able to produce a viable solution despite the challenges listed in the previous section, several decisions have been made that affected the design and scope of the implementation. What follows are some of the important design decisions that were made.

- *Use a publicly available model checker.* Developing a model checker takes a significant amount of theoretical understanding and effort while good model checkers are publicly available as open source software. However, such an available model checker won't support `STS` models.
- *LTSmin as model checker.* LTSmin has been selected over several other candidates, such as SPIN, DiVinE, etc. for several reasons. In short, LTSmin is generally faster, more scalable and more extensible than its competitors. These and several other reasons are discussed in section 5.5.
- *Do not extend the model checker to support `STS`.* As `STS` is an extension of the Ruby language, modifying a model checker such as LTSmin to support the Ruby language would take too much effort. LTSmin supports several model specification languages, among which ETF directly maps to the internal representation of the state space. Thus, any advantage gained by extending LTSmin with Ruby could also be gained with converting `STS` to ETF or any other language already supported by LTSmin.
- *Use DVE as input language for LTSmin.* Compared to the other languages that LTSmin accepts, it is relatively easy to automatically convert `STS` to DVE. However, `STS` has idiosyncratic features that cannot be directly translated to DVE. These features and how they were dealt with are described in section 5.7.
- *Use of Ruby as development language.* All important code that was developed during this product is written in the Ruby programming language. Ruby enables fast development of prototypes for which fast execution speed is not an essential issue. It is also the primary programming language in use at Axini and several necessary utilities such as the `STS` parser and the Treesolver constraint solver are available as Ruby libraries. See section 5.4 for a small overview of the Ruby programming language.
- *Only allow certain models to be closed.* This prototype cannot close most of the `STS` models as the required theoretical background wasn't available and couldn't be developed given the available time. Several research projects that discussed a general approach accepted the existence of false negatives [49, 12]. Within this project, the decision was made to ensure

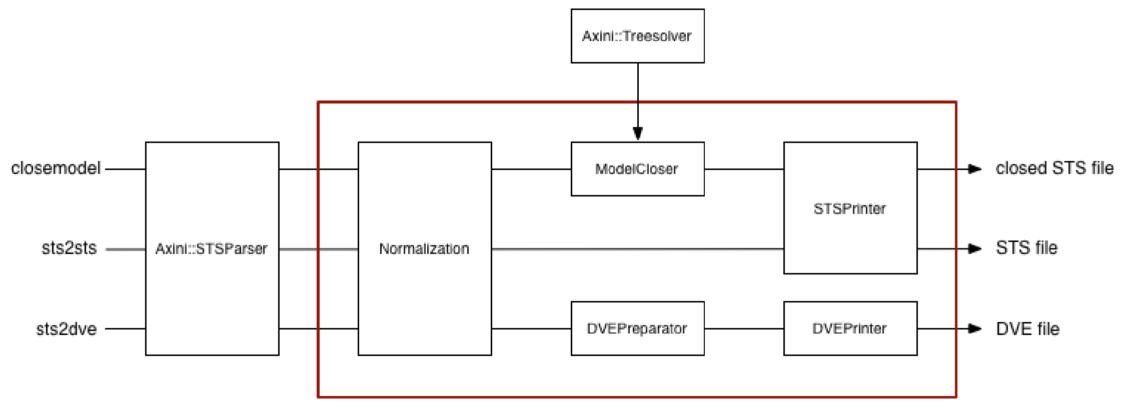


Figure 5.1: Global overview of the implementation.

the correctness of the results at the cost of reduced applicability. Future work might either deliver a general algorithm without false negatives or produce a method that can help discern between the negatives that are false and those that are true. Section 5.8 discusses the method of closing, and describes which models can be closed. Chapter 7 and appendix C discuss possible future work for dealing with this limitation.

5.3 Overview

The implementation exists as a small toolset of executable Ruby scripts where each tool has its own separate task. Combining these applications allows one to close an `STS` model, translate it to DVE and model check it. Listing 5.1 shows how these scripts can be combined into a bash shell one-liner as each script reads its input through `stdin` and writes its result to `stdout`. Following is a list of the individual scripts found in the toolset.

- `sts2sts`: reads an `STS` model, normalizes it and writes the result to `stdout`. The normalization is discussed in section 5.6 and the output is `STS` code that is conform to the language specification described in appendix A. While this script doesn't appear in listing 5.1, it can be used to inspect the result of the normalization that is used during both closing and DVE translation.
- `sts2dve`: reads an `STS` model, prepares the normalized model for DVE translation and writes the result as DVE code to `stdout`. The preparations that are necessary for DVE translation are described in section 5.7.
- `closemodel`: reads an `STS` model, closes it, and writes the resulting model as `STS` to `stdout`. The method of closing is described in section 5.8.
- `checkmodel`: is a wrapper for the LTSmin model checker. It reads a DVE model from `stdin` and passes it on to LTSmin. LTSmin's output is written to `stdout`. Listing 5.2 shows the command LTSmin is executed with. The `-d` flag instructs the model checker to check for deadlocks and terminate as soon as it has found one.

Listing 5.1: A POSIX shell one-liner that enables model checking

```
cat <model.sts> | closemodel | sts2dve | checkmodel
```

Listing 5.2: How LTSmin is executed for model checking.

```
dve2lts-seq -d <model.dve>
```

Three of the four previously listed scripts use a common library that contains utilities for parsing models that are written in the `STS` language, normalizing, closing and finally printing the resulting model. Figure 5.1 shows how these scripts use the components within the library to produce their output. The components within the red box were developed during the course of this project while `Axini::STSParser` and `Axini::Treesolver` already existed as a part of TestManager. What follows is a list of the components that are shown within the image.

- `Axini::STSParser`: an external component that was developed by Axini for use with TestManager. It parses `STS` models and generates an internal representation that forms a *symbolic transition systems* and consists solely of Ruby objects. This representation is passed on to the normalizer.

- `Normalization`: a component that takes an internal representation that is generated by the `Axini::STSParser` and prepares it for further processing. The `STS` parser contains some bugs that alter the internal representation by e.g. introducing deadlocks. These introduced deadlocks are different from those that belong to the model and are recognizable as such. `Normalization` removes these and other wrongly-introduced issues from the model. Furthermore, `Normalization` minimizes the model by applying *bisimilarity minimization* and subsequently removing all duplicate transitions. The entire process is described in section 5.6.

- `STSPrinter`: a component that prints a normalized internal representation as `STS`. The generated output is conform to the language as it is describes in appendix A. Due to the fact that the model is normalized and won't contain any Ruby method declarations, the output could be drastically different from the original `STS` model. However, using *bisimilarity checking* to compare the original input with the output, we managed to ensure to some extent that the original model is behaviorally equivalent to the result. This verification was done using several different models of varying complexity and this result validates, to some extent, both the results of the `STSPrinter` and the `Normalization` components.

- `ModelCloser`: a component that first determines whether a given normalized internal representation of a model can be closed. If not, it throws an exception. Otherwise, this component closes the model according to the method described in section 5.8. An environment process is added for each process that is in the original model and communicates over an external channel. The original processes remain unaffected, safe for the modifications that are applied by the `Normalization` component.

- `Axini::Treesolver`: another external component that was developed by Axini. It is a finite domain constraint solver that is mainly implemented in Prolog and accessible through a Ruby library. It was originally designed for use with TestManager so that it could generate message values for communication with the SUT. Such messages are specified in the `STS` model but without a concrete value.

- `DVEPreparator`: a component that alters certain parts of the model structure so that it can be printed in DVE. More specifically, the transmission of each valuated message is combined with a feedback mechanism where the receiver informs the sender whether the received value adheres to the constraint that was placed on the receiving transition. More details on the implementation are given in section 5.7.

- `DVEPrinter`: a component that converts a prepared model to a DVE string. The DVE code that is generated is correct according to the 2.4 version of the DiViNE model checker. LTSmin uses this specific version of DiViNE to read DVE models.

5.4 Ruby

As was previously mentioned, the programming language Ruby was selected for the development of the prototype. The main reasons for this decision are that this language was already used extensively at Axini and that many of the components that were necessary for the parsing of `STS` models and the communication with the constraint solver `Treesolver` were available as Ruby libraries. As Ruby is a mature general-purpose programming language, there would be no reason why the prototype could not be developed in this language.

Most of the remarks in the remainder of this section have been extracted from interviews with the creator, Yukihiro Matsumoto [51, 60, 59]. Ruby is a relatively young, dynamic object-oriented language that is mainly based on Perl and has adopted important features from Smalltalk, Lisp and Python. While Ruby is in itself a potent language, its incorporation into the web development framework *Ruby on Rails* significantly increased its popularity.

Ruby has some interesting features that outline its design philosophy and differentiate it from most other popular general-purpose programming languages. The following list will describe some of the most important of these features.

- Ruby was designed for writing succinct and compact code. Its main focus lies on programmer productivity, with less focus on producing fast or robust applications. The language includes few special safety features that would help programmers with writing bug-free code and they are presumed to test their software extensively during development. Consequentially, Ruby is a language that is well-suited for creating prototypes in a relatively fast time.

- With the correct preparations, C code can be loaded into the Ruby runtime and from that point on be called by Ruby methods. The C code first needs to be wrapped with the correct interface handlers and then compiled into a shared object before it can be dynamically loaded into the Ruby runtime. While many programming languages offer this feature as well, this feature seems to be more easy and powerful with Ruby than with other popular languages. This feature complements the previous feature, in that a prototype, once it ceases to be a prototype and becomes a mature software product, can be further optimized by replacing efficiency bottlenecks with highly optimized C code.

- Ruby has a less strict syntax than many other languages. Syntactical elements such as parentheses and braces are only enforced by the interpreter when an ambiguity would appear otherwise. This makes Ruby a popular language for designing domain specific languages with relatively little effort as such languages can be built from keyword sequences that are underneath the domain specific layer actually Ruby method calls. Good examples of DSLs built on top of Ruby are Axini's modelling language `STS`, the unit test specifications for `rspec` and the configuration files for the Ruby package maintainer RubyGems.

- Finally, the Ruby development framework contains several useful and mature tools that help the Ruby programmer with detecting bugs, maintaining application dependencies, and releasing the application to the user. These tools are readily available to the programmer through RubyGems and can be installed automatically along with the developed application.

5.5 LTSmin

LTSmin is a model checker that is maintained by the *Formal Methods and Tools* research group operating at the University of Twente[2]. LTSmin has a modular design that enables language-agnostic implementations of state space generation and property verification algorithms. This design is mainly facilitated by the PINS module.

PINS, short for *Partitioned Next-State Interface* is a module placed between the language-specific components, and the language-agnostic components. PINS enables language-agnostic features by forcing the language-specific components to adopt a certain interface so that they expose the part of the model structure that is sufficient to enable high-performance generation and verification algorithms [41].

The design of PINS is based on the idea of *event locality* which is the notion that most events within the model only affect a small portion of the system [7, 8]. Special state space generation techniques such as *partial order reduction* benefit from event locality, as they counter the combinatorial explosion of all possible event sequences by only generating a single sequence when these events all depend upon or affect separate non-overlapping regions of the state space. A language-specific component is required to generate a dependency matrix in which groups of transitions are linked to the portions of the state vector that they read or write to. Such a matrix can be generated with static analysis [41].

Language-specific components are also required to implement a successor generating function. Such a function takes an explicit state vector and returns all its successor states. LTSmin is capable of reading popular model specification languages such Promela [53], Uppaal [22], DVE and several others by modifying the existing model checkers for these languages so that they produce successor generating code in C or C++ instead of checking the model themselves. The resulting code is then compiled to a shared object and loaded dynamically into LTSmin. By this method, individual models are converted into language-specific components.

LTSmin has been selected over other available model checkers for several reasons. The most important contender for LTSmin was SPIN [32, 31] which is also a high-performance model checker popular tool. Other popular model checkers are Uppaal and DiVinE, but these were not considered for selection as they lacked many of the important features that were available in LTSmin and SPIN. The following list contains several of the important reasons for selecting LTSmin over SPIN.

[2]http://fmt.cs.utwente.nl/

- SPIN only accepts models in the Promela language and successor generating code written in C. LTSmin accepts Promela and several other model specification languages as well as successor generating code written in C and C++. This variety gives us more freedom in selecting an intermediate language for the `STS` models.

- LTSmin supports symbolic state space representation, allowing it to store larger state spaces in sublinear memory space. Currently, SPIN can only generate and store state spaces in an explicit fashion. A recent comparison of LTSmin and SPIN revealed that SPIN was only capable of generating the state space for GARP2 with 3.3×10^{11} states using lossy compression, while LTSmin managed to both correctly deal with GARP2 and store the entire state space for another model with 7.8×10^{20} states within 39MB [53]. Such symbolic methods would allow Axini to check some of their larger `STS` models.

- LTSmin accepts properties expressed in CTL and LTL [55, 53]. SPIN only accepts correctness claims specified in LTL [31].

- Recent comparisons have shown that for reachability analysis on several selected models from BEEM [45], LTSmin is faster than SPIN on single-core executions. LTSmin also benefits from a nearly linear speed-up with multi-core executions, while SPIN doesn't appear to use algorithms that were developed to exploit multiple CPU cores [53]. As multi-core machines are nowadays common in both workstations and servers, Axini can exploit this nearly-linear speedup to generate verification results in significantly less time.

5.6 Normalization

The normalization step prepares the internal representation of the `STS` model for further processing. This is done by *a*) removing unreachable states that were introduced by `Axini::STSParser`, *b*) removing certain deadlocks that were also introduced by `Axini::STSParser`, *c*) merging states that are only, directly and indirectly, connected by τ transitions with no update or constraint statements, *d*) removing any redundant transitions between states, and finally *e*) reducing the number of states and transitions by applying *bisimilarity minimization.* To verify the correctness of the output, i.e. to ensure that the deadlocks and property violations within the original model are retained during this normalization, several models of varying complexity were normalized and the output was compared to their original counterparts using *bisimilarity checking.*

The deadlocks introduced by `Axini::STSParser` are recognizably different from those that could be introduced by the original `STS` model. Such states have no outgoing transitions (as deadlocks often do), are reachable from the initial state, and the transitions that directly lead into them are τ-transitions without `update` or `constraint` modifiers. Removing these states and the silent τ-transitions may produce new deadlocks that are also not part of the original model, so this step is repeated until such states are no longer found.

The method of *bisimilarity minimization* is based upon an algorithm of *bisimilarity checking* that was originally developed by Kanellakis and Smolka [35, 1]. This method compares two models by first merging their states into a single set Pr and then splitting this set into the *coarsest partition* π, where

- $\pi = \{B_0, B_1, \ldots, B_k\}$,
- $B_i \cap B_j = \varnothing$, for all $0 \le i < j \le k$, and
- $Pr = B_0 \cup B_1 \cup \cdots \cup B_k$.

The sets B in a partition π are called *blocks.* A *splitter* for a block B_i is a block B_j within the same partition so that for some event a, B_i contains states that have an a-labeled transition leading to a state in B_j as well as states that do not have such a transition. When there exists a splitter B_j for block B_i and an event a so that $\exists s \in B_i, s' \in B_j : s \xrightarrow{a} s' \wedge \exists s \in B_i : \nexists s' \in B_j : s \xrightarrow{a} s'$ then B_i can be split to produce B_i^1 and B_i^2 where

- $B_i^1 = \{\ s \in B_i \mid \exists s' \in B_j : s \xrightarrow{a} s'\ \}$, and
- $B_i^2 = B_i \setminus B_i^1$.

Within the existing partition π, B_i is replaced by B_i^1 and B_i^2 to produce a new partition π'. This algorithm will always terminate, because eventually a fixpoint will be reached where no splitter can be found for any block. Such a coarsest partition will either consists of only blocks that each contain a single element, or of blocks containing only states that agree on their leaving transitions. Two models are bisimilar iff each block in the resulting coarsest partition contains states from both models.

The events are represented by the transitions between the states. For STS, two transitions represent the same event, iff they have the same string representation and this representation includes their constraint and update modifiers. Listing 5.3 shows two distinct STS processes that are bisimilar because they can emit exactly the same behavior. The process process_b is actually a reduced form of process_a as its underlying graph contains less states and less transitions while still being able to display the exact same behavior.

Listing 5.3: Two processes that are bisimilar

```
external 'a_channel'

sts('process_a') {
  response 'value_is', {'_value' => :integer}, 'a_channel'

  var 'x', :integer, 0

  choice {
    o { update 'x = 5;'
        update 'x = x * 2;'
        send 'value_is' { :constraint '_value = x' }}
    o { update 'x = 7;'
        update 'x = x * 2;'
        send 'value_is' { :constraint '_value = x' }}
  }
}

sts('process_b') {
  response 'value_is', {'_value' => :integer}, 'a_channel'

  var 'x', :integer, 0

  choice {
    o { update 'x = 5;' }
    o { update 'x = 7;' }
    update 'x = x * 2;'
    send 'value_is' { :constraint '_value = x' }
  }
}
```

Partition refinement algorithms such as Kanellakis and Smolka can also be used for bisimilarity minimization. The algorithm is modified to work on a single model instead, that is: Pr will only contain states from a single model. The algorithm will then partition these states into blocks where all states within the same block will allow the same future behavior. The reduced model is generated from the blocks within the final partition by selecting a single state from every block and generate the transitions between these states accordingly. Partition refinement algorithms can be used to minimize labeled transition systems, but there are some problems when they are applied to symbolic transition systems [10].

Figure 5.2 shows how bisimilarity checking can be used to minimize a model. The initial partition π places all states within a single block B_0. The partition π' is then produced by the splitting of block B_0 with itself on event a. The coarsest partition π'' is subsequently created from the partition π' by splitting the block B_1' with itself on event c. Finally, a single state is selected from each block to produce a minimal model.

Due to the fact that the algorithm was designed to minimize labeled transition systems, and not symbolic transition systems, it is possible that the resulting reduced model is not the minimal bisimilar model. Future work is needed to ensure that the minimalization is done correctly and produces the smallest bisimilar model.

5.7 DVE Printing

DVE is a model specification language that was originally designed for the DiVinE model checker [4]. The prototype converts the STS models to DVE so that they can be read by the LTSmin model

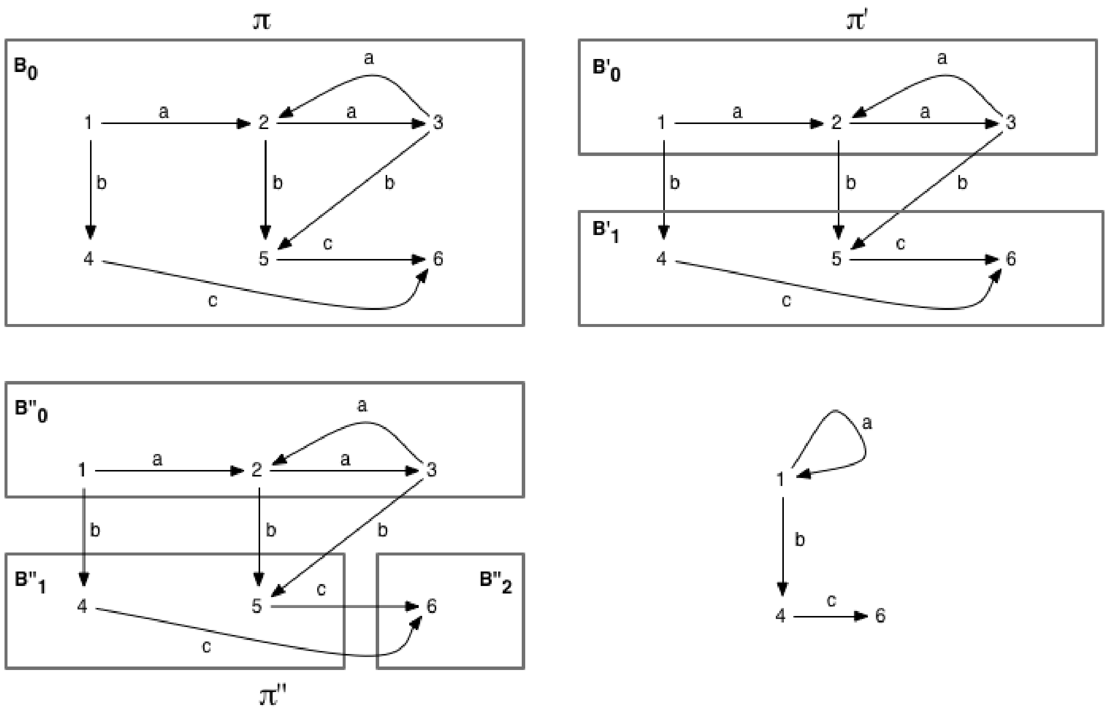


Figure 5.2: Application of the bisimilar minimization algorithm to a small model.

checker. DVE was selected over Promela and other languages supported by LTSmin because the model specifications are relatively easy to generate in this language.

However, there are two important differences between STS and DVE that prevent a direct translation. The underlying graph structure had to be modified to deal with these issues and enable a valid translation. These issues are as follows.

- *STS allows guards on message receiving transitions that refer to message parameters.* These constraints could prevent a value from being received - and the message from being communicated - if the transmitted value fails the constraint. DVE only allows constraints to refer to process variables.
- *STS supports messages with multiple parameters.* Although such messages are also supported by the latest version of the DiVinE model checker, the DVE parser used by LTSmin doesn't support such messages.

The latter issue was dealt with by splitting multi-valued messages into several single-valued messages that must be transmitted successively The former issue required a feedback mechanism to be added to every message transition. In case of a transition that received a multi-valued message and placed a constraint on at least one of these parameters, the feedback mechanism was added before the message was broken up.

The feedback mechanism consisted of two new messages that are to be send by the receiving process to the sending process. These messages are `success` and `failure`. The sending process was modified to only continue with its normal operation when it receives the `success` message. Otherwise, if it receives `failure`, it will fall back to the state that it was at before it transmitted the valued message. Within the receiving process, the transmitted values are temporarily stored within process variables where they are then subjected to the constraints. Figure 5.3 shows receiving and sending processes after the structural modifications have been applied to the respective STS fragments shown in listing 5.4.

A subtle issue with this method is that it might remove certain deadlocks from the original model and replace them with inescapable loops that only contain the transmissions of the valued message and the message `failed`. Models with such deadlocks contain a process that can reach a point from where it can only transmit a message that is not accepted by any other process. This issue is somewhat mitigated by the fact that these deadlocks can still be detected with liveness properties.

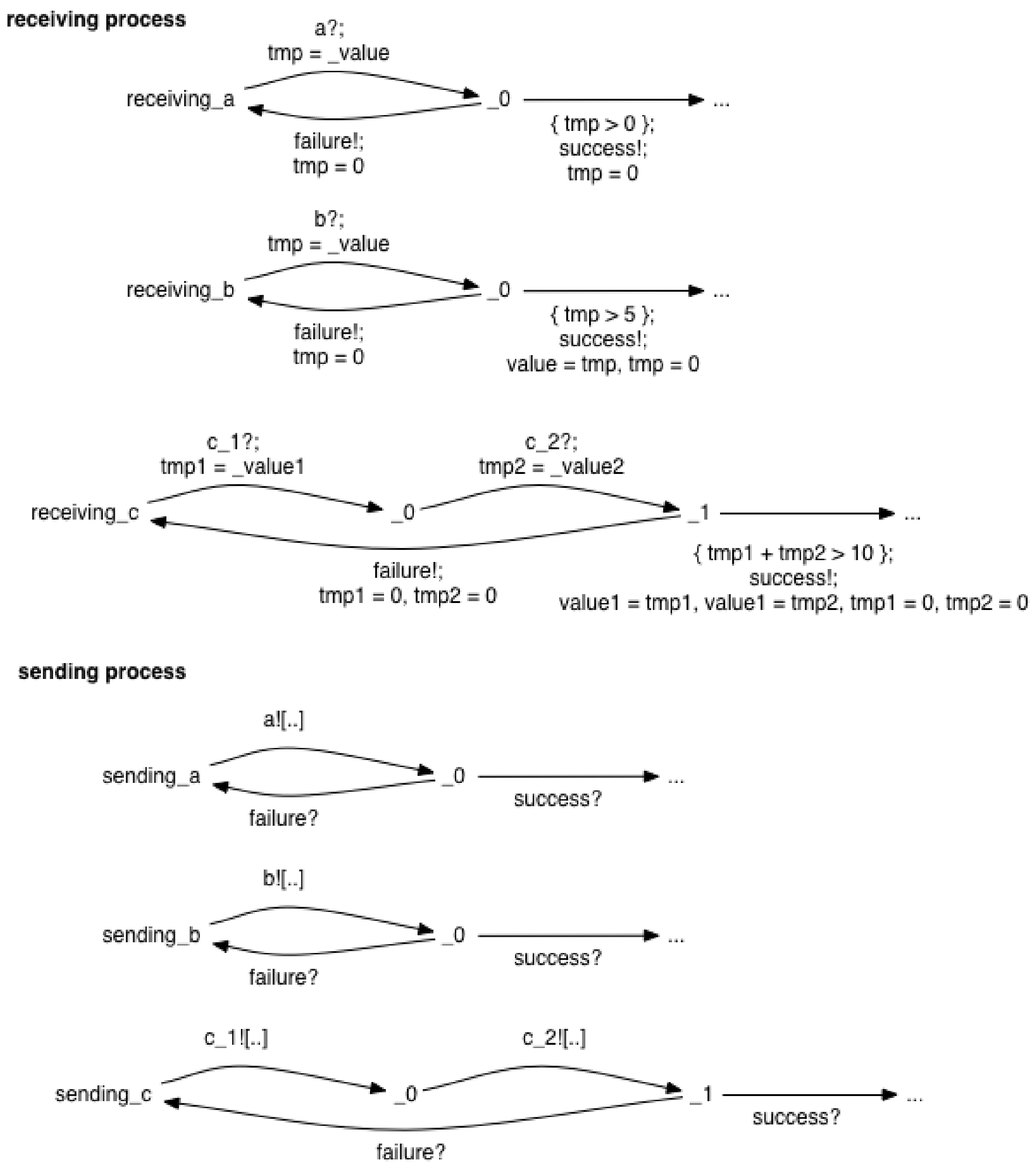


Figure 5.3: Result of DVE restructuring applied to fragments shown in listing 5.4.

Listing 5.4: Several STS fragments for receiving transitions

```
channel('channel_a') {
  stimulus 'a', {'_value' => :integer}
  stimulus 'b', {'_value' => :integer}
  stimulus 'c', {'_value1' => :integer, '_value2' => :integer}
}

state 'receiving_a'
  receive 'a' { :constraint => '_value > 0' }

state 'receiving_b'
  receive 'b' { :constraint => '_value > 5'; update => 'value = _value;' }

state 'receiving_c'
  receive 'c' { :constraint => '_value1 + _value2 > 10',
                :update => 'value1 = _value1; value2 = _value2' }
```

5.8 Model Closing

An STS model needs to be closed so that it can be model checked by LTSmin. An STS model is open when it contains a process that communicates with another process that is not contained within the model. TestManager can be such an external process. The approach used during this project closes the model by adding a finite process that represents the *maximum environment*. Combined with this new process, the original process will retain all of its behavior and all of its verifiable properties. However, to keep this environment small enough and so to minimize the resulting state explosion, a sufficiently small representation of this maximum environment is produced instead while still enabling the original model to retain its properties. Unfortunately, this algorithm is not a general one: there are many STS models that fall outside its scope and cannot be closed with this algorithm.

The process of closing a model consists of several steps. First, before the closing process is started, some form of static analysis is used to determine whether the model is within scope. If this isn't the case, then an exception is thrown and the process aborted. The closing step uses static analysis to gather all constraint sequences and determines which values the environment needs to send to ensure that all paths within the original processes remain traversable. These steps are discussed within the remainder of this section. Listing 5.5 contains a simplified pseudocode representation of the entire model closing algorithm. This version has been designed to be as succinct as possible and many important details have been omitted for clarity.

5.8.1 Scope

There are three important limitations that are placed upon the STS models that determine whether these models can be closed with the implemented algorithm or not. The first limits the datatypes that can be used, the second limits the way in which two processes can communicate, and the last limits the sequence in which variables can be read from or written to within cycles.

Currently, models can only use integer datatypes. Such datatypes are readily accepted by the DVE parser that is used by LTSmin. Extending the set of accepted datatypes is difficult, because DVE only accepts integer datatypes. A possible solution is to map the non-supported datatypes to integers datatypes, but this is difficult because not only do the distinct values need to be mapped to integers, but the datatype-specific operations need to be translated to integer operations as well. For obvious reasons, support for non-integer datatypes has been omitted for this project.

Another limitation prevents model checking on STS models with multiple processes that internally communicate valued messages. The current algorithm closes each process on its own, ignoring the values that it receives or sends over internal channels. Processes that do receive values over internal channels cannot be correctly closed with this algorithm, as this algorithm doesn't consider the variables that are influenced by neighboring processes for its static analysis. The sending processes might therefor not transmit the values that are necessary to keep all paths open.

The third limitation pertains to the sequence of read/write operations on variables within cycles. Static analysis is used to detect each cycle and determine, for every variable that is referred to, what the read/write pattern is. The underlying notion is that when a variable is read from before it is written to in a single cycle iteration, or over an unbroken sequence of cycle iterations, the following constraint or update expressions that refer to the affected variable could be irreducible. The model closer normalizes the constraint and update expressions by replacing all references to process variables with the expressions that were last assigned to them. Given a model with such a cycle pattern, an expression that replaces the reference to the affected variable, could be infinitely long, or there could be an infinite amount of different expressions to replace that variable with. The model closer is then not capable of ensuring that all sufficient values are generated to keep all paths open and all verifiable properties preserved.

An out of scope model is shown within listing 5.6. The process within this model can emit a certain number of `sum_<=_10` messages and then either send `sum_==_11` or halt in a deadlock. Determining a minimal environment for such a process is very difficult as its behavior does not only depend upon the most recent input, but could also depend upon the input values that it has received in the past. Such an minimal environment has to ensure that the deadlock is retained so that LTSmin may detect it. An environment that only transmits the message `add` with the value `1`, will cause the model to lose that deadlock.

Listing 5.5: Pseudocode for the model closer

```
def modelcloser(sts_model)
  # First, check if model is in scope.
  if in_scope(sts_model) == false
    error("Model cannot be closed!")
  end

  closed_model = sts_model.duplicate
  for every process pr ∈ closed_model:
    env_name     = pr.name + "_env"
    env_process = new sts_environment_process(env_name)

    # normalize all constraints: the normalized constraints
    #  only consist of operations on constants and interaction
    #  variables.
    normalized_process = normalize_constraints(pr)

    for every non-cyclic path pa ∈ normalized_process:
      path_constraint = ⊤

      # follow path from start to end
      for every guarded transition t ∈ pa:
        path_constraint = path_constraint ∧ t.constraint

        # Ensure that any deadlocks are preserved
        if there is a guarded transition t_ with t_.origin == t.destination:
          deadlock_constraint = path_constraint
          for every guarded transition t_ with t_.origin == t.destination:
            c = t_.constraint
            deadlock_constraint = deadlock_constraint ∧ ¬c
          end

          # If there is a solution for deadlock_constraint, then
          #   env_process must send it to retain the deadlock.
          solution = Axini::Treesolver.solve(deadlock_constraint)
          for every interaction variable i ∈ path_constraint:
            m = pr.message_for_interaction_variable(i)
            env_process.send_message_with_value(m, solution[i])
          end
        end
      end

      # After all constraints in pa have been gathered, ensure that
      #   pa remains traversable by making the environment send the
      #   values that are sufficient for it to be traversed.
      solution = Axini::Treesolver.solve(path_constraint)
      for every interaction variable i ∈ path_constraint:
        m = pr.message_for_interaction_variable(i)
        env_process.send_message_with_value(m, solution[i])
      end
    end
    closed_model.add_process(env_process)
  end
  return closed_model
end
```

Listing 5.6: `broken_summer` model, which cannot be closed

```
external 'my_channel'

sts('summer') {
  var 'sum', :integer, 0

  channel('my_channel') {
    stimulus 'add', {'_value' => :integer}
    response 'sum_==_11'
    response 'sum_<=_10'
  }

  state 'init'
    receive 'add', {:constraint => '_value > 0';
                    :update => 'sum = sum + _value;'}
    choice {
      o { constraint 'sum == 11'
          send 'sum_==_11'
          update 'sum = 0;'
      }
      o { constraint 'sum <= 10'; send 'sum_<=_10' }
    }
    goto 'init'
}
```

5.8.2 Constraint gathering and analysis

The process of closing consists of the following steps:

1. Normalize all constraint expressions. References to process variables are replaced with the expressions that were last assigned to these variables. These normalized expressions consist solely of operations on integer constants and message variables.

2. Generate the constraint sequences:

    - Traverse all possible paths and collect the constraints on the individual transitions into constraint sequences. For each path that contains at least one guarded transition, a single constraint sequence is made. The elements of these sequences are the constraints as they are part of the transitions and due to the preceding normalization, these constraints only consist of operations on integer constants and message parameters. A solution for such a sequence would allow the represented path to be traversable.

    - Gather all states with guarded transitions that leave them, and generate for each state a constraint sequence that makes such states reachable but inescapable. Such sequences are generated by combining a constraint sequence that make the state reachable, with the negations of all guards on the leaving transitions. If this combination has a solution, then the state can be inescapable.

3. For every generated constraint sequence, use TestManager's Treesolver to determine the values for the message parameters so that all individual elements of the sequence are satisfied.

4. Finally, for every generated value, create a message that is to be transmitted by the environment process.

Due to the third scope limitation, all references to process variables can be replaced with assignment expressions that are of finite length. When there are parallel assignments, then any constraints that have references with multiple replacements are duplicated. An example of this duplication is shown in listing 5.7 where the `choice` block allows multiple parallel assignments to the same variable. The second part of this fragment shows what the results of the normalization and duplication are: the single path with the two constraints is now duplicated and any references to `var` are replaced with `_value / 2` and `_value + 3` respectively.

Listing 5.7: Normalized and duplicated STS constraints

```
sts('process_a') {
  receive 'message', { :constraint '_value > 0',
                       :update 'var = _value;' }
  choice {
    o { update 'var = var / 2;' }
    o { update 'var = var + 3;' }
  }
  constraint 'var > 3'
  constraint 'var < 5'
}

sts('normalized_a') {
  receive 'message', { :constraint '_value > 0',
                       :update 'var = _value;' }
  choice {
    o { update 'var = _value / 2;' }
    o { update 'var = _value + 3;' }
  }
  choice {
    o { constraint '_value / 2 > 3' }
    o { constraint '_value + 3 > 3' }
  }
  choice {
    o { constraint '_value / 2 < 5' }
    o { constraint '_value + 3 < 5' }
  }
}
```

For the STS fragment shown in listing 5.7, two sets of constraint sequences will be generated. The first set will contain the sequences that enable all paths to be traversed.

- `[_value > 0, _value / 2 > 3, _value / 2 < 5]`, and
- `[_value > 0, _value + 3 > 3, _value + 3 < 5]`

The second set consists of the constraint sequences that retain the deadlocks.

- `[_value > 0, _value / 2 <= 3]`
- `[_value > 0, _value + 3 <= 3]`
- `[_value > 0, _value / 2 <= 3, _value + 3 <= 3]`
- `[_value > 0, _value / 2 > 3, _value / 2 >= 5,]`
- `[_value > 0, _value / 2 > 3, _value + 3 >= 5]`
- `[_value > 0, _value / 2 > 3, _value / 2 >= 5, _value + 3 >= 5]`
- `[_value > 0, _value + 3 > 3, _value / 2 >= 5]`
- `[_value > 0, _value + 3 > 3, _value + 3 >= 5]`
- `[_value > 0, _value + 3 > 3, _value / 2 >= 5, _value + 3 >= 5]`

The solutions for the first set are 8 and 1 respectively. All paths of the program will be traversable when the environment sends messages with these values. The solutions for some of the sequences in the second set are 1, 2, 8 and 10. The other sequences have no solution, and no messages are created for them. A final observation to make is that when there are solutions for sequences in the second set, there is strong possibility that the model has a deadlock and that this deadlock can be triggered when it receives the solution as a value. It is uncertain whether this is always the case when there are solutions for sequences in the latter set.

Chapter 6

Results

This chapter consists of two parts. The first section, 6.1, describes the results that were produced during the execution of this product. Section 6.2 describes the functionality of the prototype in detail by using case studies.

6.1 Deliverables

The following results were produced during the execution of this project:

- A prototype that enables `STS` models to be checked, containing
  - Implementations of the Kanellakis and Smolka *bisimilarity checking* algorithm and the *bisimilarity minimization* algorithm based upon it. These are described in section 5.6.
  - A method that converts the internal *symbolic transition system* representation to a string of `STS`, according to the language definition given in appendix A.
  - A method that converts the internal *sts* representation to a string of DVE, described in section 5.7.
  - An algorithm for model closing that retains model properties, described in section 5.8.
  - A method for determining whether models can be closed by the model closing algorithm, as is described in section 5.8.1.
  - A set of unit tests that test various aspects of the prototype.
- An implementation of the algorithm by Gerth, et al. that converts LTL formula to Büchi automata [30]. This algorithm is implemented as it was described by Clarke et al. in [19].
- A new interface for the TREESOLVER. This new interface is implemented in C and offers increased stability and support for generating multiple solutions per query.
- Research that was done to answer the questions that are listed in section 1.3. The results are discussed in chapter 7 and quickly listed here:
  - results on how `STS` models can be model checked.
  - results on how `STS` models can be closed.
  - results on how deadlocks can be detected in `STS` models without using a model checker.

6.2 Prototype functionality

The prototype that was developed during the course of this project is capable of checking certain `STS` models for deadlocks. This section demonstrates the usability of this prototype by showing some interesting `STS` models along with the verdicts from the model checker. The implementation itself contains a test suite, written in `rspec`, that tests the model closing, DVE translation and model checker verdict with respect to certain interesting `STS` models. For all of these models, the model checker managed to correctly determine whether the model contained a deadlock or not. Several of these models are included within this thesis and one has been selected for a more in-depth case study given in section 6.2.2.

name	listing	deadlock
`sends`	B.1	yes
`sends+loop`	B.2	no
`receive+constraints`	B.3	yes
`receive+constraints+different`	B.4	no
`sends_sum_of_values+virtual`	B.5	yes
`sends_sum_of_values+lazy`	B.6	yes
`constrained_receive`	B.7	no
`minimal_values`	B.8	yes

Table 6.1: Selected models

As the developed product is only a prototype, this chapter not contain any measurement of speed, CPU usage or memory utilization. Furthermore, most of the intensive work is done by the LTSmin model checker on the closed DVE model, while the remainder of the prototype only serves to prepare the `STS` model for model checking. This conversion only needs to be done once per `STS` model.

6.2.1 Case studies

The prototype was tested with 20 small `STS` models. These models were originally constructed to test different elements of the DVE translator and the closing process. These models each consist of a single process that communicates with its environment through several messages. Some of these processes contain multiple paths, of which some can only be taken when the environment sends a message with a specific value. The model closer must ensure that all these paths remain open by creating an environment process that is capable of sending the values that are sufficient for keeping them open. The model closer must also ensure that any deadlocks within the original model are retained within the closed model. This is done by ensuring that the environment also sends the values that cause these deadlocks to appear.

For all 20 models, the model checker returns the correct result; that is: it correctly determined whether the model did or didn't contain a deadlock. This would indicate that for the tested models, the closing and translating was also done correctly. Table 6.1 lists some of these 20 models which were included in appendix B. One of these models, `sends_sum_of_values+virtual`, is discussed more in-depth in section 6.2.2.

For the list of models, those named `sends`, `receive+constraints`, `sends_sum_of_values+lazy`, `sends_sum_of_values+virtual` and `minimal_values` contain deadlocks. The first model doesn't contain a cycle, so it will only execute a finite amount of steps before reaching the final state. Such types of deadlocks are trivial to detect, as the underlying symbolic transition system will contain at least a single state with no outgoing transitions. The other three models do contain a cycle, but also place some constraints on the received values which are too strict with respect to the constraints they place on the receiving transitions. For example, `sends_sum_of_values+lazy` contains two receiving transitions which store their values in variables `value1` and `value2`. The constraints on the receiving transitions only state that the two values must be larger than zero. However, the deadlock is introduced by the constraint `value1 > 3`. The environment might send a value $0 < x < 3$ that is sufficient for the constraint on the receiving transition, but not for the second constraint. As there are no other possible transitions leaving the state between transitions $\xrightarrow{b?,\ \texttt{\_value > 0}}$ and $\xrightarrow{\tau,\ \texttt{value1 > 3}}$, the system cannot continue from that state.

A different model is `constrained_receive`, which does not have a deadlock. The value that is received and stored in `value` conforms to $0 < x < 100$. The optional branch can be taken whenever the received value is larger than 10 and continued when it is less than 100, but since the constraint on the receiving transition already forces the received value to be less than 100, the second constraint will always be satisfied when it is reached and therefor not cause a deadlock.

6.2.2 In-depth study of `sends_sum_of_values+virtual`

`sends_sum_of_values+virtual` (listing B.5) is a model that contains a deadlock. Its state graph is shown in figure 6.1. There are two processes: the original process `process_a` and environment `process_a_env` that was generated by the model closer. Both processes start at their `start` state and independently from each other traverse their transitions, only syncing when one traverses a `send` and the other a corresponding `receive` transition. The original process receives two numbers and stores them in variables `value1` and `value2`. If these numbers are both greater than or equal to 5, an optional branch may be taken and which can only be continued when the sum of these numbers is 10. Of course, when these numbers are both equal to 5, then this optional branch cannot be continued. This is a deadlock and is successfully detected by the LTSmin model checker.

Using the commands shown in listing 6.1, a user can access the counterexample for the deadlock. Such a counterexample is a single path that leads from the initial state of the model to the deadlock. Figure 6.2 shows the entire state space of the model with the counterexample highlighted in red. This image was created manually and is based on LTSmin's output. From this graph, several edges have been omitted for clarity. These edges connect the leaves, except for the deadlock, to the root through the $\texttt{\_6} \xrightarrow{\texttt{value1=0, value2=0}} \texttt{start}$ transition.

Listing 6.1: Commands for obtaining a counterexample

```
dve2lts-seq -d <model.dve> --trace=<trace.gcf>
ltsmin-printtrace <trace.gcf>
```

The states within this graph are shown with the vector representation that is internally used by LTSmin. This notation contains for each process an element that stores the current state of the process, and furthermore contains all process variables. For this model, the state vector is $\langle S_A, \texttt{value1}_A, \texttt{value2}_A, \texttt{tmp}_A, S_{env}, \texttt{tmp}_{env} \rangle$, where S_A and S_{env} store the current locations of the two processes. The two `tmp` variables are inserted by the DVE translator and the `value1` and `value2` variables appear within the original `STS` model.

Looking at figure 6.2, one can see that the deadlock occurs when `process_a` is in state `_2` and the values of `value1` and `value2` are both equal to 5. As can be seen within 6.1, there is only one transition that leaves state `_2` and which can only be taken when the sum of `value1` and `value2` equals 11 or higher. The state space image also shows that there is a single configuration in which this transition can be taken. This is the case when the received values are 5 and 6, and it is shown by the rightmost part of the tree.

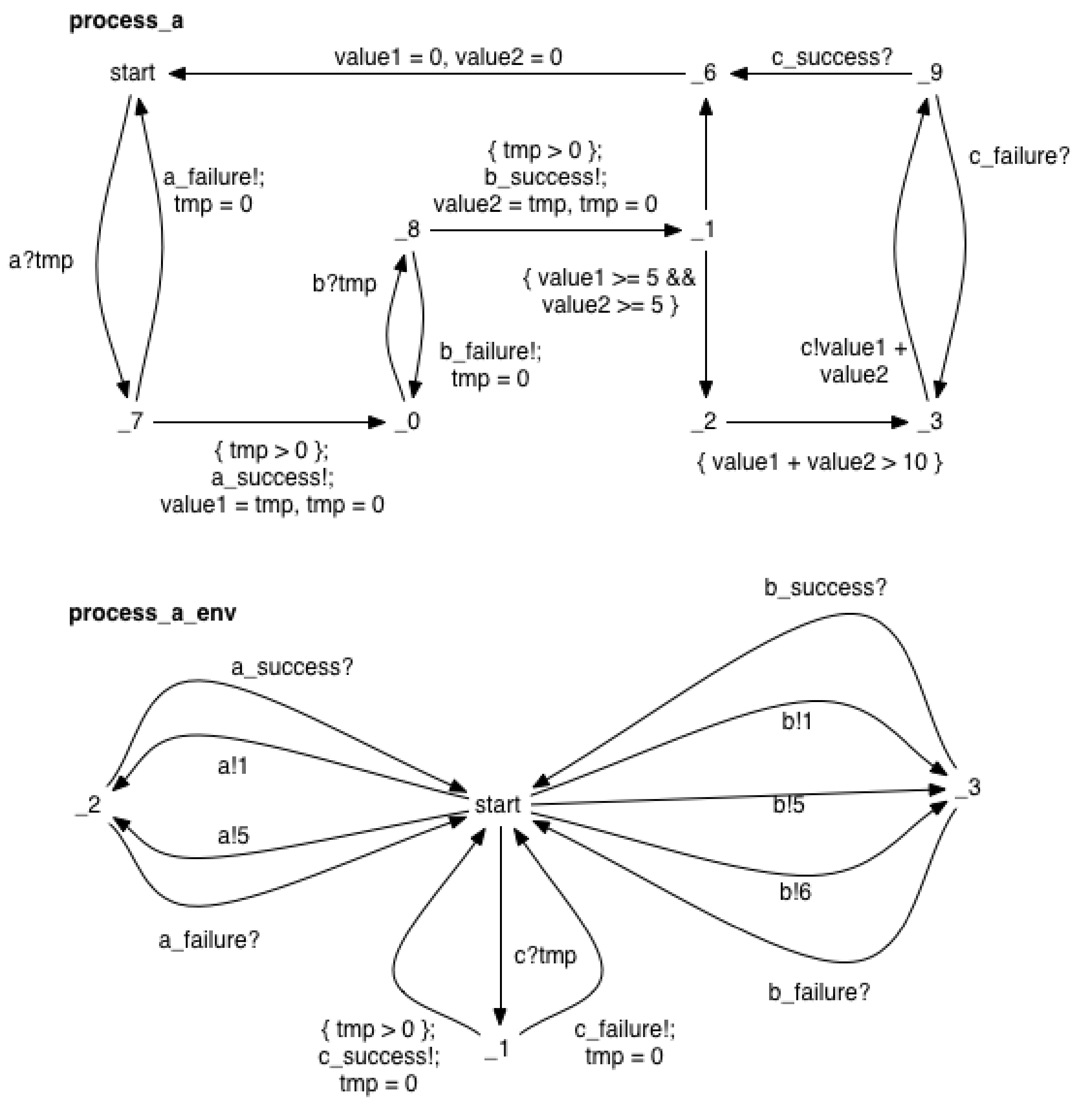
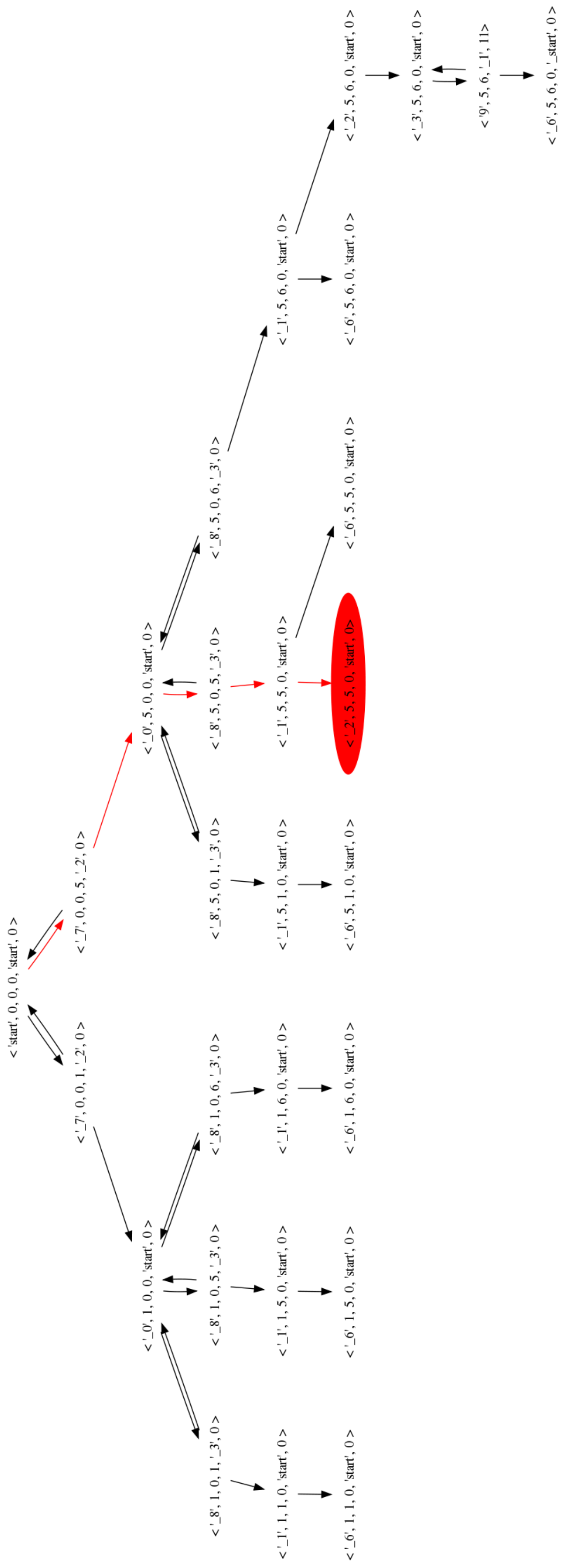


Figure 6.1: State graph for model `sends_sum_of_values+virtual`



< 'start', 0, 0, 0, 'start', 0 >
< '_7', 0, 0, 1, '_2', 0 >
< '_7', 0, 0, 5, '_2', 0 >
< '_0', 1, 0, 0, 'start', 0 >
< '_0', 5, 0, 0, 'start', 0 >
< '_8', 1, 0, 1, '_3', 0 >
< '_8', 1, 0, 5, '_3', 0 >
< '_8', 1, 0, 6, '_3', 0 >
< '_8', 5, 0, 1, '_3', 0 >
< '_8', 5, 0, 5, '_3', 0 >
< '_8', 5, 0, 6, '_3', 0 >
< '_1', 1, 1, 0, 'start', 0 >
< '_1', 1, 5, 0, 'start', 0 >
< '_1', 1, 6, 0, 'start', 0 >
< '_1', 5, 1, 0, 'start', 0 >
< '_1', 5, 5, 0, 'start', 0 >
< '_1', 5, 6, 0, 'start', 0 >
< '_6', 1, 1, 0, 'start', 0 >
< '_6', 1, 5, 0, 'start', 0 >
< '_6', 1, 6, 0, 'start', 0 >
< '_6', 5, 1, 0, 'start', 0 >
< '_2', 5, 5, 0, 'start', 0>
< '_6', 5, 5, 0, 'start', 0 >
< '_6', 5, 6, 0, 'start', 0 >
< '_2', 5, 6, 0, 'start', 0 >
< '_3', 5, 6, 0, 'start', 0 >
< '9', 5, 6, '_1', 1>
< '_6', 5, 6, 0, '_start', 0 >

Figure 6.2: State space for model `sends_sum_of_values+virtual` with the deadlock highlighted.

Chapter 7

Evaluation and future work

This chapter contains an evaluation on how the project was executed and on the results that were described in the previous chapter. We reflect on some of the decisions that were made and previously listed in section 5.2, discuss the current limitations of the developed prototype and finally list related and future work.

7.1 Evaluation

Reviewing the original research goals and the results that were obtained, we can claim that this project was at least a minor success. The developed prototype shows that some of the `STS` models can be verified for some of the verifiable properties (i.e. deadlocks). As support for LTL and CTL verification are already available within the model checker, it should not require too much effort to modify the prototype into making these features available as well. We may therefor argue that some of the `STS` models can also be verified for deadlock-freedom and LTL-expressible properties. There are some other issues that limit the effectiveness and applicability of the prototype, but we will argue that once these issues have been resolved or bypassed, the developed prototype will be a useful addition to the Axini workflow, as it should allow fast, automatic and correct verification of the existing `STS` models.

The existing prototype can currently detect deadlocks within certain `STS` models. Such an ability is useful for detecting the circumstances in which a model can escape or evade its main cycle and reach a terminal state (n.b. all finite models without cycles have deadlocks, so we will assume that all interesting `STS` models have cycles,) or enter a state from which it cannot leave due to the specific values that are stored within the process or message variables. These deadlocks were previously defined as *explicit* and *implicit deadlocks* respectively. While explicit deadlocks can be readily detected by a manual inspection of the `STS` code (the final expression within an `STS` process is reachable but is not within a cycle), detecting an implicit deadlock requires for every state an enumeration of the values that can be stored in all process and message variables along with an evaluation of the relevant transition constraints. An implicit deadlock is only discovered when the evaluation has revealed that for a reachable state there exists a reachable value assignment to the variables that prevents all of the transitions that leave this state from being traversed: none of the available transitions can be taken until the values have been changed, but the values cannot change until a transition is taken. An example of this was previously discussed in section 6.2.2 where $\{ \texttt{value1} \mapsto 5, \texttt{value2} \mapsto 5 \}$ is a reachable assignment and state `_2` a reachable state, but none of the transitions that leave `_2` can be traversed given that assignment.

Implicit deadlocks can appear in reactive processes that internally do not account for all the possible values that they receive. In other words: there exists an input that the process accepts but cannot deal with. An ability to detect such issues within the models is a useful feature for Axini as it allows the detection of missing behavior in models and the detection of constraints that are too strict or too loose. Furthermore, if an `STS` model is shown to adhere to all SUT requirements, but still contains deadlocks, then this issue could indicate that the requirements themselves are not complete and need to be further developed.

As the prototype can currently only detect deadlocks, there is no support for the verification of properties other than *deadlock freedom.* There are also several other issues that prevent the prototype from becoming a part of the Axini workflow. The prototype is limited in the following ways: *a*) LTL and CTL properties cannot be verified yet, *b*) the implemented model closing algorithm cannot deal with many STS models, and *c*) several software bugs cause the prototype to terminate prematurely while closing a small group of models. While most of these limitations can be removed without too much significant effort, developing a closing algorithm that can deal with all possible STS models requires some theoretical effort to be made. Furthermore, there are suggestions that there exist some STS models that cannot be model checked, or at least not completely. These limitations are discussed more in-depth later in this section. The future work that is necessary to deal with these issues is discussed in section 7.3 and in appendix C.

Assuming that future work is capable of dealing with these issues, then model checking can be of significant use to Axini. If the SUT requirements can be converted to temporal logic formulas, then the model can be automatically verified after the model or the requirements have been changed. This operative independence could allow Axini and the other users of TestManager to place more focus on ensuring that the SUT is correct as models can be developed and maintained in less time. Such a shift would allow faster, or a more in-depth testing of the SUT, as less resources are needed to verify the model. It is therefor important to continue with this project and improve the prototype so that it becomes a useful, stable, reliable and fast component.

7.1.1 Existing Issues

What follows is a list of the issues that still trouble the prototype. Some of them need to be fixed before the prototype can be incorporated within the Axini workflow. Others, such as the effects of the DVE restructuring, need to be considered when checking for deadlock freedom. Future work needed to deal with these issues is discussed in section 7.3.

- Model checking cannot deal with STS models that have an underlying infinite labeled transition system. For several kinds of properties, model checkers have to examine every state of the transition system before they can declare that the system is correct. This would mean that for infinitely large transition systems, the model checker can never terminate and the problem of model checking be undecidable. However, the process will terminate if the model contains a counterexample to the property, or when the property can be verified by only examining a finite subset of the state space. A possible future approach that only checks a subset of the state space is *bounded model checking* [6], which only inspects the states that are a certain predetermined distance away from the initial states.

- The prototype cannot deal with models in which the updates and constraints can be normalized to infinitely long expressions. As was previously described in section 5.8.1, the model checker generates a maximum environment by gathering all the constraint paths and generating the values that will keep all paths traversable as well as retain all deadlocks. These constraint paths are finite collections of constraint expressions, which need to be finite themselves if a constraint solver were to generate the solutions.

- The prototype cannot deal with models that use datatypes other than integers. Translating the various datatypes to non-negative integers, not only requires a mapping of non-$\mathbb{N}_0$ values to non-negative integers, but also a translation of the datatype-specific operators such as string concatenation.

- The prototype cannot deal with processes that communicate values to each other over an internal channel. Again, as was previously explained in 5.8.1, the model closing algorithm only analyses each process independently. To correctly close such a process, its behavior to its neighbors should also be retained, and so the static analysis needs to enter the neighbors and determine which values the transmitting process needs to remain sending to keep its neighbors' behavior identical.

- The implemented bisimilar minimization algorithm (Kanellakis and Smolka's [35]) was originally designed for labeled transition systems, but is executed on the internal Ruby runtime representation of STS, i.e. a symbolic transition system. There are several consequences of this, one consequence is that the result is not necessarily the minimal bisimilar model.

- The graph restructuring that is necessary to translate STS to DVE replaces certain deadlocks with inescapable progressless cycles. Deadlock detection can not detect these issues, but certain liveness properties still can.

- The current implementation is too slow. It is uncertain where the bottlenecks are, but the algorithms were not designed to be efficient.

- The prototype contains several issues that cause it to prematurely terminate when it is model checking certain models. One of such models is SCRP, an industrial model that was available during this project and developed by Axini. The set of models for which the prototype crashes is small in comparison to the set of models that are in scope and do not cause the prototype to crash.

7.1.2 Reflecting on Decisions

Previously in this document, a list of decisions was discussed that affected the design, implementation and scope of the prototype. In this section, we reflect upon these decisions.

The most important decision was to limit the scope of the closing algorithm. This allowed us to develop a small and relatively simple algorithm within the available time, while still being able to handle certain complex models. However, this was at the cost of reduced applicability. Future work can be done to remove or relax this limit, thereby allowing more or all `STS` models to be model checked. For some of the models that cannot be completely checked, it might still be possible to only verify them up to a certain extent. Also, due to the fact that we were capable of generating a working algorithm with reduced effort, we were also capable of developing the entire framework that surrounds the closing algorithm. Together with the closing algorithm, the entire framework can model check the models that are accepted by the model closer.

It is often better to initially focus on developing a small, but working prototype. This prototype would be able of demonstrating the potential of its approach, and allow the developers to detect and deal with any important issues prematurely.

Another important decision is to generate only correct results. There are some other approaches to the problem of model checking open models, which accept the existence of false negatives. We have deemed such results to be detrimental to the adoption of automatic model checking into the Axini workflow. Forcing the user to manually verify the results that are produced by the prototype, introduces again the element of uncertainty that we wanted to cast out by using formal methods, both also requires more of the user's effort into maintaining the model. Future work should either ensure that the properties remain correct, or add a method with which the user can differentiate false from true results.

The decision to use DVE as in the intermediate model specification had both positive and negative effects. It requires relatively less effort to design and implement a system that converts the output of the model closer to DVE, than it would have for e.g. the Promela language. However, translating an `STS` model to DVE requires the message receiving transitions to be modified, as their guards may not refer to the message variables, but only to the process variables. The implemented restructuring retains the guards and their effects, but also removes certain deadlocks from the model. The effect of this restructuring can not be demonstrated by the current prototype, as this effect only appears when two processes attempt to communicate over an internal channel, and models that contain such processes fall outside the scope of the model closing algorithm.

Finally, the decision to use LTSmin and not another model checker has not significantly affected the prototype thus far. However, once the previously listed limitations have been removed, LTSmin will be a better choice for the prototype than DiVinE or SPIN as LTSmin outperforms both by a significant margin in both execution speed, and the size of the models that it can deal with.

7.1.3 Research

We start this section by repeating the two questions that were raised in the introduction (section 1.3).

- Can we model check `STS` models?
- How can an open model be properly closed?

A positive answer to the first question has been demonstrated by the prototype: `STS` models can be model checked. Our approach uses a publicly available model checker and transforms the model so that it is accepted by the selected model checker. Furthermore, an answer to this question can be brought about in the following way:

- SUT requirements can be translated to formulas of temporal logic. A model checker can check whether a labeled transition system is correct with respect to the specified property.
- `STS` models encode symbolic transition systems, these can be converted to labeled transition systems.

Together, these two parts suggest that STS models can be checked with respect to the SUT requirements. Unfortunately, not all STS models can be model checked. Only those STS models that can be correctly translated to a finite labeled transition system, which would consist of a finite number of states and transitions, can be model checked as a model checker can only deal with finite systems. Such infinite models can be very trivial, for example, listing 5.6 contains such a model that cannot be completely model checked. However, if a STS model were to be altered so that it would produce a finite labeled transition system then such a model can be model checked, but this could possibly produce fictitious results.

Future research is needed to answer three new questions:

Future Research Question 1. *How to detect whether an* STS *model produces an infinite labeled transition system?*

Future Research Question 2. *How to alter an* STS *model with a infinite labeled transition system so that it has a finite labeled transition system, without losing the properties or retaining as many properties as possible?*

Future Research Question 2a. *If this method allows for false results from the model checker, how could we detect them as such?*

A possible alternative approach would be to avoid transforming the STS model to a labeled transition system and directly model check the symbolic transition system instead. Furthermore, it seems that this approach would allow data to be preserved so that models that use datatypes other than integers, can be reliably model checked.

Future Research Question 3. *What is the current research on model checking symbolic transition systems?*

Future Research Question 3b. *Is existing work in this field sufficient for model checking models with data?*

Future Research Question 3c. *Does existing work in this field make for a viable alternative to the current approach of model checking labeled transition systems?*

As for the second question mentioned in the introduction, we have uncovered several approaches to model closing. Essential to such attempts is the confinement of the external decisions, the resulting model which then only contain internal decisions and won't depend upon external agents for its specific behavior. The approaches that are relevant for this project are the following:

- Inserting the maximum environment as a process or set of processes. This is the standard approach, but is susceptible to the state explosion problem [33]. To minimize the resulting state explosion, our approach tried to produce a minimal process that could still represent the maximum environment.

- *Chaos embedding*, which closes the model by removing the guards that refer to externally-influenced variables and the transitions that perform the external communication [49]. The effect is that the external decisions will no longer have any effect on the model. However, this approach can generate many false negatives, and was therefor not selected.

There are several other approaches, such as *module checking* [36], which we're not deemed relevant to this project as we trying to use a standard model checker to do the actually task of verifying the models.

As the implemented approach doesn't necessarily generate the smallest environment process, new research is needed to determine how such a smallest maximum environment can be generated. As this generation needs to be performed for every version of an STS model, this process should be done in a sufficiently quick matter so that it can be a valuable tool during the model development. Furthermore, this future approach to model closing should also either retain the model properties, or allow for a method that can discern between the verification results that are fictitious and those that aren't.

Future Research Question 4. *How can the smallest process be generated which represents the maximum environment for the given model?*

Future Research Question 4d. *If this method allows for false results from the model checker, how could we detect them as such?*

7.2 Related Work

No research has been found that has tried to model check test models. However, as `STS` test models are a specific form of *open* or *reactive models*, we were able to find related work on model checking such models.

Mocha[1] is a publicly available model checker that tests reactive models that are written in the ReactiveModules language [2]. The verifiable properties have to be specified in alternative temporal logic (ATL).

A separate approach that uses a more standard model checker has been proposed by Sidorova et al. [49, 50]. This approaches transforms open models so that they become closed, and then uses an existing standard model checker, such as SPIN, to verify the properties. Their approach abstract a model to a *safe abstraction* by removing all externally-influenced variables and external communication. The resulting model could possess more behavior than the original, and so certain properties could have become lost. However, certain precautions are made so that positive verification results remain accurate.

Another approach by Kupferman and Vardi is called *module checking* [36, 37]. The set of states of a *module*, or *open model*, are separated into the *environment states* and the *system states*, from where the environment and the system make their transitions respectively. They then discuss algorithms for the verification of CTL and CTL* formulas in such *modules*, but refrain from discussing practical applications.

Finally, the approach that we used to close a model is similar to *boundary value analysis*, which is used in software testing [43]. The important common aspect is that both approaches split the set of input values into various, possibly non-overlapping, subsets and then select a single value from each subset. However, where boundary value analysis selects the values that are on or near the boundaries between the separated sets, the approach used in this project can select any value from a subset to represent its subset.

7.3 Future Work

Future work that is necessary for the continuation of this project consists of the following actions:

1. Fix the issues that are listed in section 7.1.1 and in appendix C.
2. Solve the research questions that are presented in section 7.1.3.

Some other tasks that are not necessary but would help the integration of the prototype into the Axini workflow, or would improve the execution speed are the following:

1. To ease the integration:
   (a) Convert SUT requirements to LTL or CTL formulas.
   (b) Convert LTL or CTL formulas to an `STS` model skeleton. Emerson's dissertation and some of his early work with Clarke would be a good starting point [24, 16, 26].
2. Investigate whether deadlocks can be accurately detected during model closing.
3. Combine the current model closing approach with elements from chaos insertion [49]. If some models cannot be closed by inserting an environment process, we can perhaps still close the model by removing the influenced variables.

[1] http://www.cis.upenn.edu/m̃ocha/

Chapter 8

Conclusion

During the execution of this project we have tried to enable model checking for automated test models, or STS models. This project was a partial success, as we managed to show that this approach is possible for some STS models, and have listed several approaches to deal with the STS models that we haven't been able to close.

However, in its current state, the developed cannot be integrated into the Axini workflow. Future work is needed to deal with the issues that were previously listed. Once these issues have been dealt with, model checking can become a valuable tool that could perhaps alter the way in which Axini constructs test models and also allow them to spend more effort on testing the models.

The main issue that hinders the integration, is that the prototype cannot deal with many STS models. Most of these issues originate from the algorithm that was used to close the models. Several alternative approaches have been discussed that may resolve this issue, but such approaches should try to retain the model properties so that when the model checker is integrated into the workflow: it should only produce results that are reliable.

Chapter 9

Bibliography

[1] L. Aceto, A. Ingolfsdottir, and J. Srba. The algorithmics of bisimilarity. In D. Sangiorgi and J. Rutten, editors, *Advanced Topics in Bisimulation and Coinduction*, Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2012.

[2] R. Alur, T. A. Henzinger, F. Y. C. Mang, S. Qadeer, S. K. Rajamani, and S. Tasiran. MOCHA: Modularity in model checking. In A. J. Hu and M. Y. Vardi, editors, *Computer Aided Verification*, volume 1427 of *Lecture Notes in Computer Science*, pages 521–525. Springer Berlin Heidelberg, 1998.

[3] C. Baier and J. P. Katoen. *Principles of Model Checking*, volume 950. MIT Press, Cambridge, MA, USA, 2008.

[4] J. Barnat, L. Brim, V. Havel, J. Havlíček, J. Kriho, M. Lenčo, P. Ročkai, V. Štill, and J. Weiser. DiVinE 3.0 – An Explicit-State Model Checker for Multithreaded C & C++ Programs. In N. Sharygina and H. Veith, editors, *Computer Aided Verification*, volume 8044 of *Lecture Notes in Computer Science*, pages 863–868. Springer Berlin Heidelberg, 2013.

[5] G. Behrmann, A. David, and K. G. Larsen. A Tutorial on Uppaal. In M. Bernardo and F. Corradini, editors, *Formal Methods for the Design of Real-Time Systems*, volume 3185 of *Lecture Notes in Computer Science*, pages 200–236. Springer Berlin Heidelberg, 2004.

[6] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic Model Checking without BDDs. In W. R. Cleaveland, editor, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1579 of *Lecture Notes in Computer Science*, pages 193–207. Springer Berlin Heidelberg, 1999.

[7] S. C. C. Blom and J. C. van de Pol. Symbolic Reachability for Process Algebras with Recursive Data Types. In J. S. Fitzgerald, A. E. Haxthausen, and H. Yenigun, editors, *Theoretical Aspects of Computing - ICTAC 2008*, volume 5160 of *Lecture Notes in Computer Science*, pages 81–95. Springer Berlin Heidelberg, 2008.

[8] S. C. C. Blom, J. C. van de Pol, and M. Weber. Bridging the Gap between Enumerative and Symbolic Model Checkers, June 2009.

[9] S. C. C. Blom, J. C. van de Pol, and M. Weber. LTSmin: Distributed and Symbolic Reachability. In T. Touili, B. Cook, and P. Jackson, editors, *Computer Aided Verification, Edinburgh*, volume 6174 of *Lecture Notes in Computer Science*, pages 354–359. Springer Verlag, Berlin, July 2010.

[10] F. Bonchi and U. Montanari. Minimization Algorithm for Symbolic Bisimilarity. In G. Castagna, editor, *Programming Languages and Systems*, volume 5502 of *Lecture Notes in Computer Science*, pages 267–284. Springer Berlin Heidelberg, 2009.

[11] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. In A. R. Meyer, editor, *Information and Computation*, volume 98, pages 142–170. Elsevier, 1992.

[12] J. Calamé, N. Ioustinova, J. C. van de Pol, and N. Sidorova. Bug Hunting with False Negatives. In J. Davies and J. Gibbons, editors, *Integrated Formal Methods*, volume 4591 of *Lecture Notes in Computer Science*, pages 98–117. Springer Berlin Heidelberg, 2007.

[13] G. Ciardo, G. Lüttgen, and R. Siminiceanu. Efficient Symbolic State-Space Construction for Asynchronous Systems. In M. Nielsen and D. Simpson, editors, *Application and Theory of Petri Nets 2000*, volume 1825 of *Lecture Notes in Computer Science*, pages 103–122. Springer Berlin Heidelberg, 2000.

[14] G. Ciardo, G. Lüttgen, and R. Siminiceanu. Saturation: An Efficient Iteration Strategy for Symbolic State Space Generation. In T. Margaria and W. Yi, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2031 of *Lecture Notes in Computer Science*, pages 328–342. Springer Berlin Heidelberg, 2001.

[15] E. M. Clarke. The Birth of Model Checking. In O. Grumberg and H. Veith, editors, *25 Years of Model Checking*, volume 5000 of *Lecture Notes in Computer Science*, pages 1–26. Springer-Verlag Berlin Heidelberg, 2008.

[16] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In D. Kozen, editor, *Logics of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer Berlin Heidelberg, 1982.

[17] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Progress on the State Explosion Problem in Model Checking. In R. Wilhelm, editor, *Informatics*, volume 2000 of *Lecture Notes in Computer Science*, pages 176–194. Springer Berlin Heidelberg, 2001.

[18] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and Abstraction. In A. W. Appel, editor, *ACM Transactions on Programming Languages and Systems*, volume 16, pages 1512–1542. ACM, New York, NY, USA, 1994.

[19] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking.* MIT Press, Cambridge, MA, USA, 1999.

[20] E. M. Clarke, K. L. McMillan, S. Campos, and V. Hartonas-Garmhausen. Symbolic Model Checking. In R. Alur and T. A. Henzinger, editors, *Computer Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, pages 419–422. Springer Berlin Heidelberg, 1996.

[21] C. Courcoubetis, M. Vardi, P. Wolper, and M. Yannakakis. Memory Efficient Algorithms for the Verification of Temporal Properties. In E. M. Clarke and R. P. Kurshan, editors, *Computer-Aided Verification*, volume 531 of *Lecture Notes in Computer Science*, pages 233–242. Springer Berlin Heidelberg, 1991.

[22] A. E. Dalsgaard, A. W. Laarman, K. G. Larsen, M. C. Olesen, and J. C. van de Pol. Multi-Core Reachability for Timed Automata. In M. Jurdzinski and D. Nickovic, editors, *10th International Conference on Formal Modeling and Analysis of Timed Systems, FORMATS 2012, London, UK*, volume 7595 of *Lecture Notes in Computer Science*, pages 91–106, London, September 2012. Springer Verlag.

[23] E. W. Dijkstra. Structured programming. circulated privately, August 1969.

[24] E. A. Emerson. *Branching time temporal logic and the design of correct concurrent programs.* PhD thesis, Division of Applied Sciences, Harvard University, 1981.

[25] E. A. Emerson. The Beginning of Model Checking: A Personal Perspective. In O. Grumberg and H. Veith, editors, *25 Years of Model Checking*, volume 5000 of *Lecture Notes in Computer Science*, pages 27–45. Springer Berlin Heidelberg, 2008.

[26] E. A. Emerson and E. M. Clarke. Using Branching Time Temporal Logic to Synthesize Synchronization Skeletons. In *Science of Computer Programming*, volume 2, pages 241 – 266. Elsevier, 1982.

[27] E. A. Emerson and J. Y. Halpern. "Sometimes" and "Not Never" Revisited: on Branching versus Linear Time Temporal Logic. In M. J. Fischer, editor, *Journal of the ACM*, volume 33, pages 151–178. ACM, New York, NY, USA, 1986.

[28] L. Frantzen, J. Tretmans, and T. A. C. Willemse. Test Generation Based on Symbolic Specifications. In J. Grabowski and B. Nielsen, editors, *Formal Approaches to Software Testing*, volume 3395 of *Lecture Notes in Computer Science*, pages 1–15. Springer Berlin Heidelberg, 2005.

[29] L. Frantzen, J. Tretmans, and T. A. C. Willemse. A Symbolic Framework for Model-Based Testing. In K. Havelund, M. Núñez, G. Roşu, and B. Wolff, editors, *Formal Approaches to Software Testing and Runtime Verification*, volume 4262 of *Lecture Notes in Computer Science*, pages 40–54. Springer Berlin Heidelberg, 2006.

[30] R. Gerth, D. Peled, M. Y. Vardi, and P. Wolper. Simple On-the-fly Automatic Verification of Linear Temporal Logic. In P. Dembinsky and M. Sredniawa, editors, *Proceedings of the Fifteenth IFIP WG6.1 International Symposium on Protocol Specification, Testing and Verification XV*, pages 3–18, London, UK, UK, 1996. Chapman & Hall, Ltd.

[31] G. J. Holzmann. The model checker SPIN. In R. A. Kemmerer, L. K. Dillon, and S. Sankar, editors, *IEEE Transactions on Software Engineering*, volume 23, pages 279–295. IEEE Press, Piscataway, NJ, USA, 1997.

[32] G. J. Holzmann. *the SPIN Model Checker: Primer and Reference Manual.* Addison-Wesley Professional, first edition, 2003.

[33] N. Ioustinova, N. Sidorova, and M. Steffen. Synchronous Closing and Flow Analysis for Model Checking Timed Systems. In F. S. Boer, M. M. Bonsangue, S. Graf, and W-P. Roever, editors, *Formal Methods for Components and Objects*, volume 3188 of *Lecture Notes in Computer Science*, pages 292–313. Springer Berlin Heidelberg, 2004.

[34] M. Jonge and T. C. Ruys. The SpinJa model checker. In J. van de Pol and M. Weber, editors, *Model Checking Software*, volume 6349 of *Lecture Notes in Computer Science*, pages 124–128. Springer Berlin Heidelberg, 2010.

[35] P. C. Kanellakis and S. A. Smolka. CCS Expressions, Finite State Processes, and Three Problems of Equivalence. In *Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing*, PODC '83, pages 228–240, New York, NY, USA, 1983. ACM.

[36] O. Kupferman and M. Y. Vardi. Module Checking. In R. Alur and T. A. Henzinger, editors, *Computer Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, pages 75–86. Springer Berlin Heidelberg, 1996.

[37] O. Kupferman and M. Y. Vardi. Module Checking Revisited. In O. Grumberg, editor, *Computer Aided Verification*, volume 1254 of *Lecture Notes in Computer Science*, pages 36–47. Springer Berlin Heidelberg, 1997.

[38] A. W. Laarman and D. Faragó. Improved On-The-Fly Livelock Detection. In G. Brat, N. Rungta, and A. Venet, editors, *NASA Formal Methods*, volume 7871 of *Lecture Notes in Computer Science*, pages 32–47. Springer Berlin Heidelberg, 2013.

[39] A. W. Laarman, R. Langerak, J. C. van de Pol, M. Weber, and A. Wijs. Multi-core Nested Depth-First Search. In T. Bultan and P-A. Hsiung, editors, *Proceedings of the 9th International Symposium on Automated Technology for Verification and Analysis, ATVA 2011, Tapei, Taiwan*, volume 6996 of *Lecture Notes in Computer Science*, pages 321–335, London, July 2011. Springer Verlag.

[40] A. W. Laarman, J. C. van de Pol, and M. Weber. Boosting Multi-Core Reachability Performance with Shared Hash Tables. In N. Sharygina and R. Bloem, editors, *Proceedings of the 10th International Conference on Formal Methods in Computer-Aided Design, Lugano, Switzerland*, pages 247–256, USA, October 2010. IEEE Computer Society.

[41] A. W. Laarman, J. C. van de Pol, and M. Weber. Multi-Core LTSmin: Marrying Modularity and Scalability. In M. Bobaru, K. Havelund, G. Holzmann, and R. Joshi, editors, *NASA Formal Methods*, volume 6617 of *Lecture Notes in Computer Science*, pages 506–511. Springer Verlag, Berlin, July 2011.

[42] L. Lamport. "Sometime" is Sometimes "Not Never": On the Temporal Logic of Programs. In *Proceedings of the 7th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '80, pages 174–185, New York, NY, USA, 1980. ACM.

[43] G. J. Myers and C. Sandler. *The Art of Software Testing.* John Wiley & Sons, 2004.

[44] R. Nicola and F. Vaandrager. Action versus State based Logics for Transition Systems. In I. Guessarian, editor, *Semantics of Systems of Concurrent Processes*, volume 469 of *Lecture Notes in Computer Science*, pages 407–419. Springer Berlin Heidelberg, 1990.

[45] R. Pelánek. BEEM: Benchmarks for Explicit Model Checkers. In D. Bošnački and S. Edelkamp, editors, *Model Checking Software*, volume 4595 of *Lecture Notes in Computer Science*, pages 263–267. Springer Berlin Heidelberg, 2007.

[46] A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, SFCS '77, pages 46–57, Washington, DC, USA, 1977. IEEE Computer Society.

[47] J. P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In M. Dezani-Ciancaglini and U. Montanari, editors, *International Symposium on Programming*, volume 137 of *Lecture Notes in Computer Science*, pages 337–351. Springer Berlin Heidelberg, 1982.

[48] T. L. Siaw. Saturation for LTSmin. Master's thesis, Universiteit Twente, February 2012.

[49] N. Sidorova and M. Steffen. Embedding Chaos. In P. Cousot, editor, *Static Analysis*, volume 2126 of *Lecture Notes in Computer Science*, pages 319–334. Springer Berlin Heidelberg, 2001.

[50] N. Sidorova and M. Steffen. Verifying Large SDL-Specifications Using Model Checking. In R. Reed and J. Reed, editors, *SDL 2001: Meeting UML*, volume 2078 of *Lecture Notes in Computer Science*, pages 403–420. Springer Berlin Heidelberg, 2001.

[51] B. Stewart. An interview with the creator of ruby. `http://www.linuxdevcenter.com/pub/a/linux/2001/11/29/ruby.html`, 2001. Accessed: 2014-02-06.

[52] J. Tretmans and E. Brinksma. TorX: Automated Model-Based Testing. In A. Hartman and K. Dussa-Ziegler, editors, *First European Conference on Model-Driven Software Engineering*, pages 31–43, December 2003.

[53] F. I. van der Berg and A. W. Laarman. SpinS: Extending LTSmin with Promela through SpinJa. In K. Heljanko and W. J. Knottenbelt, editors, *11th International Workshop on Parallel and Distributed Methods in verifiCation, PDMC 2012, London, UK*, volume 296 of *Electronic Notes in Theoretical Computer Science*, pages 95–105, Amsterdam, September 2012. Elsevier.

[54] M. van der Bijl. *On changing models in Model-Based Testing.* PhD thesis, Universiteit Twente, May 2011.

[55] T. van Dijk. The Parallelization of Binary Decision Diagram operations for model checking. Master's thesis, Universiteit Twente, April 2012.

[56] T. van Dijk, A. W. Laarman, and J. C. van de Pol. Multi-core and/or Symbolic Model Checking. In G. Lüttgen and S. Merz, editors, *12th International Workshop on Automated Verification of Critical Systems, AVoCS 2012, Bamberg, Germany*, volume 53 of *Electronic Communications of the EASST*, pages 773:1–773:7, Berlin, September 2012. EASST.

[57] M. Y. Vardi. An Automata-Theoretic Approach to Linear Temporal Logic. In F. Moller and G. Birtwistle, editors, *Logics for Concurrency*, volume 1043 of *Lecture Notes in Computer Science*, pages 238–266. Springer Berlin Heidelberg, 1996.

[58] M. Y. Vardi. Branching vs. Linear Time: Final Showdown. In T. Margaria and W. Yi, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2031 of *Lecture Notes in Computer Science*, pages 1–22. Springer Berlin Heidelberg, 2001.

[59] B. Venners. Dynamic productivity with ruby. `http://www.artima.com/intv/tuesday.html`, 2003. Accessed: 2014-02-06.

[60] B. Venners. The philosophy of ruby. `http://www.artima.com/intv/ruby.html`, 2003. Accessed: 2014-02-06.

Appendix A

The STS language

STS is a *domain specific language* designed to formally express the behavior of a *System Under Test.* Axini uses this language to model existing systems. These systems communicate with their environment, or more specifically, with other systems within that environment. STS allows an engineer to model the behavior of a system by specifying the messages that the system responds to, the messages that the system can emit, and the internal process that defines how and when the system communicates.

An STS model expresses a forest of directed graphs, in particular, it expresses a set of *symbolic transition systems.* These represent multiple processes (but a valid model contains at least one,) which can operate independently from each other, and these processes communicate with each other by passing messages. When a message is passed, both the receiving and sending process synchronously take an event-emitting transition. As both the receiving and sending process can place constraints on these transitions, the message can only be passed, and the transitions can only be taken, when both constraints permit this.

There are several languages that are similar to STS in their intent. That is, they are designed to allow engineers to express the behavior and internal processes of a system. Promela is a modelling language that was designed for use with the SPIN model checker. As a model checker, SPIN can verify whether a model adheres to certain desired properties. This model doesn't need to represent an implemented system as it is the model itself which is being verified.

The important commonalities between STS and Promela are

- Processes are the top-level entities. These operate independently from each other.
- Processes can have variables.
- Communication between processes is done through messages.
- Messages can carry values

The important differences

- Promela supports process creation and termination. The processes defined in the STS model start executing simultaneously. STS processes can terminate, but only in a deadlock.
- STS doesn't require that the values that are communicated are concretely specified, TestManager works with a constraint solver to determine the appropriate values.

As a language, STS is an extension of the Ruby programming language and the interpretation is mostly done by the standard Ruby interpreter. The STS extension defines several specific keywords and Ruby interprets these keywords as method calls. During the interpretation, the called methods alter the representation of the model that is internal to the Ruby runtime. They create new State or Transitions objects and these are combined to form a *symbolic transition system.*

Finally, while STS can support asynchronous communication, we only consider synchronous communication. Also, while STS provides a wide set of datatypes, we only use the integer datatype in our examples. STS's support for timed automata is also not discussed in this document.

A.1 Channels and Processes

The top level elements of an STS model are the channels, which are declared at the top of the model, and the processes. A channel has a unique name, and is either *internal* or *external*. Internal channels are pathways for (invisible) communication between the processes defined in the model, while external channels are meant for observable communication with entities outside the model. A model that receives messages over an external channel is an open system, as its behavior now depends on decisions that are not included within the model.

Following the channel declarations are the processes. Each is also identified with a unique name. A process declaration consists of its private variables, a listing of the messages it sends or receives, and finally a specification of its behavior.

Listing A.1 shows an open STS model with two communicating processes. These processes are defined with the sts keyword, and two processes communicate with each other through send and receive statements.

Listing A.1: Declaration of simple open model with two communicating processes.

```
external 'an_external_channel'
internal 'an_internal_channel'


sts('process_a') {
  stimulus 'public_message_for_a', {}, 'an_external_channel'
  channel('an_internal_channel') {
    stimulus 'private_message_for_a'
    response 'private_message_for_b'
  }

  state 'start'
    receive 'public_message_for_a'
    send 'private_message_for_b'
    receive 'private_message_for_a'
    goto 'start'
}

sts('process_b') {
  channel('an_internal_channel') {
    response 'private_message_for_a'
    stimulus 'private_message_for_b'
  }

  state 'start'
    receive 'private_message_for_b'
    send 'private_message_for_a'
    goto 'start'
}
```

Processes can have variables to hold values. All variables are assigned a value when the process starts. If no value is specified, the default value for the corresponding datatype (e.g. 0 for integer) is assigned. The values in these variables can be transmitted along with messages, or can be used in constraint or update expressions. Listing A.2 shows an inactive process with two integer variables, both holding the same value.

Listing A.2: Process with integer variables.

```
sts('process_a') {
  var 'var_1', :integer
  var 'var_2', :integer, 0
}
```

A.2 Communication

Communication between two processes happens when a message is sent from one to the other. When process P_A is at a certain state from where it can make a transition when it receives the message m and another process P_B is at a certain state from where it can make a transition by sending the message m, then the processes P_A and P_B can communicate the message m by synchronously taking these transitions from their current states. In STS, the receive and send constructs denote such transitions. Every message is named and assigned to the channel over which they are transmitted. At the start of the process definition, following the variable declarations, a process lists the messages that it sends or receives, along with the channels over which sends or receives them respectively. In listing A.1, `process_a` communicates over an internal channel with `process_b` and over the external channel with an entity not defined within the model. The two messages that `process_a` can receive are expressed with the `stimulus` keyword and `response` is used to denote the message that it sends.

Messages can carry values. A process that receives a valued message can either directly store this value inside a state variable, use it to alter a state variable, or ignore the value while still making the transition. Listing A.3 gives an example of these three applications.

Listing A.3: A process that can do three different things with the received value.

```
external 'a_channel'

sts('process_a') {
  var 'value', :integer

  stimulus 'a_message', {'v' => :integer}, 'a_channel'

  state 'start'
    choice {
      o { receive 'a_message', { :constraint => 'v >= 0',
                                     :update => 'value = v;' }}
      o { receive 'a_message', { :constraint => 'v >= 0',
                                     :update => 'value = value + v;' }}
      o { receive 'a_message', { :constraint => 'v >= 0' }}
    }
    goto 'start'
}
```

Receive and send transitions for valuated messages must specify which values they can receive or send respectively. This is done by placing a constraint on the message variable. The three receive transitions in listing A.3 each require a non-negative integer to be passed. These transitions will not be taken if an external process passes a negative number.

If the receiving and sending transitions communicate the same message and both place different constraints on the message variable, then the value domains specified by the two constraints must overlap if the message were to be send. If the value constraint of the receiving transition does not overlap with the value constraint of the sending transition, then both transitions cannot be taken as no message can be passed. If the value domains do overlap, then the value that is transmitted will be selected from the shared domain. Listing A.4 shows two processes that can only communicate the value 4, even though this value is not explicitly specified.

Listing A.4: Two processes that only communicate the value 4.

```
internal 'a_channel'

sts('process_a') {
  stimulus 'a_message', {'_value' => :integer}, 'a_channel'

  state 'start'
    receive 'a_message', { :constraint => '_value < 5' }
    goto 'start'
}

sts('process_b') {
  response 'a_message', {'_value' => :integer}, 'a_channel'

  state 'start'
    send 'a_message', { :constraint => '_value > 3' }
    goto 'start'
}
```

A.3 Control flow

Constraints on transitions can either restrict the value of a message or prohibit the transition from being taken within the current configuration of the process. Such a configuration is defined only in terms of process variables and their values. A constraint of the latter form would therefor have to refer to such variables and only allow the transition to be taken when the referred-to variables store acceptable values.

When a process is at a state from which it can only leave through a transition for which the constraint does not the hold, the process is in a deadlock. As the blocking constraint refers to the process configuration, and the process configuration can only be altered by taking transitions, this deadlock is permanent. Branching the control flow is an essential construct for avoiding this as it enables the specification of multiple transitions that lead out from the same state. These transitions don't need to lead towards the same state and each can be associated to a different constraint or effect expression. In `STS`, diverging branches are denoted with `choice` which allows an arbitrary number of branches from the state in which the `choice` was declared. Listing A.5 shows an `STS` fragment in which for any process configuration, either both or a single one of the branches can be navigated because there exists process configurations in which both constraints allow navigation. Listing A.6 shows an if/else statement encoded in `STS`, the else branch has a constraint that is the negation of the if-constraint. If the else-constraint was omitted or different, then the else branch could have been followed even when the if-constraint was permissive and that is not what an if/else statement is. Finally, listing A.7 shows an `STS` `choice` fragment along with its syntactic sugar.

Listing A.5: An STS fragment that sometimes allows two branches to be taken.

```
choice {
  o { constraint 'value < 8'
      send 'value_is_smaller_than_8' }
  o { constraint 'value > 5'
      send 'value_is_larger_than_5' }
}
```

Listing A.6: An STS fragment that models an if/else statement

```
choice {
  o { constraint 'value < 8'
      send 'value_is_smaller_than_8' }
  o { constraint 'value >= 8'
      send 'value_is_not_smaller_than_5' }
}
```

Listing A.7: An STS fragment that models an optional branch along with its syntactic sugar.

```
choice {
  o { constraint 'value < 8'
      send 'value_is_smaller_than_8' }
  o { }
}

optional {
  constraint 'value < 8'
  send 'value_is_smaller_than_8'
}
```

All branches of the choice meet again at the state directly after the `choice` unless the branches leave the `choice` construct with a `goto`, or enters an inescapable loop.

Under the assumption that `STS` models are finite, processes that do not use loops will eventually terminate as all possible paths within the process consists of finite steps. `STS` supports both an explicit and an implicit loop construct. The explicit method uses the `repeat` keyword to denote a set of instructions that will be repeated indefinitely until an escape with the `stop_repetition` or `goto` keywords. The implicit method uses `goto` to return to the start of the loop. Listing A.8 gives an example of an explicit loop where for each iteration, one out of two separate paths can be taken, and where one path always ends outside the loop. Listing A.9 gives an example of an implicit repeat, using goto to both return to the start and to escape.

Listing A.8: An STS fragment with an explicit loop

```
repeat {
  o { send 'stay_in_loop' }
  o { send 'will_quit_loop'; stop_retition }
}
send 'escaped_loop'
```

Listing A.9: An STS fragment that shows an implicit loop equivalent to the one in listing A.8.

```
state 'start_loop'
  choice {
    o { send 'stay_in_loop'; goto 'start_loop' }
    o { send 'will_quit_loop'; goto 'outside_loop' }
  }
state 'outside_loop'
 send 'escaped_loop'
```

Finally, the last important keyword is `update` which is used to assign a value to a process variable. See listing A.10 for a small assign the integer 5 to a process variable.

Listing A.10: An STS fragment that shows how a process value could be capped to 5.

```
choice {
  o { constraint 'value > 5'
      update 'value = 5;' }
  o { constraint 'value <= 5' }
}
```

Appendix B

Selected Models

Listing B.1: sends

```
external 'channel_a'

sts('process_a') {
  var 'value', :integer, 0

  channel('channel_a') {
    response 'a', {'_value' => :integer}
  }

  state 'start'
    update 'value = 10;'
    send 'a', { :constraint => '_value == value' }
}
```

Listing B.2: sends+loop

```
external 'channel_a'

sts('process_a') {
  var 'value', :integer, 0

  channel('channel_a') {
    response 'a', {'_value' => :integer}
  }

  state 'start'
    repeat {
      o {
        update 'value = 10;'
        send 'a', { :constraint => '_value == value' }
      }
    }
}
```

Listing B.3: receive+constraints

```
external 'channel_a'

sts('process_a') {
  var 'value', :integer, 0

  channel('channel_a') {
    stimulus 'a', {'_value' => :integer}
  }

  state 'start'
    receive 'a', { :constraint => '_value > 0', :update => 'value = _value;' }
    constraint 'value > 5'
    constraint 'value < 10'
}
```

Listing B.4: receive+constraints+different

```
external 'channel_a'

sts('process_a') {
  var 'value', :integer, 0

  channel('channel_a') {
    stimulus 'a', {'_value' => :integer}
  }

  state 'start'
    choice {
      o { receive 'a', { :constraint => '_value > 0',
                         :update => 'value = _value;' }}
      o { receive 'a', { :constraint => '_value > 12 && _value < 100',
                         :update => 'value = _value + 2;' }}
    }
    optionally {
      constraint 'value > 10'
    }
    goto 'start'
}
```

Listing B.5: sends_sum_of_values+virtual

```
external 'channel_a'

sts('process_a') {
  var 'value1', :integer, 0
  var 'value2', :integer, 0

  channel('channel_a') {
    stimulus 'a', {'_value' => :integer}
    stimulus 'b', {'_value' => :integer}
    response 'c', {'_sum' => :integer}
  }

  state 'start'
    repeat {
      o {
        receive 'a', { :constraint => '_value > 0',
                       :update => 'value1 = _value;' }
        receive 'b', { :constraint => '_value > 0',
                       :update => 'value2 = _value;' }
        optionally {
          constraint 'value1 >= 5 && value2 >= 5'
          constraint 'value1 + value2 > 10'
          send 'c', { :constraint => '_sum == value1 + value2' }
        }
        update 'value1 = 0; value2 = 0;'
      }
    }
}
```

Listing B.6: sends_sum_of_values+lazy

```
external 'channel_a'

sts('process_a') {
  var 'value1', :integer, 0
  var 'value2', :integer, 0
  var 'sum', :integer, 0

  channel('channel_a') {
    stimulus 'a', {'_value' => :integer}
    stimulus 'b', {'_value' => :integer}
    response 'c', {'_sum' => :integer}
  }

  state 'start'
    repeat {
      o {
        receive 'a', { :constraint => '_value > 0',
                       :update => 'value1 = _value;' }
        receive 'b', { :constraint => '_value > 0',
                       :update => 'value2 = _value;' }
        constraint 'value1 > 3'
        update 'sum = value1 + value2;'
        optionally {
          constraint 'sum > 10'
          send 'c', { :constraint => '_sum == sum' }
        }
      }
    }
}
```

Listing B.7: constrained_receive

```
external 'channel_a'

sts('process_a') {
  var 'value', :integer, 0

  channel('channel_a') {
    stimulus 'a', {'_value' => :integer}
    response 'b', {'_value' => :integer}
  }

  state 'start'
    repeat {
      o {
        receive 'a', { :constraint => '_value > 0 && _value < 100',
                       :update => 'value = _value;' }
        optionally {
          constraint 'value > 10'
          constraint 'value < 100'
          send 'b', { :constraint => '_value == value' }
        }
      }
    }
}
```

Listing B.8: `minimal_values`

```
external 'channel_a'

sts('process_a') {
  var 'value', :integer, 0

  channel('channel_a') {
    stimulus 'a', {'_value' => :integer}
  }

  state 'start'
    repeat {
      o {
        receive 'a', { :constraint => '_value > 0',
                       :update => 'value = _value;' }
        choice {
          o { constraint 'value == 1' }
          o { constraint 'value == 2' }
        }
      }
    }
}
```

Appendix C

Task Assignment Successor

C.1 Installation

The recommended operation system is any recent version of Linux. Early experiments have shown that LTSmin's version of `divine` is difficult to compile and install on Mac OS X. The prototype has been developed and tested within an Arch Linux installation that is running as a virtual machine on top of a Mac OS X machine.

To install the prototype, perform the following instructions:

1. Ensure that a recent version of git[1] is installed. Version 1.8.3.4 is known to work.
2. Ensure that LTSmin's version of the DiVinE model checker[2] is installed and is within `$PATH` available as `divine`.
3. Ensure that there is a VPN connection to the Axini internal network.
4. Within the appropriate folder, execute the commands that are shown in listing C.1

Listing C.1: The command needed to download and install the prototype source code.

```
> git gitolite@git.axini.com:students/dberk
> cd dberk
> bundle install
```

The prototype should now be installed. Use the following steps to determine if all installation steps were successful. The commands are executed from the root folder of the `dberk` git repository within a standard bash session.

- Check if the bundle is correctly installed:

  ```
  > cat test/cookbook/checking_age.sts
    | bundle exec bin/sts2dve > /tmp/checking_age.dve
  ```

  The result should be a non-empty DVE file that is located at `/tmp/checking_age.dve`

- Check if the LTSmin version of `divine` has been correctly compiled and installed with:

  ```
  > divine compile --ltsmin /tmp/checking_age.dve
  ```

  If the command was executed successfully, then there should be two files located within the `dberk` root folder named and `checking_age.dve.cpp` and `checking_age.dve2C`. The former file is the C++ source code that, when compiled to a shared object, produces the latter file. If `divine` claims that `--ltsmin` is an unknown option, then the wrong version of `divine` is placed within `$PATH`.

- Check if the entire prototype works. The result of the following execution should be similar to the result shown in C.2.

  ```
  > bundle exec cat spec/models/normalizeexpressions/sends_sum_of_values.sts |
    bin/closemodel | bin/sts2dve | bin/checkmodel
  ```

[1] http://git-scm.com/

[2] http://fmt.cs.utwente.nl/tools/ltsmin/#secx

name	description
`annotate`	Translate the Axini STS to an internal representation used by the prototype and then annotate the elements with important keywords to guide future transformations.
`fix`	Remove the errors that were introduced by `Axini::STSParser` (see section 5.6).
`normalize`	Remove any redundant transitions, states and minimize the sts with a bisimilarity minimization algorithm (see section 5.6).
`close`	Check if the model can be closed, then close the model.
`create_env`	Normalization the constraint expressions and generate the necessary counterprocess constraint expressions (see section 5.8).
`dve`	Restructure the model to allow a DVE translation (see section 5.7)

Table C.1: Several transformation recipes within the prototype.

- Finally, execute all `rspec` unit tests with:

  ```
  > bundle exec rspec
  ```

  It is expected that one test should fail, but the failure of this specific test unit is inconsequential and can be ignored.

Listing C.2: The expected result when model checking `sends_sum_of_values.sts`

```
Success
DVE: /tmp/kxPJF3oHXCw.dve
> dve2lts-seq -d /tmp/kxPJF3oHXCw.dve
dve2lts-seq: Precompiled divine module initialized
dve2lts-seq: loading model from /tmp/kxPJF3oHXCw.dve
dve2lts-seq: length is 5, there are 12 groups
dve2lts-seq: There are 14 state labels and 0 edge labels
dve2lts-seq: got initial state
dve2lts-seq: state space 9 levels, 13 states 21 transitions
```

C.2 Source code overview

An overview of the important folders that contain the source code is shown in figure C.1. The most important architectural design pattern that is used within the prototype is the *recipe*. The different transformations that are applied to the symbolic transition system produced by the `Axini::STSParser` are collected into recipes and table C.1 lists the most important of these. They are either stored in `recipestore.rb`, or loaded dynamically as in `spec/dberk/printer/stsprinter_spec.rb`. Each transformation accepts an *sts* model, and outputs an *sts* model that can be immediately inserted into the next transformation in the recipe. A recipe is thus a collection of transformations that are applied sequentially to a symbolic transition system.

C.3 Technical issues

Following is a list of some significant technical issues that may need to be resolved.

- Several bugs cause the prototype to terminate prematurely. The prototype source code contains 93 different `STS` models and listing C.2 shows how much of these cause of the four important applications to crash. To obtain the list of these troublesome model files, execute `dberk/bin/crashedfiles`.
- The version of Axini::STSParser that is used by the prototype is outdated, but the newer versions have a different interface. The prototype cannot benefit from the more recent bugfixes until it it modified to work with the most recent interface.
- Different from what was documented in section 5.7, the DVE restructuring transformation inserts a single *tmp* process variable for every received interaction variable. This needs to be altered so that only a single tmp variable is used for single-valued messages, or a set of tmp variable for multi-valued messages.

```
dberk/
├── bin/ :  Prototype binaries:  see section 5.3 for a list
├── lib/
│   ├── etc/ :  Büchi automaton generator, see section 4.2.1
│   ├── ontology/ :  All important class definitions from which the
│   │   internal sts representation is constructed
│   ├── printer/ :  Printer code for DVE, STS and raw
│   ├── recipes/ :  All sts transformations and the recipestore
│   ├── sts-engine-interface/ :  Wrapper for Axini::STSParser
│   └── utils/ :  Utilities used during model closing, etc
├── spec/
│   ├── dberk/ :  rspec files for testing the prototype
│   └── models/ :  Models that are used by the rspec unit tests
├── test/ :  Some STS models used for testing
└── Gemfile :  Configuration file for the package dependencies
```

Figure C.1: Directory tree for the prototype source code.

command	**crashing STS files**
sts2sts	0/93
sts2dve	0/93
sts2raw	0/93
closemodel	22/93

Table C.2: The number of crashing `STS` models for each command.

- Different from what was documented in section 5.8.1, the implemented model closer scope test doesn't check for the datatype of the variables. This module needs to be altered so that an exception is thrown when a model has non-$\mathbb{N}_0$ variables.
- The implemented bisimilarity minimization algorithm is too slow. Use the Paige-Tarjan implementation and rewrite in C if still necessary.
- Many of the transformations that are contained within the recipes are too slow. Locate the bottlenecks and either rewrite the algorithms, or implement them in C.

C.4 Task assignment

- Fix the issues that are listed in sections C.3 and 7.1.1.
- Integrate the prototype into the Axini workflow by implementing the following features
  - Add support for LTL and CTL formulas. See section C.4.1 for a starting point on how this feature could be implemented.
  - Translation of SUT requirements to LTL or CTL formulas.
  - Parse the output of the modelchecker. If LTSmin finds a counterexample, then the counterexample need to be generated and parsed. See listing 6.1 for the relevant instructions.
  - Utilize multicore or distributed machines by using `dve2lts-mc` or `dve2lts-dist` when such hardware is available.
  - Utilize symbolic state space representation when the model is too large to be stored explicitly with `dve2lts-sym`.

C.4.1 Support for LTL and possibly CTL

The LTL formulas that LTSmin accepts are denoted in a special syntax. For this syntax, see the `ltsmin-ltl` manpage. As LTSmin combines all processes into a single state vector representation, the variables that are referred to within an LTL formula are made to refer to the elements of the state vector. To prevent naming collision, their names are modified to include the name of their enclosing process. To extend the prototype with support for LTL formulas, these formulas need to be modified so that they are made to refer to the correct elements.

The current command that is used by the prototype, `dve2lts-seq`, only generates the state space, and optionally checks for deadlocks when it is passed the `-d` flag. This program needs to be replaced with `dve2lts-mc`, which exploits multiple cores and can check for deadlocks as well. To enable CTL formulas, `dve2lts-sym` needs to be used instead as `dve2lts-mc` doesn't seem to be able to handle CTL.