Reachable States for Symbolic Transition Systems

August 23, 2015

Master Thesis
BSc Kai Bakker
August 23, 2015
Univeristy of Amsterdam
Supervisors
Prof. Dr. J. van Eijck
Ir. R.G. de Vries

Contents

Abstract

We will introduce an algorithm to calculate all reachable states for Symbolic Transition Systems. In Symbolic Transition System variables and constraints determine if transitions can be used, this means that the paths and valuations we chose previously influence which paths we can use later on. Currently there is no good techique to decide what path will take you to a certain point in the system. That is why we developed an algorithm to calculate all reachable states and the paths that lead to these states for determinist Symoblic Transition Systems.

1 Introduction

Our society has become more and more dependent upon computer systems and their correctness. At the same time these systems have become more complex. While malfunctioning websites and apps are annoying, there are an increasing amount of computer-systems where our lives dependent upon. A small failure in airplanes or medical devises could potentiality cost lives.

Axini is a company in Amsterdam with a product called TestManager, which is a tool for automated testing. They are specialized in testing through a Symbolic Transition System (STS), which is a formal description of a system to validate its functionality and correctness. The main advantage of STSs is that it enables the use of data and logical constraints to describe more complex systems.

As an example for how these STS work, lets use a coffee machine. In the first state the user has to put in a coin from a certain value. In the next state the machine will make you coffee if the value is equal to 50 cents, otherwise it will give you your coin back. To test if the coffee machine works we would like to activate the machine to make coffee. Currently TestManager is not able to understand that it should put in a coin of 50 cents to test if the machine is able to make coffee.

If we would try to cover every possible input, we would test the whole system. But for most STSs there is an infinite amount of possible inputs, and the test would never finish. To go back to the coffee machine example, imagine that there exists a coin for every possible value. A reasonable compromise is to create a test set that only lead to all different parts of a system. In our example, getting coffee and getting the coin back.

In this research we will define reachable states for STSs. Create an algorithm that finds all reachable states and proposes a path to these states. We will implement this algorithm in TestManager. Such an algorithm could be a valuable tool as it would make it possible to automaticaly create testruns that test all different parts of a system. We think a navigation system for STSs would be a welcomed by the Axini and the research field.

2 Research Question and Methodology

In this chapter we will shortly explain our research questions and motivation behind these questions. We will also describe our approach for answering these questions in a scientific way.

- Could we create a theoretical framework for reasoning about reachability for STSs?

Currently there is no formal definition of reachability for STSs, we will create a sound definition of reachability from a closly related formalisms like LTSs. Furter we need to redefine constraints and update functions for STSs.

- Could we create an algorithm to calculate all the reachable states and the traces to these states for a given STS?

With these definitions we would like to find an algorithm to calculate all the reachable states. Preferably an algorithm that executes in reasonable time, and one that works for all or most STSs.

- Could we create a testrun selection strategy using such an algorithm?

As the last step we would like to make sure our algorithm has value in practice, to do this we would like to use the algorithm to make a better test set selection strategy for TestManager.

Now we will shortly state some of the most important literature that we have been reading. We did a literature review where we read the work of Clark [1] about Model Checking and Abstractions. And we read about the foundational theory behind LTSs [2] and STSs [4] by Tretmans and Franzen, where we learned the differnces and the connections between te two formalisms.

3 Theoretical Framework

In the first part of this chapter we will define a constraint grammar and update function, that we need to give a formal definition of STSs, and we will discuss the valuation cycle of transitions.

In this chapter we will explain the most important theory of STSs, combined with some definitions that are specificly made for this research. Such as the constraint grammars and an update function. Before we go into the depth of these formal definitions, we will first show a figure that describes the STS of a coffee machine. We will referer to this STS in the of the theoretical framework, as we are going to formaly define the concept of STSs.

$$\begin{tikzcd}
 & l_0 \arrow[d, "\begin{array}{c} coin \\ {[x \geq 0]}; \\ total := x \end{array}"] & \\
 & l_1 \arrow[dl, "\begin{array}{c} coffee \\ {[total = 50]}; \\ total := 0 \end{array}"'] \arrow[dr, "\begin{array}{c} return \\ {[total \neq 50]}; \\ total := 0 \end{array}"] & \\
l_2 & & l_3
\end{tikzcd}$$

Figure 1: Coffee machine

Before giving formal definitions, we will explain some of the terms that we use in STS theory. We will use the term in combination with the value for the STS in Figure 1. Every STS is created with the following elements: a set of locations $L = \{l_0, l_1, l_2, l_3\}$, a set of location variables $\mathcal{V} = \{total\}$, a set of interaction variables $\mathcal{I}$ =$\{x\}$, a start state (l_0 with $total = 0$) and a set of transitions $\{coffee, coin, return\}$. A state is described by a location and a valuation, like the start state. When we move from a state over a transition, we have to choose certain values for the interaction variables so that the constraint of the transition is *true*. We end up in a new state with the updated valuation. For example when moving from l_0 to l_1, we have to choose a value for x, so that the constraint ($x \geq 0$) holds for example $x = 10$. Then the state gets updated and we are at location l_1 with valuation $total = 10$.

We call the the set of location variables $\mathcal{V}$ and the set of interaction variables $\mathcal{I}$. Notice that these sets do never overlap: $\mathcal{V} \cap \mathcal{I} = \emptyset$. We call the set of combined variables $\mathcal{W} = \mathcal{V} \cup \mathcal{I}$. Throughout this paper we will define the domain $\mathcal{D} = \{0..n\}$ for a certain $n \in \mathbb{N}$, but it can be defined as any other finite domain.

3.1 The constraint grammar

We will start creating a well defined constraint grammar, in this grammar we will define constraints for transitions. We will also define evaluation and substitution for this grammar, this gives us the power to reason about these constraints. In definition 3 we define a constraint grammar $\mathcal{F}$, which contains operators, variables and constants.

Definition 1. Constraint grammar $\mathcal{F}$ is defined by the set of nonterminals $\{H, E\}$, the set of terminals $\mathcal{D} \cup \mathcal{W}$, the start symbol H and the following production rules, where $c \in \mathcal{D}$ and $w \in \mathcal{W}$

$$H \rightarrow H \vee H | H \wedge H | \neg H | E = E | E > E$$
$$E \rightarrow w | c | E + E$$

We can only evaluate a constraint when we substitute the variables for values, that is why we will define syntactic variable substitution for $\mathcal{F}$.

Definition 2. Variable substitution for $\mathcal{F}$ is denoted as $\mathcal{F}_x^e$ for constant $c \in \mathcal{D}$, variables $x, w \in \mathcal{W}$, expressions $e, e', e" \in E$ and $H_1, H_2 \in H$.

$$\begin{aligned}
c_x^e &::= c \\
w_x^e &::= \begin{cases} w & \text{if } x \neq w \\ e & \text{if } x = w \end{cases} \\
(e + e')_x^{e"} &::= (e)_x^{e"} + (e')_x^{e"} \\
(e = e')_x^{e"} &::= (e)_x^{e"} = (e')_x^{e"} \\
(e > e')_x^{e"} &::= (e)_x^{e"} > (e')_x^{e"} \\
(\neg H_1)_x^e &::= \neg(H_1)_x^e \\
(H_1 \wedge H_2)_x^e &::= (H_1)_x^e \wedge (H_2)_x^e \\
(H_1 \vee H_2)_x^e &::= (H_1)_x^e \vee (H_2)_x^e
\end{aligned}$$

Lets take a simple example of a variable substitution on a constraint.

Example. We are going to substitute $(x \leq 10 \wedge x \neq 3)_x^5$

$$\begin{aligned}
(x \leq 10 \wedge x \neq 3)_x^5 &= (x \leq 10)_x^5 \wedge (x \neq 3)_x^5 \\
&= 5 \leq 10 \wedge 5 \neq 3
\end{aligned}$$

We can also define a way to evaluate a constraint, for a given valuation $g : \mathcal{W} \to \mathcal{D}$.

Definition 3. The evaluation of $\mathcal{F}$ is defined as $[\![F]\!]_g : \mathcal{F} \to (\mathcal{W} \to \mathcal{D}) \to \mathbb{B}$ with constraint $F \in \mathcal{F}$, valuation $g : \mathcal{W} \to \mathcal{D}$, constant $c \in \mathcal{D}$, variable $w \in \mathcal{W}$ and expressions $e, e' \in E$ and $H_1, H_2 \in H$.

$$\begin{array}{rcl}
[c]_g & ::= & c \\
[w]_g & ::= & g(w) \\
[e+e']_g & ::= & [e]_g + [e']_g \\
[\![e=e']\!]_g & ::= & [e]_g = [e']_g \\
[\![e>e']\!]_g & ::= & [e]_g > [e']_g \\
[\![\neg H_1]\!]_g & ::= & \neg[\![H_1]\!]_g \\
[\![H \vee H_2]\!]_g & ::= & [\![H_1]\!]_g \vee [\![H_2]\!]_g \\
[\![H_1 \wedge H_2]\!]_g & ::= & [\![H_1]\!]_g \wedge [\![H_2]\!]_g
\end{array}$$

We needed a helper function $[F]_g : E \to (\mathcal{W} \to \mathcal{D}) \to \mathcal{D}$ which is mapped to the whole domain, instead of only to *true* or *false*.

Often it will be interesting to calculate the set of all valuations where for the constraint is *true*.

Definition 4. The set of all valuations where for F is *true*: $[\![F]\!] : \mathcal{F} \to \mathcal{P}(\mathcal{D}^{\mathcal{W}})$ is defined as $[\![F]\!] ::= \{g \in \mathcal{D}^{\mathcal{W}} : [\![F]\!]_g\}$

Example. Here are the two basic examples, $[\![true]\!] = \mathcal{D}^{\mathcal{W}}$ and $[\![false]\!] = \emptyset$

We will now define valuation substitution and than we will prove that valuation substitution has the same results as variable substitution over $\mathcal{F}$. This proves that syntactic variable substitution is the same as semantic variable substitution.

Definition 5. Valuation substitution for $g : \mathcal{W} \to \mathcal{D}$ with variables $x, y \in \mathcal{W}$ and expression $e \in E$

$$g[e/x](y) ::= \begin{cases} g(y) & \text{if } x \neq y \\ e & \text{if } x = y \end{cases}$$

Lemma 6. *The substitution lemma for $\mathcal{F}$ with substitute variable $x \in \mathcal{W}$, expression $e \in E$, constraint $F \in \mathcal{F}$ and valuation $g : \mathcal{W} \to \mathcal{D}$*

$$[\![F]\!]_{g[[e]_g/x]} \Leftrightarrow [\![F^e_x]\!]_g$$

Proof. We will prove this lemma with induction, most of the equations have the same pattern. The first step is showing that all elements in $e \in E$ are substitutable.

$$[e]_{g[[e']_g/x]} \Leftrightarrow [e_x^{e'}]_g$$

E is defined as $E ::= w|c|E + E$, so the base cases are $w \in \mathcal{W}$ and $c \in \mathcal{D}$, we will prove those to be substitutable.

$$\begin{aligned}
[c]_{g[[e]_g/x]} &= c \\
&= c_x^e \\
&= [c_x^e]_g
\end{aligned}$$

To prove variable substitution over variables, we need to distinct two different cases: The case where $w = x$.

$$\begin{aligned}
[w]_{g[[e]_g/x]} &= g[[e]_g/x](w) \\
&= [e]_g \\
&= [w_x^e]_g
\end{aligned}$$

And the other case where $w \neq x$.

$$\begin{aligned}
[w]_{g[[e]_g/x]} &= g[[e]_g/x](w) \\
&= g(w) \\
&= [w]_g \\
&= [w_x^e]_g
\end{aligned}$$

We have had all the base cases, now we will follow with the inductive case. We want to prove that if $e, e' \in E$ are substitutable, than $e + e'$,$e = e'$,$e > e'$ are substitutable too, so we will assume:

$$[e]_{g[[e"]_g/x]} = [e^{e"}_x]_g \wedge [e']_{g[[e"]_g/x]} = [e\text{'}^{e"}_x]_g$$

Now we helps us prove the three cases in a similar fashion.

$$\begin{aligned}
[e + e\text{'}]_{g[[e"]_g/x]} &= [e]_{g[[e"]_g/x]} + [e\text{'}]_{g[[e"]_g/x]} \\
&= [e^{e"}_x]_g + [e\text{'}^{e"}_x]_g \\
&= [(e)^{e"}_x + (e\text{'})^{e"}_x]_g \\
&= [(e + e\text{'})^{e"}_x]_g
\end{aligned}$$

Now we proved by induction that every $e \in E$ is substitutable. We will now prove this also for F, we will repeat the proof of $e + e'$ for $e = e'$ and $e > e'$.

$$\begin{aligned}
[\![e = e\text{'}]\!]_{g[[\![e"]\!]_g/x]} &= ([e]_{g[[e"]/x]} = [e\text{'}]_{g[[e"]_g/x]}) \\
&= ([e^{e"}_x]_g = [e\text{'}^{e"}_x]_g) \\
&= [\![(e)^{e"}_x = (e\text{'})^{e"}_x]\!]_g \\
&= [\![(e = e\text{'})^{e"}_x]\!]_g
\end{aligned}$$

$$\begin{aligned}
[\![e > e\text{'}]\!]_{g[[\![e"]\!]_g/x]} &= ([e]_{g[[e"]_g/x]} > [e\text{'}]_{g[[e"]_g/x]}) \\
&= ([e^{e"}_x]_g > [e\text{'}^{e"}_x]_g) \\
&= [\![(e)^{e"}_x > (e\text{'})^{e"}_x]\!]_g \\
&= [\![(e > e\text{'})^{e"}_x]\!]_g
\end{aligned}$$

Proving $H_1 \wedge H_2$ and $H_1 \vee H_2$ is also identical to the previous proofs. When we assume that H_1 and H_2 are substitutable we can prove the following.

$$\begin{aligned}
[\![H_1 \wedge H_2]\!]_{g[[\![e]\!]_g/x]} &= ([\![H_1]\!]_{g[[e]_g/x]} \wedge [\![H_2]\!]_{g[[e]_g/x]}) \\
&= ([\![(H_1)^e_x]\!]_g \wedge [\![(H_2)^e_x]\!]_g) \\
&= [\![(H_1)^e_x \wedge (H_2)^e_x]\!]_g \\
&= [\![(H_1 \wedge H_2)^{e"}_x]\!]_g
\end{aligned}$$

Than there is the last operator, where we want to prove the $\neg H$ case, assuming H is substitutable, we get.

$$\begin{aligned}
[\![\neg H]\!]_{g[[e]_g/x]} &= 1 - [\![H]\!]_{g[[e]_g/x]} \\
&= 1 - [\![H^e_x]\!]_g \\
&= [\![(\neg H)^e_x]\!]_g
\end{aligned}$$

With these results we have proven that every expression in $e \in \mathcal{F}$ is substitutable. $\square$

3.2 The update function

We will now explain and define the update function for transitions, an update function $U : \mathcal{V} \rightarrow E$ is a function that maps every location variable to an expression. The expressions describes the new valuation of the location variable in the new state. This process is formaly described in Definition 7.

Definition 7. Updated valuation $g_U : \mathcal{V} \rightarrow \mathcal{D}$ for update function $U : \mathcal{V} \rightarrow E$ and old valuation $g : \mathcal{W} \rightarrow \mathcal{D}$

$$g_U(v) ::= [U(v)]_g$$

Example. Lets take the following example, where the update function of a transition is $x = x + 10$. Then $U(v) = \begin{cases} x + 10 & \text{if } x = v \\ v & \text{if } x \neq v \end{cases}$, with the following calculation we calculate the new valuation for g_U.

$$g_U(v) ::= [U(v)]_g = \begin{cases} [x+10]_g & \text{if } x = v \\ [v]_g & \text{if } x \neq v \end{cases} = \begin{cases} g(x)+10 & \text{if } x = v \\ g(v) & \text{if } x \neq v \end{cases}$$

3.3 Symbolic Transition Systems

A STS is a mathematical formalism that describes a model of a system. The System Under Test (SUT) is tested by comparing the expected behavior of the STS with the real behavior of the SUT. STSs are an extension of the more widely used Labeled Transition Systems (LTS) below you will find the definition of LTSs from Tretmans [2].

Definition 8. A Labelled Transition System (LTS) is a tuple $\mathcal{L} = \langle S, s_0, \Sigma, \rightarrow \rangle$ where S is a set of states and $s_0 \in S$ is the initial state, the set Σ is a set of observable action labels and $\rightarrow \subset S \times \Sigma \times S$ is the set of transitions.

STSs are more complex than LTSs and support modeling state behavior of data and first order logic. Modeling the control flow of data-intensive systems is possible due to transitions which are equipped with a constraint and an update function. The constraint acts as guard and is an element from the grammar $\mathcal{F}$. The update function U makes it possible to update location variables when moving over a transition.

Our definition of STSs is based on the definition of Frantzen and Tretmans [4].

Definition 9. A Symbolic Transition System is a tuple $\mathcal{S} = \langle L, \mathcal{V}, \mathcal{I}, s_0 \rightarrow \rangle$:

– L is a countable set of locations

– $\mathcal{V}$ is a set of location variables.

– $\mathcal{I}$ is a set of interaction variables, disjoint from $\mathcal{V}$.

– $s_0 = \langle l_0, \iota \rangle \in L \times \mathcal{D}^{\mathcal{V}}$ is an initialization of the start state.

– $\rightarrow \subseteq L \times \mathcal{F} \times \mathcal{P}(\mathcal{W})^{\mathcal{V}} \times L$ is the transition relation. So for a transition $\langle l, F, U, l' \rangle \in \rightarrow$ with F is referred to as the constraint and U as the update function.

We will only do research into STSs with deterministic behavior, because of the limited time available. This means that we are not working with input-output STS.

Now that we defined STSs, we will name certain aspects of a STS that we will use throughout this paper.

3.4 Transitions

We have seen the definitions of different parts of STSs, constraints, update function and transitions. In this section we will describe how all these parts come together.

First of all, it is important to notice that there are different type of valuations, we use valuations for location variables $\mathcal{V}$, interaction variables $\mathcal{I}$ or both variables $\mathcal{W}$. Combining two valuations can be done with the following definition.

Definition 10. The combined valuation $g \dot\cup f : \mathcal{V} \cup \mathcal{I} \to \mathcal{D}$ for location valuation $g : \mathcal{V} \to \mathcal{D}$ and interaction valuation $f : \mathcal{I} \to \mathcal{D}$ is

$$(g \dot\cup f)(w) ::= \begin{cases} g(w) & \text{if } w \in \mathcal{V} \\ f(w) & \text{if } w \in \mathcal{I} \end{cases}$$

Now we can formaly define what moving from a state over a transition, by combining all the theory we learned in the rest of this chapter.

Definition 11. Moving from a state $s = \langle l, g \rangle$ over transition $t = \langle l, F, U, l' \rangle$ results in a set of states

$$s \xrightarrow{t} ::= \{\langle l', (g \dot\cup f)_U \rangle |, f \in \mathcal{D}^{\mathcal{I}}, [\![F]\!]_{g \dot\cup f}\}$$

We can extend this definition, for moving from a set of states instead of from one state.

Definition 12. Moving from a set of states S over transition t

$$S \xrightarrow{t} ::= \bigcup^{s \in S} s \xrightarrow{t}$$

We have now defined everything around STSs, so that we can dive into the concept of reachable states in the next chapter.

4 Calculating Reachability

In this chapter, first we will describe a definition of reachable states and we will explore what makes calculating all reachable states hard. In the second part we will present an algorithm that calculates all the reachable states for a given STS and we will prove its correctness.

4.1 Definition of reachable states

To calculate all reachable states, we will first explore what a reachable state is and explore methods how to calculate all reachable states by hand. We could say that a state is reachable if there exists a trace and a corresponding valuation to that state. Tretmans [2] gave the following definition for the set of all reachable states for LTSs.

Definition 13. All reachable states from state $p \in S$ for LTS $\mathcal{L} = \langle S, s_0, \Sigma, \rightarrow\rangle$

$$der(p) ::= \{p' | \exists \sigma \in \rightarrow^*: p \overset{\sigma}{\Rightarrow} p'\}$$

We get the following definition if we extend Tretmans approach for STSs.

Definition 14. All reachable states from state $s \in L \times \mathcal{D}^{\mathcal{V}}$ for STS $\mathcal{S} = \langle L, l_0 \mathcal{V}, \iota, \mathcal{I}, \rightarrow\ \rangle$

$$der(s) ::= \{s' | \exists \sigma \in \rightarrow^*: s' \in (s \overset{\sigma}{\Rightarrow})\} = \bigcup^{\sigma \in \rightarrow^*} (s \overset{\sigma}{\Rightarrow})$$

Calculate all reachable states for a STS $\mathcal{S} = \langle L, \mathcal{V}, \mathcal{I}, s_0 \rightarrow\ \rangle$ is the equivalent to calculating $der(s_0)$, for obvious reasons. Now we will show how we calculate all reachable states for a given STS.

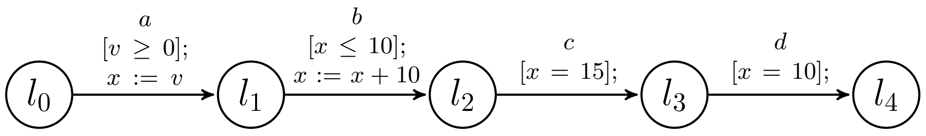

Figure 2: Example 15

Example 15. Assume the STS as found in Figure 1 with location variables $\mathcal{V} = \{x\}$, interaction variables $\mathcal{I} = \{v\}$ and start states $\{s_0\} = l_0 \times [\![x = 0]\!]$

To calculate all reachable states for this STS. We need to calculate all valid traces. For every trace we calculate the constraint on x for that states.

Trace	Location	Constraint on x	Resulting states
ϵ	l_0	$x = 0$	$l_0 \times [\![x = 0]\!]$
a	l_1	$x \geq 0$	$l_1 \times [\![x \geq 0]\!]$
ab	l_2	$10 \leq x \leq 20$	$l_2 \times [\![x \geq 10 \wedge x \leq 20]\!]$
abc	l_3	$x = 15$	$l_3 \times [\![x = 15]\!]$
$abcd$	l_4	$false$	$\emptyset$

Notice that there is no valuation for location l_4, this is because the constraints of transitions c and d contradict. All reachable states are the conjunction of the resulting states.

$$der(s_0) = l_0 \times [\![x = 0]\!] \cup l_1 \times [\![x \geq 0]\!] \cup l_2 \times [\![x \geq 10 \wedge x \leq 20]\!] \cup l_3 \times [\![x = 15]\!]$$

This is a simple but illustrative example that shows how we compute all reachable states for a given STS. Notice that this example does not contains any cycles, for a STS with cycles it is important to distinguish which traces could potentially lead to unvisited states and which will only find known states.

4.2 A reachability algorithm using LTSs

In this short section we will describe an algorithm for calculating all reachable state, by converting the STS to an LTS. Frantzen and Tretmans [4] described how we can transform every STS into an LTS. We will not go into the details of this transformation, but it is important to know that at the end of the transformation we have an one to one mapping from every state in the STS and the LTS.

How can we calculate all reachable states for an LTS? An LTS is not very different from a standard transition system and does not have complex elements like constraints and update functions. To calculate all reachable states we could use a standard graph-search algorithm to visit all reachable states. If we would map the found states in the LTS to their corresponding state in a STS, and so we would find all reachable states for a STS.

The reason why this approach is unusable, is because STSs often have an infinite or very big number of states. If we could have solved this problem by converting STS to LTS, there would be little reason to use STS at all.

4.3 The reachability algorithm

In this chapter we are going to define an algorithm to calculate all reachable states and prove its correctness and termination. It has the structure of a graph search algorithm. The power of this algorithm is that it can make computations about infinite groups of states instead of single states. Much like STS can describe an infinite number of transitions in one transition, where an LTS only describes that one transitions.

Below you will find the pseudocode of Algorithm 1 we will be using.

```
**Data**: Given a STS $\mathcal{S} = \langle L, \mathcal{V}, \mathcal{I}, s_0, \rightarrow \rangle$
**Result**: A finite set of traces that cover all reachable states
$C = \{\epsilon\}$
$M = \emptyset$
while $C \neq \emptyset$ do
    select $\sigma \in C$
    $C := C \setminus \{\sigma\}$
    foreach $t \in \rightarrow$ do
        if $(s_0 \overset{\sigma t}{\Rightarrow}) \not\subseteq \bigcup^{m \in M, target(m)=target(\sigma)} (s_0 \overset{m}{\Rightarrow})$ then
            $M := M \cup \{\sigma t\}$
            $C := C \cup \{\sigma t\}$
        end
    end
end
return $M$
```

Algorithm 1: Calculating a set of traces that cover all reachable states

This algorithm returns a finite set of traces that covers all reachable states, we will describe what happens step by step.

- It initializes two sets of traces, with the current traces C and the marked traces M.
- Every iteration, it selects a trace from C. It will select all the transitions for which it is useful to extend the trace. This means that we will only add new traces to C if . These traces will be added to M and C.
- The algorithm is finished when C is empty and will return a finite set of traces that cover all reachable states.

$$(s_0 \overset{\sigma t}{\Rightarrow}) \not\subseteq \bigcup^{m \in M, target(m)=target(\sigma t)} (s_0 \overset{m}{\Rightarrow})$$

We will give some special attention to the following constraint that we use in the algorithm, here new transitions are selected to add to the trace σ. For every transition we evaluate if it leads to unvisited states, by calculating all already found states per location and comparing it with all the states we will find following the new trace.

$target(\sigma)$ returns the target state of the last transition of σ, this way we make sure that we only use traces that lead to the state we talked about. Now N only contains traces which contain unvisited states.

Notice that this algorithm does not return all reachable states, but instead returns a set of traces that cover all reachable states. There are 2 reasons why we made this decision, first the set of all reachable states can be infinite. Second this makes it easy to calculate which trace led to a specific state. The following formula helps us find all traces that lead to state p, $\{m | m \in M, p \in (\iota \overset{m}{\Rightarrow})\}$.

All reachable states can be found by calculating $\bigcup^{m \in M} : (s_0 \overset{m}{\Rightarrow})$, where M is the output of the algorithm.

We will now prove the that Algorithm 1 behaves correctly.

Proposition 16. *The set of traces M that Algorithm 1 returns covers all reachable states for STS $\mathcal{S} = \langle L, \mathcal{V}, \mathcal{I}, s_0, \rightarrow \rangle$*

$$\bigcup^{m \in M} : (s_0 \overset{m}{\Rightarrow}) = der(s_0)$$

Proof. First we will prove the statement from right to left: M is a subset of all traces $\leftarrow^*$, so $\bigcup^{m \in M} : (s_0 \overset{m}{\Rightarrow}) \subset \bigcup^{\sigma \in \leftarrow^*} (s_0 \overset{\sigma}{\Rightarrow}) = der(s_0)$. Now we want to prove the statement from left to right, we could do this by proving that:

$$s_n \in der(s_0) \Longrightarrow s_n \in \bigcup^{m \in M} : (s_0 \overset{m}{\Rightarrow})$$

Assume $s_n \in der(s_0)$, lets find a trace $\sigma = \sigma_1 \sigma_2 .. \sigma_n \in \leftarrow^*$ and a valuation for this trace. From here we compute all the states $s_0 .. s_n$ on this path, starting with s_0 and leading to the end state s_n. Now we start with the process by looking for the longest sub-trace of σ that can be found in M, so $\sigma_1 .. \sigma_k \in M$ which means $s_k \in \bigcup^{m \in M} : (s_0 \overset{m}{\Rightarrow})$. If $k = n$ we are finished, otherwise the algorithm did not accept σ_{k+1} in M, the only reason why it did not accept σ_{k+1} is because there is an alternative trace $\sigma' \in M$ which does already contain s_{k+1}. This means that $s_k \in \bigcup^{m \in M} : (s_0 \overset{m}{\Rightarrow})$. Now we will start looking for the longest trace of the following form $\sigma' \sigma_{k+2} .. \sigma_{k'} \in M$. We will repeat this process till we found an $m \in M$ for which $s_n \in (s_0 \overset{m}{\Rightarrow})$, this means we did prove $s_n \in der(s_0) \Longrightarrow s_n \in \bigcup^{m \in M} : (s_0 \overset{m}{\Rightarrow})$. And so we can conculde that all reachable states are covered by M.

$$\bigcup^{m \in M} : (s_0 \overset{m}{\Rightarrow}) = der(s_0)$$

□

We did prove that Algorithm 1 returns a set of traces that cover all reachable states, and thereby works as expected. Now we will prove that the algorithm terminates.

Proposition 17. *Algorithm 1 terminates for every STS $\mathcal{S} = \langle L, \mathcal{V}, \mathcal{I}, s_0, \rightarrow \rangle$ with a finite number of states.*

Proof. For every trace σ added to M we know that $(s_0 \overset{\sigma}{\Rightarrow}) \not\subset \bigcup^{m \in M, target(m)=target(\sigma)}(s_0 \overset{m}{\Rightarrow}$), in other words: there exist a state $s = \langle target(\sigma), g\rangle$, forwhich $g \in (s_0 \overset{\sigma}{\Rightarrow})$ and $g \notin \bigcup^{m \in M, target(m)=target(\sigma)}(s_0 \overset{m}{\Rightarrow})$. So we could make a mapping where every trace that is added to M is mapped to the unique state s, next iteration s cannot be mapped to another trace because σ will be in M. When every element in M can be mapped to a unique state, and the set of states in finite: M is finite too. If M is finite the number of iterations is finite too, so Algorithm 1 terminates. □

We proved that the algorithm terminates, but that does not mean that it terminates fast or within a reliable time frame. This is not easy to solve, because the slow execution time can be because of different reasons. There are a lot of differnt types of models, some contain a lot of transitions, others contain repeating loops. There is not one type of model that takes a long time to execute. A simple measure to rejuce execution time for a certain type of model, is by setting some constraint on which the algorithm stops. This implies a tradeof between finding every state and the execution time of the algorithm. In this case we will not find every state, but finding most states can still be very valuable.

One approach we would like to propose would be setting a maximal trace length, instead of finding all reachable states it would find all states that are reachable in less than n steps. Most testrun selection strategies in TestManager have a maximal amount of steps per testrun as well. This could make the algorithm terminate much earlier for certain models. In Algorithm 2 we describe a possble implementation for such an algorithm.

```
Data: Given a STS $\mathcal{S} = \langle L, \mathcal{V}, \mathcal{I}, s_0, \rightarrow \rangle$ and a max trace length $n$
Result: A finite set of traces that cover all reachable states
$C = \{\epsilon\}$
$M = \emptyset$
while $C \neq \emptyset$ do
    select $\sigma \in C$
    $C := C \setminus \{\sigma\}$
    if $|\sigma| \leq n$ then
        foreach $t \in \rightarrow$ do
            if $(s_0 \overset{\sigma t}{\Rightarrow}) \not\subset \bigcup^{m \in M, target(m) = target(\sigma)} (s_0 \overset{m}{\Rightarrow})$ then
                $M := M \cup \{\sigma t\}$
                $C := C \cup \{\sigma t\}$
            end
        end
    end
end
return $M$
```

Algorithm 2: *ReachableStates* algorithm with maximal trace length

5 Implementation

In this first part of this chapter we will show how we implemented Algorithm 1 that was described in the previous chapter. We will mainly explain some of the more complex parts of the implementation. We will lead you through the problems we had to solve and solutions we came up with. In the second part we will implement the *DataReachabilityAlgorithm* as a testrun selection strategy that uses Algorithm 1, to show the value of the algorithm as a tool for *TestManager*.

Axini's main product is called *TestManager*, it is written in *ruby* and has a wide range of responsibilities. In *TestManager* models are written in a DSL and converted to STSs, from which we can start testing and run test sets. For our implementation we will often need to manipulate and solve constraints. *Axini* has its own constraint language called *Dumont*, this is a useful tool to modify constraints. It has the buildin functionality to covert these constraints into Prolog constraint and solve them with Prolog.

5.1 Implementing the reachability algorithm

To implement Algorithm 1 we created a new class called *Trace*, which contains data about the start state, the transitions and the constraint that describe the valuations the trace lead to. The *Trace* class contains functionality to create and update constraints from transitions, and the functionality to compute if one constraint is a subset of the other. This section will be mainly about the challenge how to calculate if one constraint forms a subset of another constraint.

Determining if the evaluation of one constraint is a subset of the other was the most complex part of the implementation. To do this we will first describe a method of converting a trace of transitions into a constraint that can be solved by a solver. In the second part we describe how we can calculate if one constraint forms a subset of another and in the last part we describe a problem we found with this approach and describe a method of fixing this problem.

We used the following method to convert a start state and a trace into a constraint. The constraint that we create should be evaluated to true for any valuation that is reachable by this state. We will show how this method works by converting trace *abc* from Figure 2 to a constraint.

Example 18. In a table we list the transtions with their constraint and update function. On the right side of the table we describe the new constraint, as would be generated by our implementation. The new constraint is created by combining the constraint and the update function from a transition. a giving all the variables that are evaluated an index i and all the variables that are updated index $i+1$.

Transition	Constraint	Update	New constraint
a	$v \geq 0$	$x := v$	$v_1 \geq 0 \wedge x_2 = v_1$
b	$x \leq 10$	$x := x + 10$	$x_2 \leq 10 \wedge x_3 = x_2 + 10$
c	$x = 15$	-	$x_3 = 15 \wedge x_4 = x_3$

When we computed all the new constraints we can make the conjunction of these constraints together with the valuation of the start state $x_1 = 0$, this will give us the following combined constraint:

$x_1 = 0 \wedge v_1 \geq 0 \wedge x_2 = v_1 \wedge x_2 \leq 10 \wedge x_3 = x_2 + 10 \wedge x_3 = 15 \wedge x_4 = x_3$.

If we send this to the *prolog* solver it will return a solution where $x_4 = 15$.

The approach as described in Example 18 is usable in for every trace. When we are able to convert every transition to a constraint we could use the following proposition to compute if $[\![F']\!] \subset [\![F]\!]$.

Proposition 19. *For constraints $F, F' \in \mathcal{F}$*

$$[\![F']\!] \subset [\![F]\!] \Longleftrightarrow \nexists g \in \mathcal{D}^{\mathcal{V}} : [\![\neg F \wedge F']\!]_g$$

With Proposition 19, we can combine the constraints F and F' into a new cosnntraint $\neg F \wedge F'$. We can send this new constraint to the *prolog* solver, the solver will search for a solution of the constraint or say that there is no solution. That is basicly the same as: $\nexists g \in \mathcal{D}^{\mathcal{V}} : [\![\neg F \wedge F']\!]_g$. So via this method we can use the solver to compute if $[\![F']\!] \subset [\![F]\!]$.

But when we implemented this, we found that we made a mistake; when F is generated from a trace it contains not only location variables, but also interaction variables and variables with extra indexes like we showed in Example 18. Lets explore the following example to see why goes wrong.

Example 20. Lets asume $\nexists g \in \mathcal{D}^{\mathcal{V}} : [\![\neg F \wedge F']\!]_g$ to be *true* for constraint $F = (x_1 = 0 \wedge v_1 \geq 0 \wedge x_2 = v_1 \wedge x_2 \leq 10 \wedge x_3 = x_2 + 10)$, if we ask the solver to solve $\neg F$, than it could find a valuation where $x_1 = 1$. In constraint F' there is no x_1, so there are no new constraints on F''s valuation. In other words $\nexists g \in \mathcal{D}^{\mathcal{V}} : [\![F']\!]_g$ is *true* and F' does not have a solution.This implies that the only subset of F is F', which is wrong.

Now we will suggest a solution to this problem, we did not implement it ourself, because there was too little time. The problems that we describe in the previous paragraph could be solved by getting rid of all variables that are not part of the current location variables. This would mean simplifying every constraint to a constraint with only the location variables.

Example 21. The following constraint $x_1 = 0 \wedge v_1 \geq 0 \wedge x_2 = v_1 \wedge x_2 \leq 10 \wedge x_3 = x_2 + 10$, could be simplified to only contain variable x_3, the new constraint would become $10 \leq x_3 \wedge x_3 \leq 20$. All the other information is not necessary to describe the valid valuation.

If we would have a function that would simplify the constraint so that it only would contain variables in $\mathcal{V}$. Then we can create the following proposition to calculate if one constraint forms a subset of the other constraint.

Proposition 22. *For grammar $\mathcal{F}$ where $\mathcal{W} = \mathcal{V}$ with constraints $F, F' \in \mathcal{F}$*

$$[\![F']\!] \subset [\![F]\!] \iff \nexists g \in \mathcal{D}^{\mathcal{V}} : [\![\neg F \wedge F']\!]_g$$

Proof. We will prove this via the following equivalent relations.

$$\begin{gathered}
[\![F']\!] \subset [\![F]\!] \\
\forall g \in \mathcal{D}^{\mathcal{V}} : [\![F']\!]_g \Rightarrow [\![F]\!]_g \\
\forall g \in \mathcal{D}^{\mathcal{V}} : \neg[\![F']\!]_g \vee [\![F]\!]_g \\
\forall g \in \mathcal{D}^{\mathcal{V}} : \neg([\![\neg F]\!]_g \wedge [\![F']\!]_g) \\
\forall g \in \mathcal{D}^{\mathcal{V}} : \neg([\![\neg F \wedge F']\!]_g) \\
\nexists g \in \mathcal{D}^{\mathcal{V}} : [\![\neg F \wedge F']\!]_g
\end{gathered}$$

□

We believe it would be pretty simple to create a function that would simplify the constraints, and would solve our problem with the current implementation and would drastically reduce the execution time.

Our current implementation reused most of Axini's functionality and we mainly added a new 170 lines long *Trace* class, that handels most of the conversion logics. The currenty implementation is not totaly functional, but we can certainly be used as a prototype as it was intended. Now we will show it's value by implementing a *DataReachabilityStrategy* to use the algorithm to improve test selection.

5.2 Implementing the *DataReachabilityStrategy*

A *Strategy* is a class in *TestManager* that is resposbile for selecting testruns on the fly. Currently most *Strategy*'s are not able to look ahead, they have a hard time solving STS like the the one in Figure 3. You could say that a *Strategy* walks through the model and chooses what the next transition should be. Most of *Axini*'s strategies in *TestManager* are focused on covering as many transitions as possible, so that these kind of tests have a big test coverage, which is generally accepted as important for testing.

One of the most important usecases for Algorithm 1 is using it as a tool to empower *Strategy*'s. The algorithm makes it easy to look ahead, by being able to calculate a trace that leads to a specific state. To showcase the value of the algorithm we created the *DataReachabilityStrategy*.

The the *DataReachabilityStrategy* is a farily simple prototype, we will explain how it works. In the first step it will calculate all reachable states via Algorithm 1.

For every new testrun the strategy will select the trace with the most uncovered transitions. The strategy will be finished when there are no more traces that contain uncovered transitions.

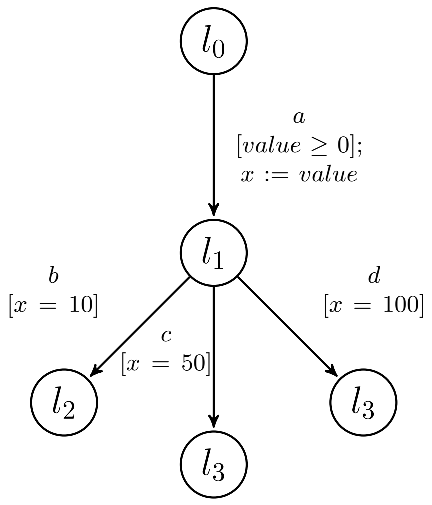

Figure 3: Tripod

We let the *DataReachabilityStrategy* test the STS from Figure 3. The strategy first selected the trace *ab*, then *ac* and and then *ad*. Then all the traces are covered and the strategy terminates. This is exactly how we expect a strategy to cover such a STS, and we could conclude that this is a great implementation for a strategy. The *DataReachabilityStrategy* is effective, fast and very easy to implement in a under 50 lines of code.

While the other strategy's in *TestManager* are not able to cover a STS like this fully. We believe that Algorithm 1 could be valuable in a lot of strategies. The Strategy at its current from is rather simple.

6 Analysis

In this chapter we will first show the results of the implementation of Algorithm 1 on different STSs. We research what future work is needed to make our theories useful. In the last part we will present the research questions and how well we solved these questions.

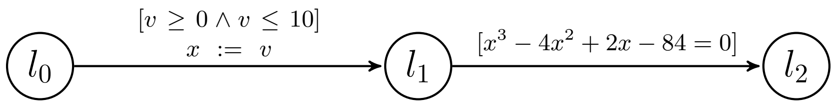

Figure 4: Cubic-formula STS

To analyze the results of the Algorithm 1, we will show the result for two small STSs. The first one is the STS of Figure 2, we calculated all reachable states with the algorithm. Which worked correctly and returned a trace of length 2 with valuation $v = 6$, which is the solution of constraint $x^3 - 4x^2 + 2x - 84 = 0$. This shows that the algorithm is able to calculate the solution for a cubic-formula.

We also tested similar STSs where we replaced the cubic-formula constraint for other constraints as shown in the table below. This table shows the different solutions for x when we let the Algorithm solve them, all the solutions are correct and expected.

Constraint	Solution	Correct
$x = 0$	$x = 0$	*true*
$x \geq 1$	$x = 1$	*true*
$x \leq 2$	$x = 0$	*true*
$x \neq 3$	$x = 0$	*true*
$x + 2 = 2 * x$	$x = 2$	*true*
$x = x * x - 6$	$x = 3$	*true*
$x = x * x - 5$	-	*true*
$x = 14$	-	*true*
$x = -10$	-	*true*
$x < 0 \vee x > 10$	-	*true*
$x \geq 0 \wedge x \leq 10$	$x = 0$	*true*

Now lets analyze how well the algorithm works for models with loops. We will have a look at the STS of Figure 3, that has two loops for the same location. The difference between the two, is that transition a can only be executed when x is even and transition b can only be executed when x is odd. So every trace should start with a and then alternate a and b. We will execute it with Algorithm 1 with a maximal trace length of 25 transitions.

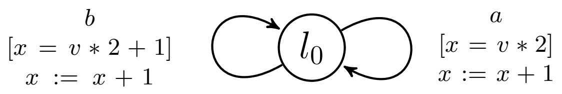

Figure 5: Loop STS

After 22.23 seconds the algorithm terminates, and shows the results as we expected above; 26 traces with alternating ab and with lengths 0 to 25. The results is correct, but it took a long time to calculate the traces for such a relatively simple STS. One of the reasons the algorithm is slow is because the constraints get big. For example to calculate if the longest trace (25 transitions) is not a subset of all the already found traces. We have to compare the constraint of the longest trace, with all the already found traces, because they are all from the same location. That means that the constraint is created from $26 * (26 - 1)/2 = 325$ transitions. For every transition we have a transition constraint, an update function and the glue between the transitions. In this case to check if the transition of length 25, contained a states that where not found in the other traces. We need to build a constraint of 2644 operators and different 650 variables, and that will all get send to the prolog solver. Here we see that for a simple STS witch contains a loop our algorithm needs $O(n^3)$ time, where n is the maximal number of traces. Because there are

We think that by simplifying the constraints to only contain the current location variables as described at the end of the previous chapter and storing a constraint for every location which describes all reachable valuations could make the algorithm significantly faster. We think that this same STS could be solved in $O(n)$ time. For now speed is a major problem of the current implementation of this algorithm and we see this the simplification improvement as an important part of the future work.

Another part of the future work we would like to suggest is nondeterminism. Nondeterminism is very important for STSs so it would be interesting to see if the algorithm and the theory around it that we developed can be extended for nondeterministic use cases.

Its time to go back to the research questions and see if we answered some questions. We created a useful theoretical framework to think about reachable states, which showed its value when we created and proved the Algorithm 1. We engineered the algorithm which finds all reachable states for a STS. The implementation is not ready for production, but we have some suggestions, how we could get to a point where it is ready for production. The algorithm showed its value as a tool when we created the *DataReachabilityStrategy*.

- Could we create a theoretical framework for reasoning about reachability for STSs?

We collected and build up a theoretical framework for constraint grammars and STSs as a whole. We created a sound definition of reachable states which was closely related to the definition of reachable states in LTSs.

- Could we create an algorithm to calculate all the reachable states and the traces to these states for a given STS?

With the definitions we made in for the previous question we engineered Algorithm 1 that calculates all the reachable states. We also implemented this algorithm in Axini's codebase, the result was an algorithm that worked for a subset of all STSs and is slow for bigger STSs. We proposed a solution for these issues, which should not be pretty easy to implement and which could make the algorithm a lot faster for many STSs.

- Could we create a testrun selection strategy using such an algorithm?

We created a test run selection strategy using Algorithm 1, inside TestManager. It is faily simple, and should not be used in production. But it could be easialy used to complement an already exisiting test strategy, that would use my algorithm when the original test is done, to reach some unreached transitions or states. The result is that with Algorihtm 1 it is easy to implement new strategy, that is able to solve problems that other strategies can not solve.

7 Concluding remarks

In Chapter 3 we created a constraint grammar and proved that syntactical and semantical variable substitution are the same. This was the start of the theoretical framework for reasoning about reachable states. In Chapter 4, we learned how to calculate all reachable states for a given STS. We engineered Algorithm 1 based on the graph search algorithm, which returns a set of traces that cover all reachable states. Then we proved its correctness and termination with the theoretical framework. In Chapter 5 we implemented the algorithm in Axini's TestManger. The resulting algorithm should be seen as a prototype that works on simple models, but is not ready to solve models with more complex structures, because we did not implement constraint simplication. Lastly we implemented a testrun selection strategy that uses the reachability algorithm for test selection.

We did create the necessary theory to research reachable states. We developed an algorithm that calculates all reachable states. The implementation of the algorithm is working for small models, but not ready for production. With this algorithm we where able to quickly create a well working testrun selection strategy, that has results that make it really easy to cover specific transitions or parts of the model.

We showed that reachable states theory is possible and practical for deterministic cases and could potentially be valuable for non-deterministic cases. Test selection in STS is was not able to look ahead in its test selection. We hope that in the future Algorithm 1 will be used as a tool, to make test selection more specialized and more specific.

Before this algorithm can be used for STSs, there should be some future research in the following two areas. First the algorithm should simplify its constraints, this could speed up calculations drastically and would make the implementation work correctly for a bigger set of STSs.

To make Algorithm 1 useful for all STSs, there should be done some research in the combination with non-deterministic STS, because non-determinism is important for STSs.

References

[1] E.M. Clark, O. Grumberg and D.E. Long, Model Checking and, Abstraction AVM Transactions on Programming Languages and Systems, Vol 16 NO. 5, Pages 1512-1542, 1994

[2] J. Tretman, Model based testing with labeled transition systems, Springer Berlin Heidelberg, Serie Volume 4949, Pages 1-38, 2008

[3] H. Hemmati, A. Arcuri, L. Briand, Reducing the Cost of Model-Based Testing through Test Case Diversity, Testing Software and Systems, Springer, 2010

[4] L. Frantzen, J. Tretmans, and T.A.C. Willemse, Test Generation Based on Symbolic Specifications, Formal Approaches to Software Testing, Springer, pages 1-15, 2005

[5] R.G. de Vries and J. Tretmans, Towards Formal Test Purposes, 2006

[6] F. Sietsma, A Case Study in Formal Testing and an Algorithm for Automatic Test Case Generation with Symbolic Transition Systems, 2009

[7] L. Bulwahn, Counterexample Generation for Higher-Order Logic Using Functional and Logic Programming

[8] A. Pretschner, T. Stauner, One Evaluation of Model-Based Testing and its Automation

[9] E. M. Clarke, O. Grumberg, K. L. McMillan and X. Zhao, Symbolic Counter-Example Generation for Model Checking

[10] T. Jéron, Symbolic Model-based Test Selection, Electronic Notes in Theoretical Computer Science, 2008