Trace Coverage Strategy for Symbolic Transition Systems

Ardavan Ghaffari[*]

August 22, 2016

Supervisors:	Prof. Dr. Jan van Eijck, Vincent de Bruijn
Host organization:	Axini, www.axini.com

Universiteit van Amsterdam
Faculteit der Natuurwetenschappen, Wiskunde en Informatica
Master Software Engineering
http://www.software-engineering-amsterdam.nl

[*]ghaffari.ardavan@gmail.com

Contents

Abstract

The research reported in this thesis was carried out at *Axini*, a company specialized in automated testing of software systems. *Axini's* main product is a *Model Based Testing* tool called *TestManager*. The underlying testing theory used in *TestManager* is *ioco*. According to this theory test cases are constructed from traces of the model obtained from the system's specification. The resulting test suite is often infinite because of the high number of combinations of data values and the presence of loops in the specification. Executing such a test suite is not practical since it cannot be done in a reasonable time. Our research has resulted in a *Trace Coverage Strategy* that creates a finite test set by bounding the model on the length of its traces and generating one test case for every trace in the bounded model. The resulting test set is then executed against the *SUT*. A reachability algorithm has been implemented that plays an important role in the generation and execution of test cases. Given a symbolic trace, the algorithm in conjunction with a constraint solver decides if the trace is ever reachable. The reachability algorithm can also be used in other areas of *Model Based Testing* such as static model checking. For example, it can be used to decide whether a model contains *deadlock* traces.

Trace Coverage is an off-line testing strategy that creates a test case for every reachable symbolic trace in the bounded model. Although the number of traces through the model becomes finite after having it bounded both in terms of loops and data, but it could still be a very large set. Test generation in *Trace Coverage Strategy* has been designed in a way that avoids extracting all the symbolic traces of length n and instead calculates the set of reachable traces recursively starting from the reachable traces of length one. This way the explosion resulted from extracting all the symbolic traces is avoided. *Trace Coverage* is capable of handling non-determinism in the models and tests with a minimum test suite. Traces that are non-deterministically followed together in a single test execution are identified and marked as covered.

The *coverage metric* that we are looking to achieve is 100 percent of all the reachable symbolic traces in the bounded model. If we achieve that then it means the *SUT* is *ioco*-conforming to its specification and if we don't its because of the failing traces in the *SUT*. Our strategy is able to detect them during testing.

Preface

I am very grateful to have been given the opportunity to do my Master thesis at Axini. I would like to thank *Machiel van der Bijl* and *Menno Jonkers* for that. I would like to show my gratitude to *Vincent de Bruijn* for spending numerous hours on discussing this project and sharing his advice and knowledge with me. I am grateful to *Jan van Eijck* as my supervisor at the University of Amsterdam for his support in writing this thesis.

I would like to thank my parents who have supported me throughout my studies. I am thankful to my brother-in-law *Amirhossein*, for his insightful advice and motivation. Last but not least, I am grateful to my sister *Irene*, for her love and support. Thank you for being there for me.

Chapter 1

Introduction

Testing is a major part of the software development process and, together with debugging, accounts for more than half of the development cost and effort [MSB11]. It is often a manual and laborious process without effective automation, which makes it error prone and time consuming [Tre08]. Testing, however, is an important activity during software development that needs to be carried out frequently. This is because it ensures a high quality for the software system and it also decreases future maintenance costs [McC96].

Model Based Testing (MBT) is a promising technique to improve the quality and effectiveness of testing, and to reduce its cost [Tre08]. The current state of practice is that test automation mainly concentrates on the automatic execution of tests, but that the problem of test generation is not addressed [Tre08]. MBT enables automation in generating test cases from a model. The model formally describes the specification of the System Under Test (SUT). The specification prescribes what the SUT should, and should not do. By extracting test cases from the model and executing them on the SUT, we can check whether the SUT is conforming to its specification. If the SUT conforms to its specification, then it is correct otherwise there is an error in the implementation of the SUT (assuming that the model of the specification itself is valid and correct).

To check whether the SUT conforms to its specification we need to know precisely what it means for a SUT to conform to a specification, i.e., a formal definition of conformance is required [Tre08]. Many different conformance relations have been proposed; see [BJK$^+$05, section 7] for an overview. A prominent example is ***ioco*** which is abbreviated from input-output conformance. According to ioco, an implementation conforms to a specification if, for each trace of the specification, the possible outputs after observing that trace in the implementation are also possible after observing that trace in the specification [VT]. The ioco testing theory has been implemented in several model based testing tools including Axini TestManager.

1.1 Motivation

In the ioco testing theory, test cases are constructed from suspension traces of the specification referred to as Straces(s), i.e., traces of the model obtained from the specification that may contain the quiescence action δ. δ is a special label which denotes the absence of output in a quiescent state in the model. It has been proved [Tre08] that Straces(s) is a complete test suite meaning that it is able to detect all the errors in the implementation (exhaustive) and that it does not make any false detection of errors (sound). Complete test suites are usually infinite and contain traces of unbounded length. Such test suites can never be executed within any reasonable limits of time and resources. We want to test input-output conformance with a finite test suite.

1.2 Research Goals

The motivation above brings us to the high level goals of this research:

- Define a finite subset of suspension traces of the specification that contains bounded-length test cases.
- Design, implement and validate a test strategy that can generate all the test cases in the finite subset and also fulfill all the requirements that are imposed upon it during both test generation and test execution. The requirements are introduced in chapter 3 where we present a full analysis of the problem.
- Integrate the new testing strategy into Axini TestManager.

1.3 Outline

This report has five more chapters: in chapter 2, we briefly review the formalism of the models used in this research as well as the notion of input-output conformance testing. In chapter 3, we provide an example driven analysis of the problem followed by the requirements to be fulfilled by our testing strategy. The research method and its execution are explained in chapters 4 and 5 respectively. Finally, conclusions are drawn and future work suggestions are made in chapter 6.

Chapter 2

Model Formalism and Framework

The starting point for Model Based Testing is an *implementation relation* that formally defines when a formal model representing the System Under Test conforms to a formal model constituting its specification [FTW06]. The ***ioco***-testing theory is a well-known formal approach to model based testing, which has been used extensively in various applications. This theory is based on the formalism of *Labelled Transition Systems* (LTSs) and a formal implementation relation called ioco defines conformance between implementations and specifications.

2.1 Labelled Transition Systems

A labelled transition system [Tre08] is a structure consisting of states with transitions, labeled with actions, between them. The states model the system states; the labeled transitions model the actions that a system can perform. The following definitions all have been taken from [Tre08] and [VT]:

Definition 1. A labelled transition system (with inputs and outputs) is a 5-tuple $\langle Q, L_i, L_u, T, q_0 \rangle$ where

- Q is a countable, non-empty set of states.
- L_i and L_u are the set of input and output labels or actions respectively, such that $L_i \cap L_u = \emptyset$. Outputs are actions initiated by the system, and inputs are actions initiated by the environment. In our case, TestManager will act as the environment. Inputs are usually decorated with '?' and outputs with '!'. We use L for $L_i \cup L_u$.
- $T \subseteq Q \times (L_i \cup L_u \cup \{\tau\}) \times Q$, with $\tau \notin L$, is the transition relation. The special label τ represents an internal unobservable transition.
- $q_0 \in Q$ is the initial state.

We write $q \xrightarrow{\mu} q'$ if there is a transition labeled μ from state q to state q', i.e., $(q, \mu, q') \in T$ and we use $q \xrightarrow{\mu}$ as an abbreviation for $\exists q' \in Q : q \xrightarrow{\mu} q'$. The labels in L represent the observable actions of the SUT; they model the system's interactions with its environment. Internal actions are denoted by the special label τ, which is assumed to be unobservable for the system's environment. Observable behavior of a system is captured by the system's ability to perform sequences of observable actions. Such a sequence of observable actions is obtained from a sequence of actions under abstraction from the internal action τ. If q_1 can perform the sequence of actions $a.\tau.\tau.b.c.\tau$ $(a, b, c \in L)$, i.e., $q_1 \xrightarrow{a.\tau.\tau.b.c.\tau} q_2$, then we write $q_1 \overset{a.b.c}{\Longrightarrow} q_2$ for the τ-abstracted sequence of observable actions. We say that q_1 is able to perform the ***trace*** $a.b.c \in L^*$ and it may end in state q_2. The use of may is important here: because of ***non-determinism***, it may be the case that the system can also perform the same sequence of actions but end in another sate: $q_1 \xrightarrow{a.\tau.\tau.b.c.\tau} q_3$ with $q_2 \neq q_3$. L^* is the set of all finite sequences over L, with ϵ denoting the empty sequence.

Definition 2. Let $s = \langle Q, L_i, L_u, T, q_0 \rangle$ be a labelled transition system with $q, q' \in Q$, $a, a_i \in L$, and $\sigma \in L^*$.

- $q \overset{\epsilon}{\Rightarrow} q' \iff q = q'$ or $q \xrightarrow{\tau} \ldots \xrightarrow{\tau} q'$
- $q \overset{a}{\Rightarrow} q' \iff \exists q_1, q_2 : q \overset{\epsilon}{\Rightarrow} q_1 \xrightarrow{a} q_2 \overset{\epsilon}{\Rightarrow} q'$
- $q \xRightarrow{a_1 \ldots a_n} q' \iff \exists q_0 \ldots q_n : q = q_0 \overset{a_1}{\Rightarrow} q_1 \overset{a_2}{\Rightarrow} \ldots \overset{a_n}{\Rightarrow} q_n = q'$
- $q \overset{\sigma}{\Rightarrow} \iff \exists q' : q \overset{\sigma}{\Rightarrow} q'$

We use the name of the labelled transition system or its initial state interchangeably, e.g., we write $s \overset{\sigma}{\Rightarrow}$ instead of $q_0 \overset{\sigma}{\Rightarrow}$. With this in mind, The set of traces of s is defined as traces$(s) = \{\sigma \in L^* | s \overset{\sigma}{\Rightarrow}\}$. We denote the set of states that are reachable from a given state q via trace σ as q **after** $\sigma = \{q' | q \overset{\sigma}{\Rightarrow} q'\}$ and by extension, given a set of states P: P **after** $\sigma = \bigcup\{q$ **after** $\sigma | q \in P\}$.

A set of actions A is refused by a set of states P if $\exists q \in P, \forall \mu \in A \cup \{\tau\} : \nexists q' : q \xrightarrow{\mu} q'$ and we write it as P **refuses** A. A state q is quiescent if $\{q\}$ **refuses** L_u. A quiescent state cannot perform any output action. Quiescence is denoted by a special output action δ ($\delta \notin L \cup \{\tau\}$). Let s_δ be the labelled transition system s to which we add a δ-loop transition $q \xrightarrow{\delta} q$ to all quiescent states, then the set of suspension traces of s is:

$$\text{Straces(s)} = \{\sigma \in L_\delta^* | s_\delta \overset{\sigma}{\Rightarrow}\}$$

where L_δ^* is the set of sequences of actions in $L \cup \{\delta\}$. From now on we will usually include δ-transitions in the transition relation, unless otherwise indicated.

Definition 3. We denote the class of all labelled transition systems over L_i and L_u as $LTS(L_i, L_u)$. If $\forall q \in Q, \mu \in L_i : q \overset{\mu}{\Rightarrow}$ we say that the system is input-enabled. The class of all input-enabled labelled transition systems over L_i and L_u is denoted by $IOTS(L_i, L_u) \subseteq LTS(L_i, L_u)$.

An example of a labelled transition system is given in figure 2.1. This LTS models the specification of a very simple coffee machine that dispenses free coffee. As can be seen from the figure, upon receiving the input ?coffee_button_pressed from the environment, the system will first respond with dispensing a cup and then dispensing coffee.

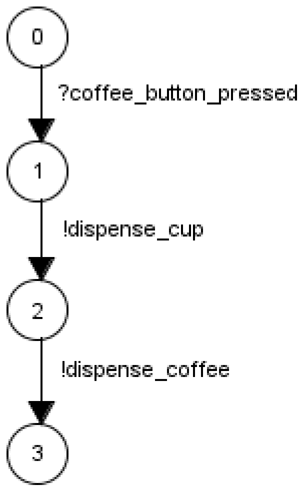

Figure 2.1: LTS modeling the specification of a coffee machine

2.2 The Implementation Relation *ioco*

The implementation relation on which Axini TestManager has built its test theory is ***ioco***. Informally, an implementation $i \in IOTS(L_i, L_u)$ is ioco-conforming to specification $s \in LTS(L_i, L_u)$ if any test case derived from s and executed on i leads to an output from i that is foreseen by s [Tre08]. Before giving a formal definition of ioco, we first need to define the set *out* of possible outputs. The following definitions all have been taken from [Tre08]:

Definition 4. Let q be a state in a transition system, and let Q be a set of states, then

- $out(q) = \{\mu \in L_u \mid q \xrightarrow{\mu}\} \cup \{\delta \mid \text{if } q \text{ is quiescent}\}$
- $out(q) = \bigcup \{out(q) \mid q \in Q\}$
- $out(q \textbf{ after } \sigma)$ gives all possible outputs occurring after having performed the trace $\sigma \in L_\delta^*$ starting from state q.

Definition 5. Given a set of input labels L_i and a set of output labels L_u, the relation $\textbf{ioco} \subseteq IOTS\,(L_i, L_u) \times LTS\,(L_i, L_u)$ is defined as follows:

$$i \textbf{ ioco } s \iff \forall \sigma \in Straces(s) : \ out(i \textbf{ after } \sigma) \subseteq out(s \textbf{ after } \sigma)$$

2.3 Symbolic Transition Systems

LTSs lack the required level of abstraction for modeling complex systems. These systems use data structures with large or infinite data domains. Using a LTS to model such systems, each data value will take up a transition of its own in the model which can also lead to a new state. This representation of the data can lead to what is known as the *state space explosion* problem [FTW04] where the number of states and transitions in the model grow enormously. Just to give an idea how a LTS would grow in size if it had to model data, consider as an example a coffee machine where we had to insert 40 cents before it would dispense coffee. Suppose it accepted coins of 10 and 20 then the LTS specifying the behavior of this coffee machine would be the model in figure 2.2. If the coffee machine would have also accepted five-cent coins then the LTS would have grown even bigger. This example shows that although LTSs are very useful but they cannot be used for modeling complex software systems. To overcome this problem Symbolic Transition Systems (STS) [FTW04] were introduced. STSs extend LTSs with an explicit notion of data and data-dependent control flow by using state variables, label parameters, constraints and update mappings.

Definition 6. A symbolic transition system is a tuple $\langle Q, q_0, L, SV, \iota, LP, T\rangle$:

- Q is a countable set of states and $q_0 \in Q$ is the start state.
- $L = L_i \cup L_u$ is the set of input and output labels.
- SV is a countable set of state variables. These variables are global with respect to the whole model and are used for storing data. They can be used in all the transitions throughout the STS. ι is an initialization of these variables.
- LP is a countable set of label parameters which is disjoint from SV. Contrary to state variables, label parameters are local to the transition in which they are used meaning outside that transition, they are not accessible.
- T is the transition relation. Observable transitions are labeled with one of the input or output labels in L depending on whether they are a stimulus or a response. Unobservable transitions do not have labels. Transitions can be constrained and/or have an update mapping. The constraints are logical formulas over state variables and/or label parameters. A transition can be taken if the constraint evaluates to true. An update mapping is a set of assignments. The left hand side of an assignment is always a state variable and the right hand side is an expression that updates the value of the state variable. The update mapping is executed whenever the transition is taken. We write $q \xrightarrow{\lambda,\varphi,\rho} q'$ instead of $(q, \lambda, \varphi, \rho, q') \in T$, where λ is the label of the transition, φ and ρ are its constraint and update mapping respectively.

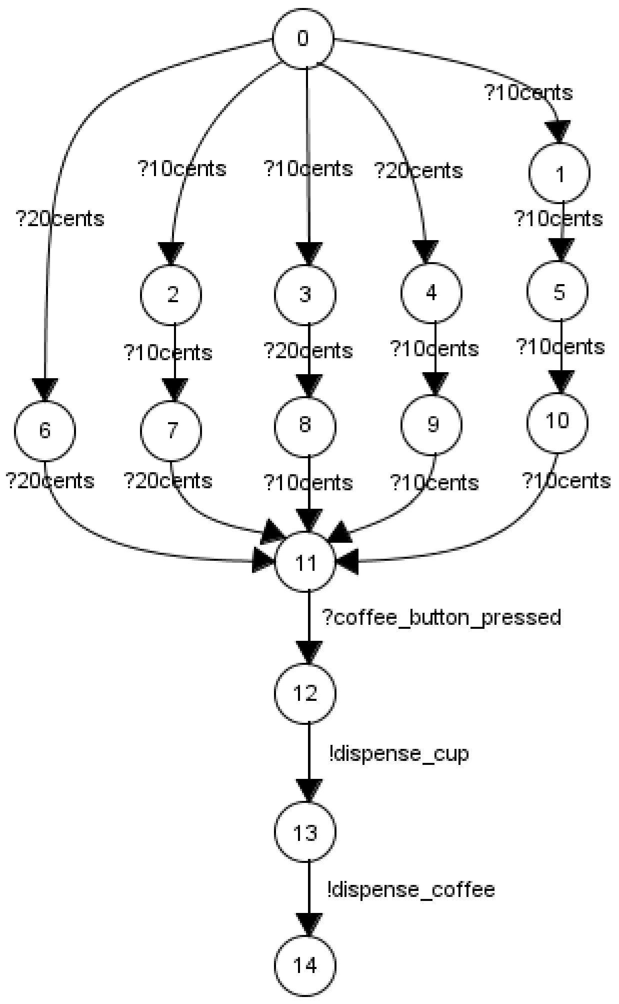

Figure 2.2: State space explosion

The STS modeling the behavior of our coffee machine example is given in figure 2.3. Compared to the LTS variant, this model is a lot more compact and it will not experience the *state space explosion* problem if we further extend the coins that can be used in the machine. This is because STSs treat the data on the transitions symbolically. Its no longer necessary to have a separate transition for each data value in the model. For example, label parameter *_coin* in the first transition represents any coin that is inserted into the machine. *_coin* can have any value greater than 0. Therefore the transition labeled with ?input_coin should not be considered as a single transition as it is actually representing infinitely many. After taking this transition, state variable *money* will be updated with the value of *_coin*. By giving input ?coffee_button_pressed we will arrive at state 2 in the model. Our next transition in the model depends on the value of *money*. If this variable has a value greater than or equal to 40 then we can observe outputs !dispense_cup and !dispense_coffee consecutively from the SUT. Otherwise we have to take the transition back to state 0 to insert more money. The loop-back transition from state 2 to 0 is unobservable.

Symbolic treatment of data, leads to a symbolic-ioco referred to as ***sioco*** [FTW06]. *sioco* does not change the original *ioco*-testing theory; it only lifts *ioco* to symbolic transition systems and gives another representation of the relation where data variables and parameters are involved. For a complete symbolic framework for model based testing see [FTW06].

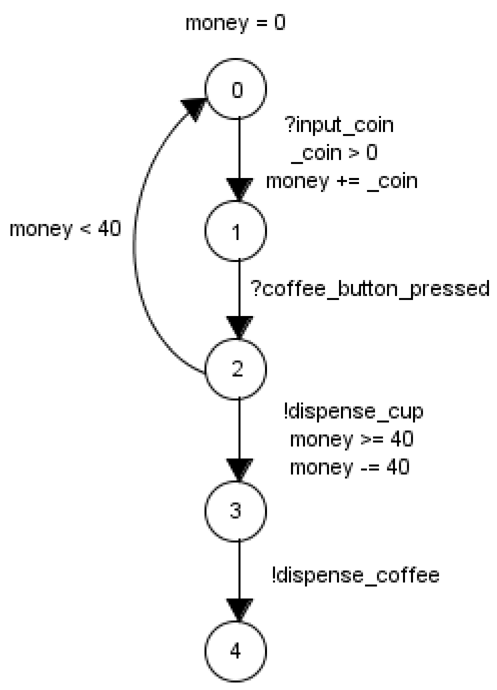

Figure 2.3: Symbolic treatment of data

Chapter 3

Requirements Considerations

There are some potential problems on the path to achieving the goals of this research. Some of these problems were apparent right from the beginning and the rest were identified during our research. We have divided these problems into two categories based on the time they may arise, either during test generation or test execution. Each problem is first explained using an example model and then a requirement is introduced that must be fulfilled by our test strategy. Solutions are given in chapter 5.

3.1 Test Generation

As we mentioned earlier, The set of suspension traces of the specification is usually infinite. There are two reasons that can make it infinite: one is presence of loops in the specification and the other is that there could be label parameters on the transitions that can be instantiated from an infinite domain of values.

3.1.1 Loops in the Specification

The transition in the model below does not have data on it but the number of traces for this model is already infinite and that is because of the loop in it. Our strategy should handle the loops in the model by only generating bounded length test cases.

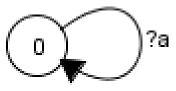

Figure 3.1: Loop

Test cases for this model: $\{?\,a,\ ?\,a?\,a,\ ?\,a?\,a?\,a,\ ...\}$

3.1.2 Instantiation with Unbounded Data Values

The most simple model that shows the problem with unbounded data is the model below. There are an infinite number of traces of length one in this model because label parameter $\_y$ can take its value from 1 to infinity and satisfy the constraint. Note that our models are symbolic, therefore the transition in the model below should not be considered as a single transition but it is representing infinitely many transitions since a transition with $\_y$ instantiated to 1 is different than a transition with $\_y$ instantiated to 2. Our strategy should handle the infiniteness of test cases resulted by unbounded data values on the transitions.

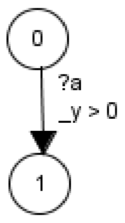

Figure 3.2: Unbounded data values

Test cases for this model:$\{?\,a(\_y = 1),\ ?\,a(\_y = 2),\ ?\,a(\_y = 3),\ ...\}$

3.1.3 Unobservable Transitions

Unobservable transitions were introduced in the formalism of LTS's (section 2.1). These kinds of transitions are widely used in the models. They are used for internal communication and are unobservable for the system's environment. Therefore during test execution, our strategy cannot interact with the SUT through these transitions, i.e., they cannot be chosen by our test strategy. This means test cases that are generated from the model by our strategy should not contain unobservable transitions. But these transitions can have constraints and/or update mappings on them so they should be taken into account during test generation because otherwise we will not be able to generate a test case from a symbolic trace or perform a reachability analysis on it. The model in figure 3.3 contains unobservable transitions. Transitions that are not decorated with either '?' or '!' are unobservable. In general unobservable transitions are not labeled but in some of the models through out this report, we have labeled them with letter T so that we can refer to them more easily.

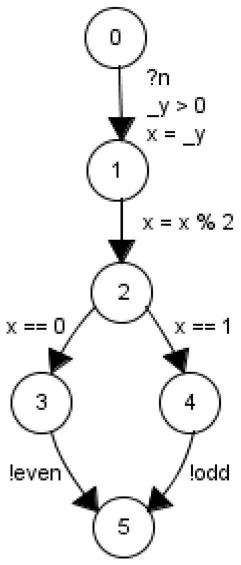

Figure 3.3: Model with unobservable transitions

Our test strategy should take unobservable transitions into account while generating test cases from the models but in the end the generated test cases should only contain observable transitions. For example, test cases generated from the model above should look like this:

Test cases for this model: $\{?\,n(\_y = 1)\,!\,odd,\ ?\,n(\_y = 2)\,!\,even,\ ?\,n(\_y = 3)\,!\,odd,\ ...\}$

We want our test cases to be of the same length therefore when our strategy is asked to generate test cases of length k, it should extract all the symbolic traces in the model that have k observable transitions and m unobservable ones and generate test cases from them. We do not have unobservable loops in our models so m is always finite. The extracted symbolic traces contain both observable and unobservable transitions and they might have different lengths, but the test cases that are generated from them will all be of the same length containing k observable transitions.

3.1.4 Computation of Reachability

Considering that the set of suspension traces of the specification is infinite, TestManager uses strategies that on the fly limits the set. A strategy determines the next step in the test by choosing an enabled transition from a set of possible transitions in the current state of the model. A transition is enabled if its constraint can be evaluated to true. If there are label parameters in the constraint, then the ability to take the transition depends on whether a valuation can be found for the label parameters such that the constraint is satisfied. TestManager uses GNU Prolog [Dia] to evaluate a constraint. GNU Prolog is a prolog compiler with constraint solving over finite domains. Given a constraint, GNU Prolog finds the possible valuations (if there is any at all) that satisfy the constraint. If the constraint has a solution, then a valuation is chosen from the solution domain based on a configurable valuation method (min, max, middle or random). The transition can then be instantiated using the chosen valuation.

Each strategy in TestManager decides among enabled transitions based on the purpose it was designed for. For example, there is a strategy that chooses its next transition randomly and there are more sophisticated strategies that aim at obtaining a high state/transition coverage. A common problem among these strategies is that sometimes they have a hard time covering a transition with a constraint that is dependent on constraints of other transitions which are located further up in the trace. The fifth transition in the STS in figure 3.4 is one example. This transition can be taken if the value for state variable x is between 50 and 60. Value of x is dependent on all the previous transitions in the trace. The values chosen for the label parameters $\_y$ in the constraints of the first and second transitions have direct impact on our ability to take the fifth transition. We will walk through this trace from the start of the model to see how we will get stuck in state 4. Transitions labeled with T_1, T_2 and T_3 are unobservable.

Assuming a *min* valuation method, the constraint solver will choose 51 for label parameter $\_y$ in transition ?a, which results in x updating to 52. In the second transition, $\_y$ should have a value less than 52. Considering that we are using the *min* valuation method, the constraint solver will choose 0 for $\_y$, which results in x updating to -3. The third and forth transitions only have an update mapping which will change the value of x to -5 and then eventually to -4. With the current value of x, we will not be able to cover the fifth transition and thus reach the end of our trace. The inappropriate values chosen for the label parameters in the first two transitions by the constraint solver has resulted in a deadlock in state 4. Deadlock happens when the strategy is not able to continue because there are no enabled transitions to choose from. If we have used any of the other valuation methods, it would not have made any difference and the same deadlock would have occurred in state 4.

The problem depicted above which we refer to as the *reachability* problem is caused by the way the strategies in TestManager use the constraint solver. The constraint solver is used locally, meaning that at each step in the test, only the constraint of the next transition is taken into account and the values for label parameters are chosen such that only that constraint becomes satisfied. There is no mechanism in place that enables the constraint solver look ahead to see if there are other constraints further down the trace whose evaluation might be influenced by the values chosen for the current step. Our strategy is required to generate test cases for those symbolic traces in the model that have the reachability problem. A symbolic trace is a trace that has not been instantiated with data yet. There is only one symbolic trace in the model below that ends in a leaf.

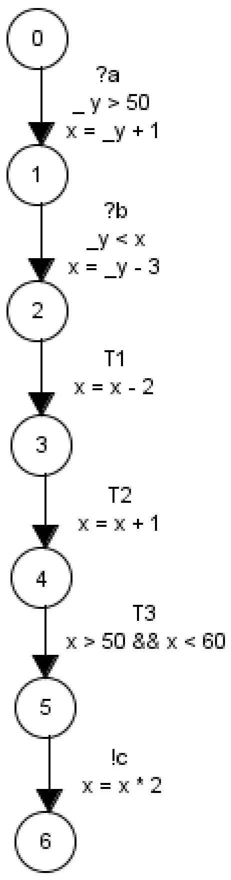

Figure 3.4: Computation of Reachability

Our test strategy should implement a reachability algorithm. Given a symbolic trace as input, this algorithm will perform a reachability analysis to decide whether the symbolic trace is ever reachable. A symbolic trace is reachable if there exists at least one valuation that can be used to instantiate the label parameters on the trace so that all the constraints along the trace can be satisfied simultaneously. For example the symbolic trace in figure 3.4 is considered reachable because if we instantiate $\_y$ in the first two transitions both with value 55, then we will be able to reach the end of the trace without encountering any deadlock.

Reachability analysis should not only focus on traces that are reachable, but it should also be able to detect traces with unsatisfiable constraints on them. These traces can never happen or in other words are impossible because we are never able to reach the end of them. No matter what valuation we use for them, we will always end up with a deadlock. In general, the reason for not being able to reach a certain state in the model could be either because there is a modeling error or it could be that the trace is not possible because of its strict length. Below we have provided two example models, one for each case. Trace !a ?b in figure 3.5 can never happen no matter what value is chosen for $\_y$. Looking at the constraints on this trace, it is clear that the modeler has made a mistake when creating this model. The model in figure 3.6 has an infinite number of traces. Trace ?input !output is impossible because taking the loop only once is not enough for satisfying the constraint on the !output transition. Here the strict number of steps in the trace has resulted in a deadlock.

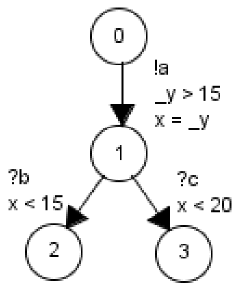

Figure 3.5: Modeling error
impossible trace: !a ?b

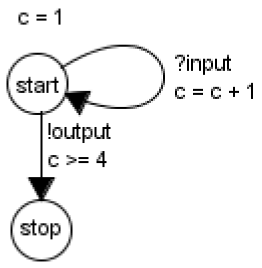

Figure 3.6: Strict trace length
impossible trace: ?input !output

The reason for including these models in our problem analysis is that models in general can become very complex and sometimes there is no way to satisfy a constraint on a certain path. If we do not take these situations into account, then our test strategy might end up with constructing traces that are not possible, while the result could be that our reachability analysis rules out these traces because it was able to detect that the path condition is always false.

3.2 Test Execution

A test case is a sequence of observable transitions (stimulus or response) each instantiated with data values. Executing a test case consists of supplying its stimuli to the SUT and observing the responses. As previously mentioned, test cases in the *ioco*-testing theory are constructed from traces of the specification. By executing each test case on the SUT we can check if the test case covered its corresponding trace in the specification, that is if the test case gets a pass verdict. A pass verdict is obtained when the observed responses correspond to the expected responses.

During test execution it is not always possible to remain on the test case that we set out to do from the start. It is likely to get off the current trace by receiving a response from the SUT which was not expected. Our test strategy should not immediately interpret this deviation as an error in the SUT. This deviation might have been caused by other reasons which we are going to explain in the remainder of this section.

3.2.1 Multiple Responses

A model could contain a state that allows multiple alternative responses. Having such states in the model could result in a deviation during test execution. Although our test strategy is in full control of which stimulus to give to the SUT, but it is entirely up to the SUT to decide which response to send back. Therefore, when there are multiple possible responses, from the point of view of the model, the SUT behaves in a non-deterministic way. Note that the SUT will almost never be truly non-deterministic. Which of the alternative responses is sent is determined in a deterministic way within the SUT, its just that we have not modeled the responsible inputs and decision process [AML]. An example model is given in figure 3.7. Suppose we want to cover the trace $?\,a\,!\,b$. By sending input $?\,a$ to the SUT, we will arrive at state 1 in the model. In this state the SUT can respond with either $!\,b$ or $!\,c$ and both are considered correct responses. If we receive $!\,b$ then we have successfully covered our intended trace. If on the other hand we receive $!\,c$ then we have gotten off the trace we set out to cover from the beginning. In a situation like this, our strategy is required to interpret this as a deviation from the current trace rather than an error. The deviation should not cause test execution to stop so our strategy is required to continue testing with another trace (in this example with the right hand side trace). Our strategy should not mark the left hand side trace as covered and it should come back to it again at a later time and test it again. In the end our goal is to cover every trace in the finite test set.

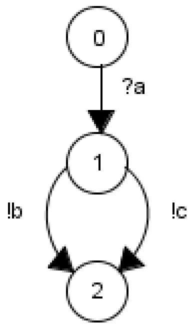

Figure 3.7: State 1 allows multiple responses

If the outgoing response transitions from a state have constraints on them, then we could still observe non-deterministic behavior from the SUT if more than one of these transitions are enabled at the same time, that is if the solution to their constraints overlap with each other.

3.2.2 Unexpected Valuation

Suppose there is a response transition !t with the following constraint on its label parameter: $1 \leq \_y \leq 5$ and suppose our test strategy has chosen to instantiate this transition with $\_y = 1$. During test execution we do not have any control over the values we get back from the SUT so when we receive !t, label parameter $\_y$ could be set to any number between 1 and 5. if the value is anything other than the one we already have on our test case then the transition after which the SUT has followed is different than the one our strategy has followed. This means we are not on our current test case anymore. The same requirements as in the case with *multiple responses* hold here as well. Upon receiving an unexpected valuation, our strategy should readjust by taking the new valuation into account and continue with test execution. There are five possibilities ahead of our testing strategy with the new valuation:

1. Continue on the same trace with the remainder of our current test case. Here the change in valuation hasn't had any impact on the remainder of our current test case and we are still able to reach the end of our trace without changing anything.
2. Continue on the same trace by generating a new test case for its remainder. With the new valuation some of the label parameters that are located further down the trace have to be recalculated therefore we are not able to reach the end of our trace using our current test case anymore.
3. Continue with another trace using the remainder of one of the test cases in our test suite.
4. Continue with another trace by generating a new test case for its remainder.
5. It is not possible to continue with any trace using the new valuation. This means the model has deadlock and it should be reported to the tester.

We are going to give an example for each of the above cases using the model in figure 3.8. This model has three symbolic traces. The trace on the left can be taken if state variable w satisfies the following constraint: $1 \leq w \leq 20 \ \wedge \ 3 \leq w \mod 6 \leq 5$. The trace in the middle can be taken if w satisfies this constraint: $20 < w \leq 40 \ \wedge \ 1 \leq w \mod 7 \leq 4$ and for the trace on the right w has to satisfy: $40 < w \leq 60 \ \wedge \ 0 \leq w \mod 4 \leq 1$. w is updated with label parameter $\_y$ in transition !a. The value for $\_y$ is determined by the SUT and it can be any number between 1 and 60. Suppose our strategy has created the following test cases from this model:

$$\{ \ \,!\,a(\_y = 3)\,?\,b(\_y = 3)\,!\,e, \quad !\,a(\_y = 22)\,?\,c(\_y = 1)\,!\,f, \quad !\,a(\_y = 41)\,?\,d(\_y = 1)\,!\,g \ \}$$

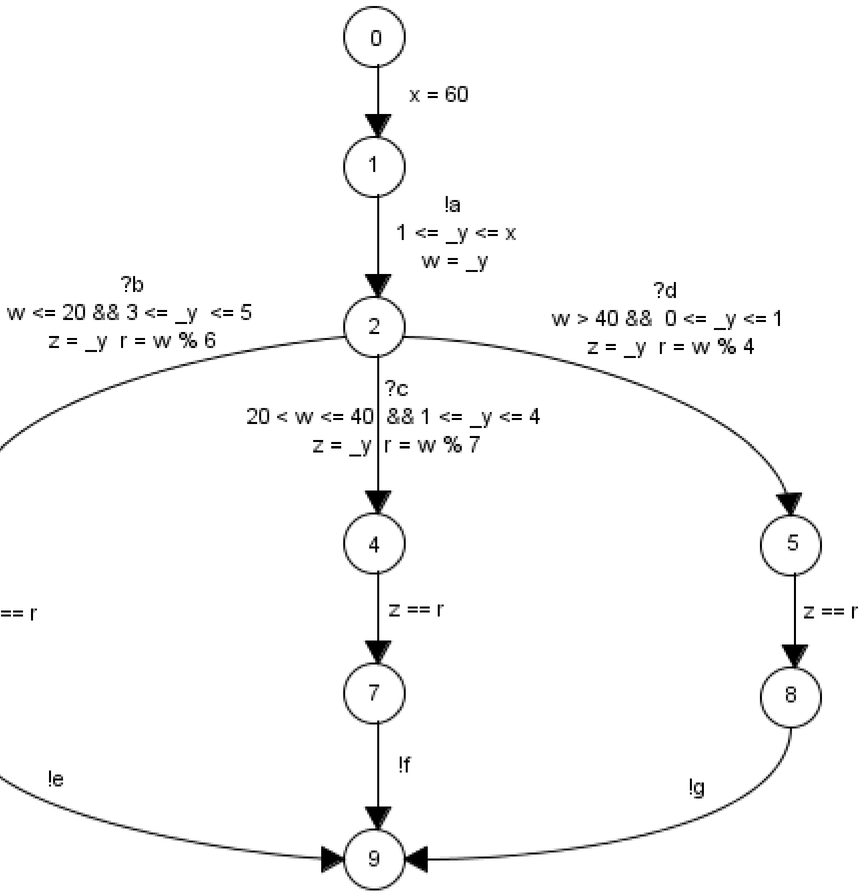

Figure 3.8: Unexpected valuation

We pick the first test case and we start executing it on the SUT in an attempt to cover the left trace in the model. When we get to state 1, the SUT responds with:

1. $!\,a\,(\_y = 9)$. With this value we can still use the remainder of our current test case to cover the left trace. label parameter $\_y$ in transition ?b does not need to change because 9 modulo 6 is 3.

2. $!\,a\,(\_y = 11)$ With this value we are still able to cover the left trace but we need to generate a new test case in which label parameter $\_y$ in transition ?b is instantiated to 5. 11 modulo 6 is 5.

3. $!\,a\,(\_y = 22)$ With this value we are no longer able to continue with the left trace but we can switch to the trace in the middle and we can use the remainder of the second test case to cover it.

4. $!\,a\,(\_y = 31)$ By receiving this value we have to switch to the trace in the middle and we also have to instantiate the remainder of our new symbolic path such that label parameter $\_y$ in transition ?c is set to 3. This is because 31 modulo 7 is 3.

5. $!\,a\,(\_y = 18)$ or $!\,a\,(\_y = 27)$ or $!\,a\,(\_y = 54)$ or ... Receiving any of these values will result in a deadlock. Our test strategy should detect deadlocks and report them to the tester.

6. Receiving any response other than !a is an error. Our strategy should detect this and report it to the tester.

What we have described above is just an example to illustrate every possible scenario that can happen after receiving an unexpected value from the SUT and the corresponding action that could be taken by our testing strategy. This is just an analysis of the problem and should not be interpreted as the final solution. Solutions are given in chapter 5.

The deviations we described so far as a result of *multiple responses* and receiving an *unexpected valuation* could happen together at the same time. We can see their combination in state 1 in the model below. In this state the SUT can non-deterministically respond with either !b or !c and at the same time it can send back a value which is not on our test case. This model also shows that two deviations can happen in a row. The second one could take place in state 6 where the SUT responds with either !g or !h. We will demonstrate these consecutive deviations with an example. This model has three symbolic traces. Suppose our strategy has generated the following test cases from this model:

$$\{ \ ?\,a\,!\,b(\_y = 1)\,!\,d\,!\,f, \quad ?\,a\,!\,c(\_y = 22)\,?\,e(\_y = 1)\,!\,g, \quad ?\,a\,!\,c(\_y = 22)\,?\,e(\_y = 1)\,!\,h \ \}$$

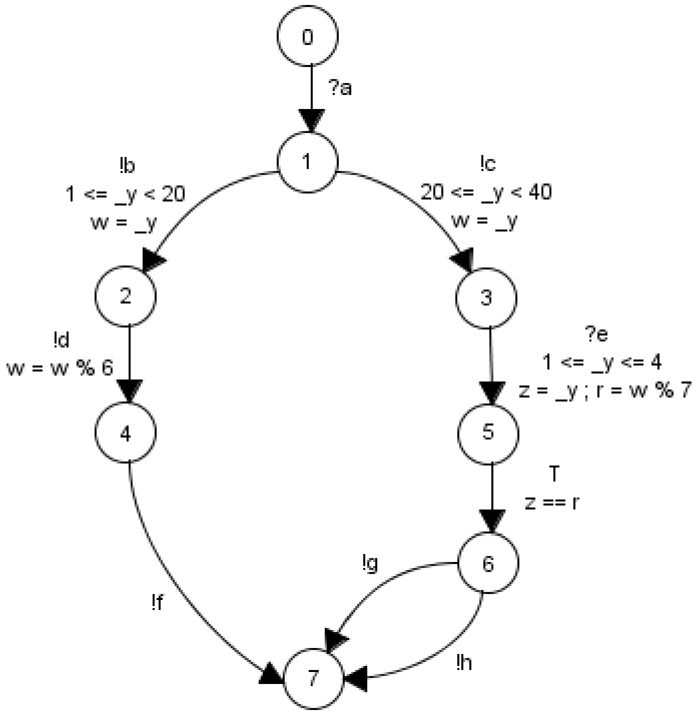

Figure 3.9: Receiving an unexpected response and valuation simultaneously

Transition labeled with T is unobservable. We start with the first test case in order to cover the left most trace in the model. But after sending input ?a, the SUT unexpectedly responds with $!\,c(\_y = 23)$. With this response we are not only thrown off our intended trace but we are also unable to proceed with any of the test cases in our test suite. At this point our strategy should find all the symbolic traces that it can continue the test with. In this example these traces would be $?\,a\,!\,c\,?\,e\,T\,!\,g$ and $?\,a\,!\,c\,?\,e\,T\,!\,h$. Our strategy should then check whether these traces are still reachable with the new valuation. In this example both traces are still reachable. In order to continue with either of these traces, label parameter $\_y$ in transition ?e should be recalculated to 2 $(23 \mod 7 = 2)$. Suppose our strategy chooses to continue with $?\,a\,!\,c\,?\,e\,T\,!\,g$. Test execution shall continue with no interruption until we arrive at state 6. Here we hope to receive output !g which is on our test case but let's assume that the SUT responds with !h. This is where the second deviation has occurred. Our strategy should take the same steps as before to calculate a new path through the model and continue with test execution. In general, our strategy is required to handle as many deviations as possible during test execution.

3.2.3 Non-determinism

Up until now we have been analyzing models where the problems with *multiple responses*, *unexpected valuation* and their combination together were happening in only one state. Although we have been observing non-deterministic behavior from the SUT but since we were only in one state, those cases should still be considered as deterministic. ***Non-determinism*** is when we are in one of several states, but we do not know which one. Presence of the following constructs in the model will result in non-determinism:

• When a state has more than one outgoing transitions and at least one of them is unobservable. State 2 in the model below is an example:

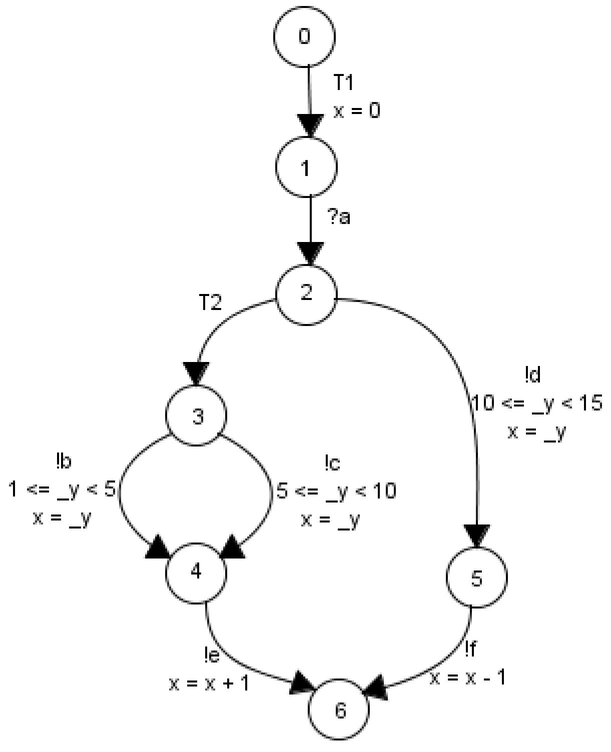

Figure 3.10: Multiple responses available from more than one state

The two unobservable transitions in the model above are labeled with T_1 and T_2. By sending input ?a to the SUT, we will end up in both states 2 and 3 at the same time. This is because unobservable transitions are automatically advanced in TestManager, so when we are in state 2 we are simultaneously in state 3 as well. It is important to note once again that the SUT is deterministic and is always in one state at any given moment. The available responses that we can expect to receive from the SUT are !b, !c and !d. Receiving any of these responses resolves non-determinism because then we can find out our position in the model. For example, if we receive output !b then we can know we were in state 3. While testing the traces in this model, it is very probable that we receive an *unexpected response* and *valuation* together at the same time when we are non-deterministically in two states. Our strategy is required to find the correct trace to continue testing with. This model has three symbolic traces. Suppose our strategy has generated the following test cases from this model:

$$\{\ \ ?\,a\,!\,b(\_y = 1)\,!\,e, \quad ?\,a\,!\,c(\_y = 5)\,!\,e, \quad ?\,a\,!\,d(\_y = 10)\,!\,f\ \ \}$$

In an attempt to cover the right most trace in the model we start by sending input ?a to the SUT. Hoping to receive output !d, the SUT non-deterministically responds with $c(\_y = 7)$. The correct symbolic trace to continue with in this example would be $T_1\,?\,a\,T_2\,!\,c\,!\,e$. Our strategy should be able to find this trace in the model and continue with it after analyzing whether the remainder of it is still reachable with the new valuation. Our strategy should also know the correct step in the trace, i.e., it should know that the next step would be to receive output !e from the SUT.

• Two or more outgoing transitions from the same state, have the same label and their constraints are not mutually exclusive, i.e., their solutions overlap. These transitions should be of the same kind, either stimuli or responses. Two example models are given in figures 3.11 and 3.12 for each case.

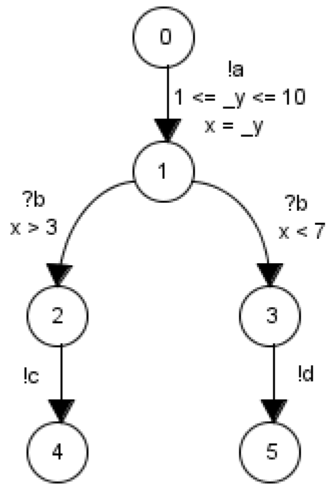

Figure 3.11: Multiple responses available from more than one state

State 1 in the model above has two outgoing input transitions with the same label and overlapping constraints. If they are enabled at the same time then we will end up in both states 2 and 3. This will lead to a non-deterministic behavior from the SUT because then we can receive either output !c or !d which in turn could result in a deviation from our current test case. Suppose we want to test the left trace and state variable x in transition !a is updated with $\_y = 5$. With this value, both transitions in state 1 are enabled so when we send input ?b to the SUT, both of them will be followed in the model. We will be able to cover our intended trace if we receive !c but since we are simultaneously in both states 2 and 3, it could be that the SUT responds with ?d. In this situation our strategy is required to find the correct trace after deviation has happened and continue on with it.

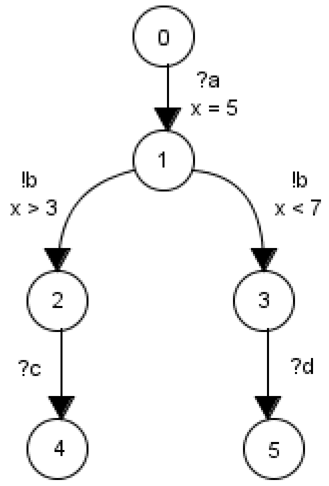

Figure 3.12: No deviation in this model

The model above is another example of non-determinism. With state variable $x = 5$, both response transitions from state 1 are enabled therefore when the SUT responds with !b, both of them will be followed, leading us to states 2 and 3 in the model. Contrary to the previous example, here non-determinism does not have any effect on our test execution and we will be able to cover whichever trace we intended to cover without any deviation happening along the way. This is because the outgoing transitions in states 2 and 3 are stimuli and as we have mentioned before, our test strategy is in full control of the inputs that are given to the SUT. It is our strategy that decides whether to give input ?c or ?d. Both of these inputs are valid at this point in the model and if our strategy chooses to send one of them to the SUT because it was on its current test case then that input will indeed be sent without any problem.

• Another form of non-determinism which is very common happens when there are transitions with the same label going out of more than one state in the model. Two example models are given in figures 3.13 and 3.14 for this form of non-determinism.

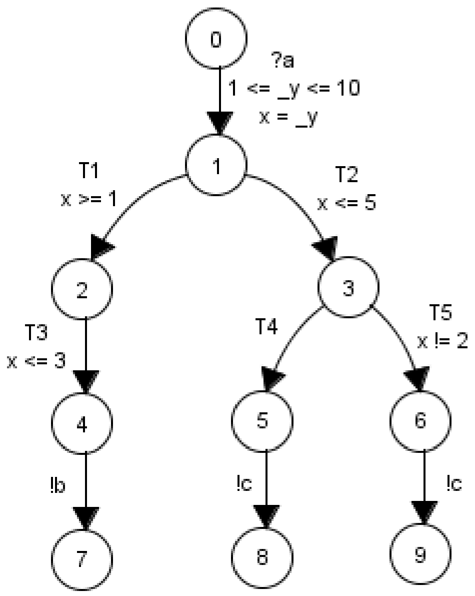

Figure 3.13: Multiple responses available from more than one state

Transitions labeled with T_1, T_2, T_3, T_4 and T_5 in the model above are all unobservable. These transitions are constrained and whether they will be followed or not depends on the value of state variable x. This model has three symbolic traces. With an intention of covering the trace $?\,a\,T_1\,T_3\,!\,b$ we start by sending input ?a to the SUT. Suppose x is updated with 3. With this value all the unobservable transitions will be followed and we will simultaneously be in all states 1,2,3,4,5 and 6. The available responses from these states are !b and !c. Receiving output !b from the SUT will resolve non-determinism because then we can know we were in state 4 in the model and we will also successfully cover our intended trace. On the other hand it is very probable that the SUT non-deterministically responds with !c. If this is the case then both transitions labeled with !c in states 5 and 6 are followed leading us to states 8 and 9 at the same time. Our strategy should handle the deviation by choosing a trace to continue testing with. At this point, we could be on either of the following traces: $?\,a\,T_2\,T_4\,!\,c$ or $?\,a\,T_2\,T_5\,!\,c$. So it would be correct for our strategy to continue with either one of them.

Let's go over this example once again, this time assuming state variable x is updated with 2 in transition ?a. Considering the constraint on T_5, this transition will not be followed, therefore after sending input ?a to the SUT we will simultaneously be in states 1,2,3,4 and 5. Although we are not in state 6 anymore but output !c can still be observed on the account of being in state 5. Consequently, the responses we can expect to receive are !b and !c, just like the previous example. The difference is that when we receive output !c, only one of the transitions labeled with !c are followed in the model and that is the one from state 5 to 6. Transition !c from state 6 to 9 could not have been followed because the unobservable transition T_5 was never enabled. Under these circumstances, there is only one trace that we can be on right now and that is $?\,a\,T_2\,T_4\,!\,c$. Our strategy is required to realize this and always pick the correct trace in case of deviation.

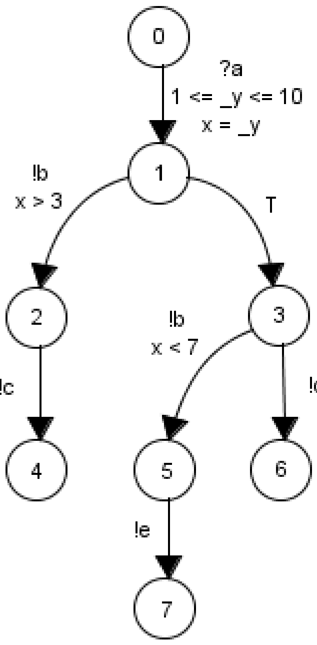

Figure 3.14: Multiple responses available from more than one state

The model given in the figure above is another example for non-deterministically being in more than one state and having more than one outgoing transition with the same label. It is similar to the previous model. The difference is that this time the observable transitions are constrained rather than the unobservable ones.

After sending input ?a to the SUT, we are simultaneously in both states 1 and 3. This is because of the unobservable transition T. The available responses from these states are !b and !d. Suppose state variable x is 5. With this value, both response transitions !b are enabled so if the SUT responds with !b then both of them will be followed leading us to states 2 and 5 in the model. At this point, our strategy can choose to be on either of the following traces $?\,a\,!\,b\,!\,c$ or $?\,a\,T\,!\,b\,!\,e$. Both of them are correct traces until we receive the next response from the SUT. In a way we can say that we are simultaneously on both of these traces. An interesting value for x in the model above is when it is equal to 1. With this value, when we are simultaneously in states 1 and 3, of the two transitions labeled with !b, only the one starting from state 3 is enabled and only this transition will be followed upon receiving !b from the SUT. In case there is a deviation and the SUT has non-deterministically responded with !b, our strategy should switch to the correct trace to continue with. The correct trace in this case would be $?\,a\,T\,!\,b\,!\,e$.

3.2.4 Over-specification

In general our test strategy is required to cover all the traces in the finite subset of Straces(s). A trace extracted from the specification can be marked as covered if the SUT produces the exact sequence of responses which are on the trace upon receiving the stimuli from our strategy. We explained that during test execution it is not always possible to remain on the trace that we set out to cover from the beginning. Using the example models in the previous sections, we illustrated all the possible reasons for it. Deviating from the current trace is something that can happen frequently during test execution. On one hand our strategy is responsible for covering all the traces in the test suite, on the other hand there could be traces that cannot be covered within any reasonable number of attempts because every time we try to cover them, a deviation happens and we end up covering another trace. Specifications could also contain traces that can never be covered. These kind of traces are never covered not because of a deviation during testing but rather because they are not present in the SUT so they are never observed by our strategy. A requirement for our testing strategy is that it should not get caught in an infinite loop trying to cover such traces. Below we have described two cases where a trace in the specification can never be covered.

• The inputs that are given to the SUT are controlled by our testing strategy but it is up to the SUT to decide which output to produce. We can force the system to perform a certain output by computing a path that includes our desired output but the system still has to be able to produce it. It is possible that there are traces in the specification that we will not see in the SUT and because of this our strategy might get caught in an infinite loop trying to cover a trace that can never happen. The models given for a SUT and its specification in the figure below is an example. The specification allows observing output !b or !c after receiving input ?a while the SUT only produces !b. The specification has over-specified the SUT. Our strategy will never be able to cover the trace $?\,a\,!\,c$ in the model of the specification because the SUT never responds with !c upon receiving input ?a. Not observing output !c does not mean we have a failed test case and our strategy should not interpret that as such.

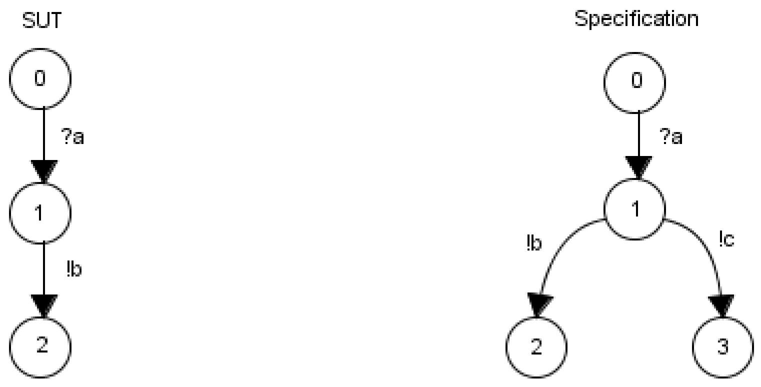

Figure 3.15: Impossible response due to over specification

• A testing strategy might try to cover a trace in the specification over and over again but it cannot because there is an error in the SUT. The models given for a SUT and its specification in figure 3.16 is an example for this case. According to the specification, the system should respond with !d after receiving input ?b from the environment but the model of the SUT shows that this is not the case and that the system responds with output !c. This is an error. A testing strategy with a naive implementation might get stuck in an infinite loop trying to cover the trace $?\,b\,!\,d$ not knowing that this trace will never be covered. In this case, the requirement on the strategy is that it should identify and report the error and also remove $?\,b\,!\,d$ from the test suite. In fact all the traces for which $?\,b\,!\,d$ is a prefix should be removed because they will all lead to the same problem.

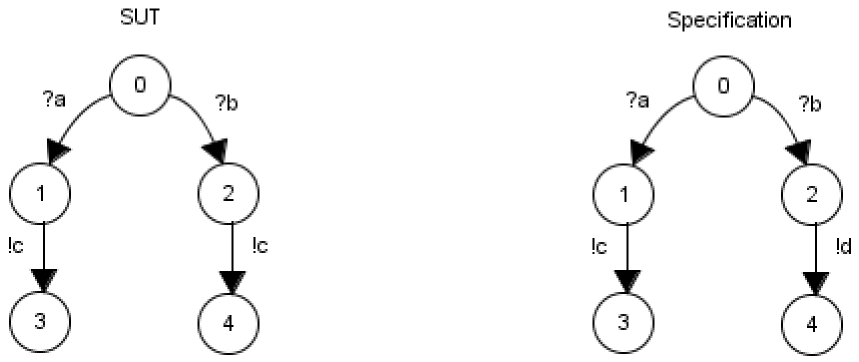

Figure 3.16: Impossible response due to an error

Before closing off this section there is one last requirement that should be fulfilled by our testing strategy and that is no expensive calculation taking place during test execution. This is especially true when there is a deviation from the current test case. At this point our strategy should find the correct trace to continue the test with. Depending on the current position in the model, a significant number of traces might become candidates for the new trace. Our strategy should find these traces and then filter those that are not reachable with the current valuation of the state variables. Computation of reachability involves some calculations that are expensive and might take some time. Our test strategy should prevent communication delays with the SUT because the server on which the SUT is running might terminate its connection with TestManager. In this case our strategy has to restart the test from the top.

Chapter 4

Research Method

Our in-depth analysis of the problem from the previous chapter shows that there are a considerable number of variables involved in the process of test case generation and execution. Each of these variables contribute to the complexity of the problem in its own way, thus we need to understand and tackle each of them separately. We have chosen an experimental approach to achieve the goals of this research. The reason is that experiments are essentially reductionist - they reduce complexity by allowing only variables of interest to vary in a controlled manner, while disregarding all other variables [ESSD08]. The results gained from each experiment can then be generalized to real world settings.

We have used the models from the previous chapter for our experiments. Each model addresses a specific problem and abstracts from the remaining problematic factors. For example the model in figure 3.4 has only one trace with a hard constraint on it and it was used for solving the reachability problem. In each experiment, a finite test set is created from the model and then it is tested. Our goal is to cover every trace in the finite test suite. We learn new things from experimenting with each of these models and we gradually develop our strategy as we go along.

For validation of our strategy we have used an industrial model which belongs to one of Axini's clients called *FEI*. It is a very big model and a problematic one for testing. This model has loops in it. Parameters are instantiated from an infinite domain. It is filled with unobservable transitions which makes room for a lot of non-deterministic transitions and consequently deviations during test execution. The number of traces at each depth grows exponentially and they all suffer from the reachability problem. This model can be provided on request, but due to confidentiality reasons it cannot be included in the thesis. The results obtained from running our strategy on the *FEI* model are discussed in section 5.3.

Chapter 5

Research

5.1 Definition of the Finite Subset of Suspension Traces

Presence of loops in the specification and instantiation of label parameters with unbounded data values make the set of suspension traces of the specification infinite. Specifications that contain loops are like cyclic directed graphs in which the number of paths is infinite. If we unroll all the loops in the specification then we will have a tree-like structure where the number of paths becomes finite. At this point, the only thing that can still make the number of traces infinite is unbounded data values and because of that we have decided to abstract from it. This does not mean that we disregard data on the transitions entirely but it means that from all the possible data values that will satisfy the constraint on a transition and consequently making it traversable, we are going to choose only one of them. Our goal is not to get a good coverage of data values. That is data coverage which we are going to abstract from, but we still need to have a data value to instantiate the transition with. By instantiating each transition in a symbolic trace with a valuation, we will obtain a concrete trace or a test case. For example, consider the following symbolic trace in the model below:

$$?\,a\,[\_y \geq 1]\ \ !\,b$$

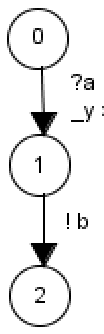

Figure 5.1: Abstracting from data coverage

Test cases for this model: $\{?\,a(\_y = 1)\,!\,b,\ ?\,a(\_y = 2)\,!\,b,\ ?\,a(\_y = 3)\,!\,b,\ ...\}$

We need a test case in order to cover the symbolic trace in the model. We cannot get a test case unless we instantiate label parameter $\_y$ with a value. On the other hand if we take data coverage into account then we will have an infinite number of test cases for the symbolic trace since $\_y$ can be instantiated from an infinite domain. By abstracting from data coverage and choosing only one instantiation for the symbolic trace, we are able to take the infiniteness of test cases under control. Therefore from all the test cases in the set above, we are only interested in having one of them tested against the SUT.

We can create a finite test set by bounding the model of the specification both in terms of loops and data. We limit the model to a certain depth and we unroll all the loops which will result in a symbolic tree. We generate one test case for every symbolic trace in the tree that leads to a leaf. The resulting set of test cases is our definition of the finite subset of suspension traces of the specification.

5.2 Trace Coverage Strategy

We have designed and implemented a *Trace Coverage Strategy* that checks the input-output conformance of a system under test with respect to the model of its specification. Given a symbolic transition system, *Trace Coverage Strategy* creates a finite test suite according to the definition we gave in the previous section and then executes it. Our strategy solves all the problems we described in chapter 3. In 5.2.1 we discuss loop unrolling and the extraction of symbolic traces from the model. In 5.2.2 we present a reachability algorithm and the process of test generation from symbolic traces. In 5.2.3 we provide a solution for finding our position in the model after a deviation. In 5.2.4 we describe the characteristics of traces that are non-deterministically covered together in a single test run and how we find and mark them during test execution.

5.2.1 Extracting Symbolic Traces

Extracting symbolic traces from the model is our starting point towards creating our definition of the finite test suite. Symbolic traces are extracted from a symbolic tree which is the result of unrolling the loops in the model. We do not use an algorithm to detect loops and then statically unroll them. Instead we limit the model to a certain depth and we use depth-first search to traverse the model and extract the traces from it. Loops are dynamically unrolled as we are doing the traversal.

The depth to which we limit the model corresponds to the length of the test cases that we want to generate which would be the number of observable transitions in each symbolic trace. This is because a test case is an instantiation of a symbolic trace and they contain only observable transitions. For example when generating test cases of length 5, our strategy will find all the symbolic traces in the model that have 5 observable transitions and x unobservable ones. The extracted symbolic traces may have different lengths but all the test cases that are generated from them will have the same length, all containing 5 observable transitions. We start from the root of the model and we continue as far as possible along each branch until we reach a state that is a leaf or we reach the maximum number of observable transitions in our trace. In either case, we extract the symbolic trace and then we backtrack to an earlier branch to find the next symbolic trace in the model. We do not check whether a state has been visited before and we do not use it as a criteria for backtracking. This is because we want to be able to take a loop more than once and have all the possible number of times a loop can be taken within the limit. Limiting the model on the length of its traces is very practical because not only the number of traces becomes finite but also we can find the shortest path to a bug in the SUT.

In figure 5.2 we have a model that contains a loop and in figure 5.3 we see the symbolic tree that has resulted from limiting the model to depth 4 and unrolling the loop in it.

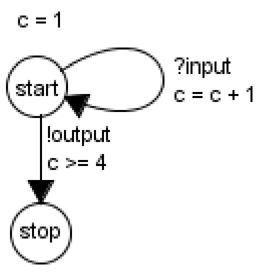

Figure 5.2: Model with a loop

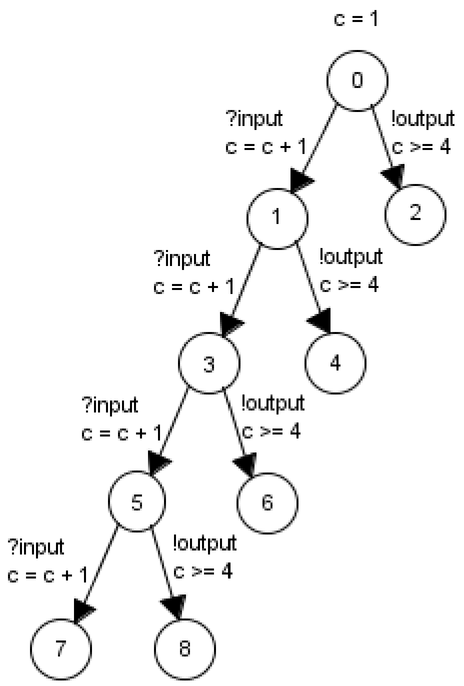

Figure 5.3: Symbolic tree

According to our definition of finite subset of suspension traces, the set of symbolic traces consists of all the paths in the symbolic tree that lead to a leaf. The set of symbolic traces for the model in figure 5.2 after having it bounded to depth four is calculated as follows:

$$\{!\,output(start->stop)\;\;[c>=4]\;\;[\,],$$

$$\begin{array}{r}?\,input(start->start)\;\;[\,]\;\;[c=c+1]\\ !\,output(start->stop)\;\;[c>=4]\;\;[\,],\end{array}$$

$$\begin{array}{r}?\,input(start->start)\;\;[\,]\;\;[c=c+1]\\ ?\,input(start->start)\;\;[\,]\;\;[c=c+1]\\ !\,output(start->stop)\;\;[c>=4]\;\;[\,],\end{array}$$

$$\begin{array}{r}?\,input(start->start)\;\;[\,]\;\;[c=c+1]\\ ?\,input(start->start)\;\;[\,]\;\;[c=c+1]\\ ?\,input(start->start)\;\;[\,]\;\;[c=c+1]\\ !\,output(start->stop)\;\;[c>=4]\;\;[\,],\end{array}$$

$$\begin{array}{r}?\,input(start->start)\;\;[\,]\;\;[c=c+1]\\ ?\,input(start->start)\;\;[\,]\;\;[c=c+1]\\ ?\,input(start->start)\;\;[\,]\;\;[c=c+1]\\ ?\,input(start->start)\;\;[\,]\;\;[c=c+1]\}\end{array}$$

There are five symbolic traces in the set. The first three result in a deadlock. Our reachability analysis will identify them and discard them from the set. In this example, our test suite will contain only two test cases, one for each of the remaining symbolic traces. In the next section, we will explain our reachability algorithm and we will show how the constraint solver becomes empowered to find valuations for all reachable symbolic traces once it joins forces with our reachability algorithm.

5.2.2 Generating Test Cases

We want to have one test case for every symbolic trace. The test case should be able to follow its symbolic trace through the model and cover all the transitions on the trace. For this purpose, the label parameters on the symbolic trace need to be instantiated in such a way that all the constraints along the trace can be satisfied simultaneously.

In this sense, generating a test case from a symbolic trace becomes very similar to a constraint satisfaction problem (CSP). A CSP is defined [Apt03] as *a tuple* $(\{x_1, ..., x_n\}, \{D_1, ..., D_n\}, \{C_1, ..., C_k\})$ *where* $x_1, ..., x_n$ *are variables,* $D_1, ..., D_n$ *are their respective domains and* $C_1, ..., C_k$ *are constraints over subsets of* $\{x_1, ..., x_n\}$*. An assignment* $d_1 \in D_1, ..., d_n \in D_n$ *of all variables to elements in their domain solves a CSP (is a solution to the CSP) iff it solves all constraints* $C_1, ..., C_k$ *in the CSP. We call the CSP consistent if it has a solution and inconsistent otherwise.*

We can think of the label parameters in a symbolic trace as the variables in a CSP. The constraints along the trace also represent the set of constraints in the CSP. If the analogous CSP given by a symbolic trace is consistent, then we know the trace is reachable and we can use the solution to instantiate the label parameters and generate a test case. Note that there could be more than one solution to the CSP but since we are not taking data coverage into account, we will adhere to a single solution. If on the other hand, the CSP happens to be inconsistent then we will know the trace is impossible and we will discard it from our set of symbolic traces.

We use a constraint solver (GNU Prolog) to solve the CSP that is given by a symbolic trace. If we use the constraint solver in the way we described in section 3.1.4 (locally, considering only one transition at any given moment), there is a high chance that we will not be able to reach the end of our symbolic trace and successfully generate a test case from it. On the other hand, we cannot just simply hand over all the label parameters and the constraints of a symbolic trace to the constraint solver and expect to receive a solution from it. This is because there are implicit dependencies between constraints and the only way to make them explicit is to take all the update functions along the symbolic trace into account as well.

If we want to generate a test case that has to cover all the transitions in a symbolic trace and there are dependencies between constraints, then we need to make the constraint solver work globally with respect to the whole trace. For this purpose, a proper translation of constraints and updates from the symbolic trace to the constraint solver needs to be established. We have implemented a reachability algorithm in Ruby that does this. Given a symbolic trace as input, it backtracks from the leaf to the root, collects all the constraints and updates along the trace and combines them into a new constraint. This new constraint represents the whole trace. By passing it to the constraint solver we can determine whether there is a solution for it. If there is a solution, then we will know the symbolic trace is reachable and we can use the solution to generate a test case that is able to cover the entire trace in one pass.

With this approach we are not taking one transition at a time any more but we are taking a sequence of transitions all at once. This concept has been formally defined as the *generalized transition relation* in [FTW06]. If l and l' are two states in the STS and we have the following generalized transition relation $l \xRightarrow{\sigma,\varphi,\rho} l'$, then state l' can be reached from state l via a series of transitions, the sequence of which is dictated by trace σ, and the values that are passed over these transitions satisfy the conditions collected in φ (called attainment constraint) and the values for state variables are specified by update-mapping in ρ [FTW06]. Attainment constraint corresponds to what is called a path condition in the literature for symbolic execution of programs. Part of our reachability algorithm is dedicated to building the attainment constraint of a symbolic trace. Note that we have used the phrase *trace-constraint* through out the report instead of attainment constraint.

The following pseudocode explains the steps that are involved in building a *trace-constraint* from a symbolic trace. We will go through the code and explain each step using an example model. The example we are going to use is the STS in figure 3.4.

Pseudocode 1 Building trace-constraint from a symbolic trace

```
function BUILD TRACE CONSTRAINT(trace)
    trace_constraint, $i$ = GET LAST CONSTRAINT AND INDEX OF PRECEDING TRANSITION(trace)
    trace_constraint = NORMALIZE LABEL PARAMETERS(trace_constraint, $i + 1$)
    for $i$.downto(0) do
        transition = trace[$i$]
        if transition has update mapping then
            trace_constraint = PROCESS UPDATE MAPPING(update_mapping, trace_constraint, $i$)
        end
        if transition has constraint then
            trace_constraint = PROCESS CONSTRAINT(transition_constraint, trace_constraint, $i$)
        end
    end
    return trace_constraint
end

function PROCESS UPDATE MAPPING(update_mapping, trace_constraint, $i$)
    for all assignment in update_mapping do
        NORMALIZE LABEL PARAMETERS(assignment.rhs, $i$)
        trace_constraint = REPLACE STATE VARIABLES(trace_constraint, assignment)
    end
    return trace_constraint
end

function PROCESS CONSTRAINT(transition_constraint, trace_constraint, $i$)
    NORMALIZE LABEL PARAMETERS(transition_constraint, $i$)
    trace_constraint.join(transition_constraint)
    return trace_constraint
end
```

The first step is to collect the last constraint on the trace. We backtrack from the leaf towards the root and we stop at the first transition that has a constraint on it. In our example it would be the fifth transition. We also store the index of the preceding transition in variable i (line 2 in pseudocode). i is now 3 and the *trace-constraint* looks like this:

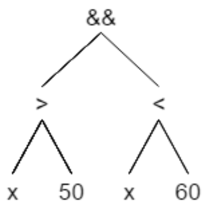

Figure 5.4: trace-constraint has a tree structure

Before proceeding to the next transition we need to normalize the label parameters in the *trace-constraint* (line 3). In order to do that, we have to traverse the *trace-constraint* and associate each label parameter with the transition to which it belongs. We do this by appending the index of the transition to the label parameter's name. This has been inspired from [Sie09]. Although label parameters are local to each transition, their names could be shared among more than one transition. Like in this example where the name of the label parameter in both transitions ?a and ?b is $\_y$. We need to have a way of distinguishing between these label parameters so when we have constructed a constraint for the whole trace and passed it to the constraint solver, we can know which value in the solution belongs to which transition. Having this distinction, we can map the label parameters back to their original names and instantiate them with the values we received from the constraint solver. The *trace-constraint* in figure 5.4 does not contain any label parameters so the normalization in line 3 results in the same *trace-constraint*.

From this point onwards, for each transition on our way to the root, we first process its update mapping (line 7) and then its constraint (line 9). Each update mapping could contain one or more assignments. The left hand side of an assignment is always a state variable and since we are sure it is never a label parameter, we just normalize the right hand side of the assignment (line 16). The update mapping in transition T_2 only has one assignment ($x = x + 1$). There are no label parameters on its right hand side so the normalization will not change anything about it. After having the right hand side of the assignment normalized, we traverse the *trace-constraint* to find the state variable that is on the left hand side of the assignment and replace it with the right hand side expression (line 17). In our example, x will be replaced by $x + 1$. Transition T_2 does not have any constraints on it so the *trace-constraint* will look like this:

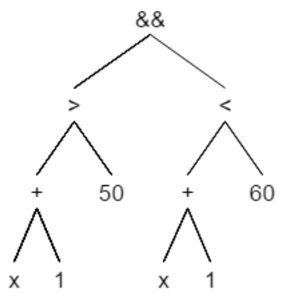

Figure 5.5

The exact same steps will be applied to transition T_1, after which the *trace-constraint* will grow further to become like this:

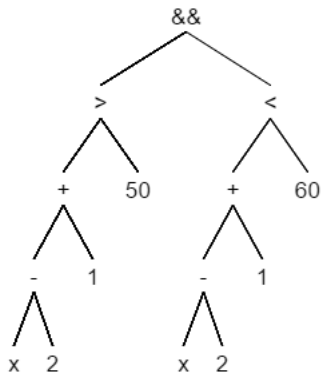

Figure 5.6

Now we are at transition ?b which has both an update mapping and a constraint. We first process the update mapping. It has only one assignment. We normalize the right hand side of the assignment and it becomes like this: $x = \_y_1 - 3$. The label parameter is now linked with transition ?b whose index in the trace is 1. After the normalization, we traverse the *trace-constraint* and replace every occurrence of state variable x with $\_y_1 - 3$. As a result, the *trace-constraint* will become like this:

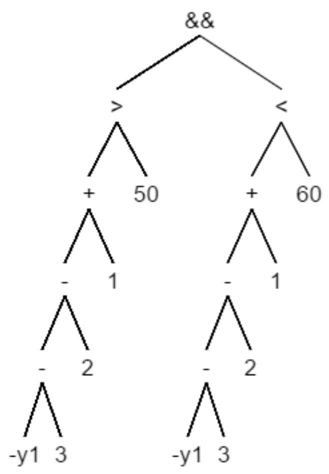

Figure 5.7

Now its time to process the constraint on transition ?b. We first normalize the label parameters in it with the index of the transition (line 23). The constraint becomes like this: $\_y_1 < x$. At this point we have to join the normalized constraint together with the current *trace-constraint* (line 24). This will result in a new *trace-constraint* which is the conjunction of the two aforementioned tree structures:

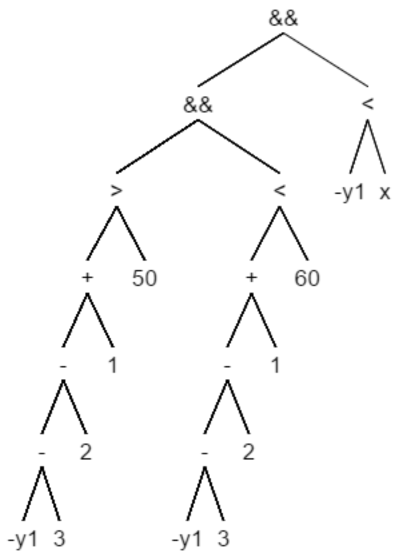

Figure 5.8

The steps for transition ?a are similar to what we had for transition ?b. After processing the update mapping, the *trace-constraint* becomes like this:

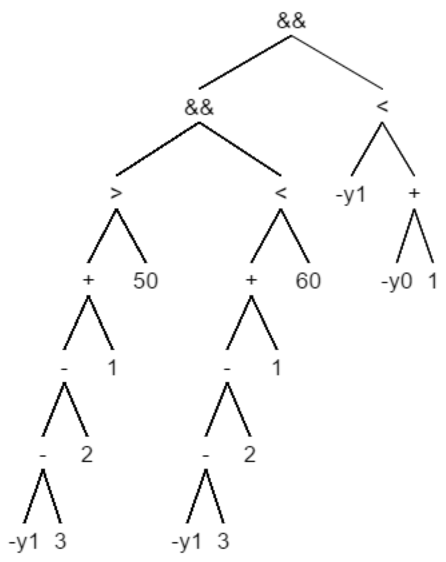

Figure 5.9

After processing the constraint of transition ?a, the *trace-constraint* becomes like this:

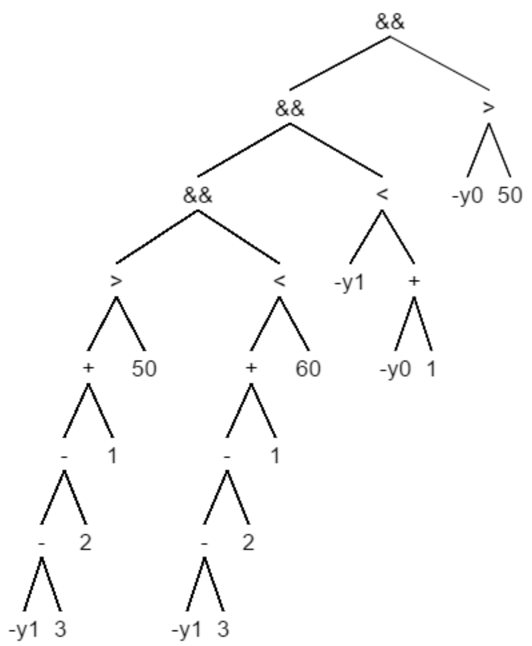

Figure 5.10

We have arrived at the root of the model and there are no more transitions to process. The above tree represents the final *trace-constraint* that overlooks the entire symbolic trace. We send it to the constraint solver. Considering that we are using the *min* valuation method the solver returns $\{"\_y_0" => 55\,, "\_y_1" => 55\}$. Receiving this solution from the constraint solver means that the symbolic trace is reachable. By instantiating both label parameters $\_y$ in the first two transitions with 55, we should be able to reach the end of our trace without encountering any deadlock. The final step is to remove the unobservable transitions from the instantiated trace which will leave us with the following test case: $?\,a\,(\_y = 55)\ \ ?\,b\,(\_y = 55)\ \ !\,c$.

For every reachable symbolic trace extracted from the bounded model we create a *Trace* object. *Trace* is a class that operates alongside our strategy. Apart from the functionalities that it provides to our strategy, each object of this class encapsulates the following information: a *symbolic trace*, a *test case*, the *trace-constraint*, a boolean variable *covered* that indicates whether the trace has been covered and an integer variable *trace attempt* that annotates the number of times our strategy has tried to cover this trace. *trace attempt* is used to avoid infinite looping over traces that cannot be covered within any reasonable number of attempts. If the number of attempts for covering a trace reaches the maximum number of trace attempts configured by our strategy then that trace will not be chosen by the strategy anymore.

5.2.3 Handling Deviation

We have reached the point where our strategy has created a test suite according to the definition of the finite subset of Straces(s). Test generation is over and now it is time to test the behavior of the SUT by executing the generated test cases against it. Our strategy picks a trace from the test suite that has not been covered yet and has the least amount of trace attempts. Each trace has a test case. By sending the stimuli to the SUT, our strategy checks whether the observed responses correspond to the expected responses. If the result of applying the test case to the SUT is successful then the corresponding trace along with all the other traces that non-deterministically have been covered as well (if any), will be marked as covered (see 5.2.4). Our strategy can then pick another trace and continue with test execution. If on the other hand a response is received which does not correspond to what we have on our test case, then our strategy should find out the reason behind this mismatch. Receiving an unexpected response could be either because there is an error in the implementation of the SUT or because of a deviation from the current trace in the model. In the former case, the unexpected response is indeed unexpected as it is not allowed by the specification. Our strategy should detect and report it. As for the latter case the received response is valid according to the specification, its just that it is not the same as our strategy's current step either because of a change in valuation, label or both. In this case our strategy should find the correct trace to switch to and continue the test with.

We have came up with a solution that does exactly what we just described. The solution has been implemented in Ruby as part of our *Trace Coverage Strategy* and it is able to handle all the possible scenarios of deviation that we described in section 3.2; both deterministic and non-deterministic variants. The pseudocode in page 36 describes the steps involved in this solution. First we will explain the solution and then we will review some of the example models from section 3.2.

Explanation of Solution

Whenever our strategy sends a stimulus to the SUT, or when the SUT returns a response, one or more transitions will be followed in the model. We say more than one transition because of non-determinism. As for unobservable transitions, they are automatically advanced in the model whenever they are enabled. During test execution our strategy should be aware of all the observable transitions that have been followed at each step of the test so that it can compare the observed response with the expected response and take necessary action in case of mismatch. This information has been made available to our strategy by subscribing to a call-back function named *Transition Followed* (line 1). This function is invoked whenever a transition is followed in the model. The first and last arguments of this function are the *source* and *target* states of the followed transition respectively. *ilabel* is the instantiated label of the transition that contains the chosen data values for the label parameters (if any). *@trace* is the current trace in the model that our strategy is aiming to cover. *test_step* is the instantiated label of the current observable transition in *@trace*. If the reported *ilabel* is the same as our strategy's current step (line 4), then everything is fine and test execution shall continue with the current trace. If on the other hand there is a mismatch we handle it (line 7, 12).

Pseudocode 2 Handling Deviation

```
function TRANSITION FOLLOWED(source, ilabel, target)
    if ilabel.observable? then
        test_step = @trace.choose_transition
        if ilabel == test_step then
            @trace.prepare_next_transition
        else
            HANDLE DEVIATION(ilabel)
        end
    end
end

function HANDLE DEVIATION(ilabel)
    iprefix = COMPUTE INSTANTIATED PREFIX(ilabel)
    trace = FIND TRACE WITH INSTANTIATED PREFIX(iprefix)
    if trace is present then
        SWITCH TRACE(trace)
    else
        FIND TRACE WITH SYMBOLIC PREFIX(iprefix)
    end
end

function FIND TRACE WITH INSTANTIATED PREFIX(iprefix)
    traces = TRACES WITH INSTANTIATED PREFIX(iprefix)
    traces = REMOVE COVERED TRACES AND SORT BY TRACE ATTEMPT(traces)
    return traces.first
end

function FIND TRACE WITH SYMBOLIC PREFIX(iprefix)
    sprefix = COMPUTE SYMBOLIC PREFIX(iprefix)
    traces = TRACES WITH SYMBOLIC PREFIX(sprefix)
    if traces.empty? then
        @trace.stop_testcase                                   ▷ We have found an error
    else
        traces = REMOVE COVERED TRACES AND SORT BY TRACE ATTEMPT(traces)
        traces = PUT CURRENT TRACE IN FRONT(traces)
        trace, solution = TRACE AND SOLUTION TO CONTINUE WITH(traces, iprefix)
        if trace and solution are present then
            spostfix = trace.compute_symbolic_postfix(sprefix)
            test_case = INSTANTIATE TRACE(spostfix, solution)
            trace.test_case = test_case
            SWITCH TRACE(trace)
        else
            @trace.stop_testcase                               ▷ We have found a deadlock
        end
    end
end
```

We start by computing an instantiated prefix (line 13). *iprefix* is comprised of all the instantiated labels that have been successfully covered since the start of the test case plus the *ilabel* we just received from the SUT. If there is a trace in our test suite with a test case that has exactly the same prefix as *iprefix* then we can immediately switch to that trace and use the remainder of its test case to continue (line 14, 22). We find all the traces with this property (line 23). We remove those traces that have already been covered and we sort the remaining according to their trace attempt in an ascending order (line 24). If the resulting array is not empty then we can use the first member and switch to it (line 16). In models where there is not a lot of instantiation of data values on the transitions, our chance of finding a trace with the exact same instantiated prefix is very high which makes this look up in the test suite very practical since we can immediately continue with one of the already generated test cases. On the other hand there are models where a lot of choosing of data happens on the transitions, especially data that is chosen and returned by the SUT. In such models, look up will usually fail to find a trace in the test suite with the same instantiated prefix. In general, if look up fails then we should try to find a trace based on the symbolic transitions that have been followed so far (line 18, 28). We can compute this symbolic prefix from *iprefix* (line 29). Same as *iprefix*, *sprefix* is constructed from observable transitions and it abstracts from the intermediate unobservables.

Before continuing with the remainder of the solution, let's review the example we gave for the model in figure 3.10 to see how *iprefix* and *sprefix* are constructed. In this example *@trace* is $T_1\,?\,a\,!\,d\,!\,f$. This is the trace the strategy is aiming to cover. *@test_step* is $!\,d(\_y = 10)$. This is the expected response. *ilabel* is $!\,c(\_y = 7)$. This is the observed response. Mismatch is evident and deviation should be handled. *iprefix* is $?\,a\,!\,c(\_y = 7)$. Look up in the test suite will not find any trace with a test case prefixed with $?\,a\,!\,c(\_y = 7)$ thus we need to turn over to computing the symbolic prefix. *sprefix* will be $?\,a\,!\,c$.

We find all the traces for which *sprefix* is a prefix of their symbolic trace when the unobservable transitions are filtered out (line 30). Coming back to our example, there is only one trace in the model with this property and that is $T_1\,?\,a\,T_2\,!\,c\,!\,e$. If we do not consider the unobservables T_1 and T_2 then $?\,a\,!\,c$ is its symbolic prefix. If there is no symbolic trace that has a prefix as *sprefix* then it means receiving of *ilabel* at this point is not allowed by the specification and that *ilabel* is an invalid response. Our strategy is able to detect this error since it is aware of all the valid traces in the model. They were calculated before the start of the test. After the error is reported, the failing trace along with all the other traces for which the failing trace is a prefix are removed from the test suite. Our strategy stops the current test case, picks another trace and continues with test execution (line 32). If on the other hand one or more traces are found with the same symbolic prefix then our strategy should choose one that correctly locates our position in the model after the deviation. Note that because of non-determinism more than one trace could be the correct trace to be on. We gave examples for this situation in the figures 3.13 and 3.14. Traces marked with covered are removed and the rest are sorted according to their trace attempt in an ascending order (line 34). If the strategy's current trace (*@trace*) is also present in the array of traces that have a symbolic prefix as *sprefix* then it is put in front of the array (line 35). This only happens in the case of receiving an *unexpected valuation* from the SUT where the observed response and the expected response have the same label but the valuations are different. It is possible that the current trace can still be used with the new valuation. That is why we give a higher priority to this trace by putting it in front of the other traces. It is like giving a second chance to the current trace before turning over to other candidates for continuation. If we do not give this priority to the current trace then there is a risk of never covering it because it is possible that every time we want to cover this particular trace, a different valuation is received and we always switch from it while it could have also been possible for our strategy to use the current trace and continue with it. In general every time we switch from a trace, its trace attempt is increased by one and if the number of attempts for covering a trace reaches the maximum number of trace attempts configured by the strategy and the trace has not been covered yet, then that trace will not be chosen by our strategy anymore. This is done to avoid infinite looping. Now if the above scenario happens a couple of times in a row then the number of attempts for the current trace hits the maximum trace attempt which will lead to its starvation.

After having the candidate traces sorted, together with *iprefix* they are sent to another function for the final selection (line 36). *iprefix* is an array of instantiated observable labels that have been followed through the model since the start of the test case. The chosen data values for the label parameters up until this point can be extracted from *iprefix*. From all the candidate traces, only those are eligible for continuation that are reachable with the valuations used so far. We perform a reachability analysis for each trace in the array of candidate traces starting from the first element. The first reachable trace in the array is returned along with its solution. The solution contains the data values that should be used for the remaining label parameters on the trace. By doing a reachability analysis we can also be sure that our strategy always chooses the correct trace and if there are more than one correct traces due to non-determinism then the one with the least number of trace attempts will be chosen. Before switching to the new trace, its symbolic postfix is computed and then instantiated with the data values in the solution (line 38, 39, 40). If none of the candidate traces are reachable with the valuations used thus far then it means we have found a deadlock in the model (line 43). In this case the strategy stops the current test case, increases the trace attempt and then picks another trace to continue.

Reviewing of the Example Models

In figure 3.9 we gave an example of a model where it is possible to receive an *unexpected response* with a *different valuation* at the same time and deviate from the current trace. In that example *@trace* is $?\,a\,!\,b\,!\,d\,!\,f$. This is the trace that our strategy is trying to cover. *test_step* is $!\,b\,(\_y = 1)$ which is our strategy's expected response. Instead The response that is observed is $!\,c\,(\_y = 23)$. Following the steps in the solution, the first thing that our strategy does is that it computes *iprefix* to see if there are already any test cases prefixed with $?\,a\,!\,c\,(\_y = 23)$. Look up over the test cases will fail. Our strategy continues with computing the *sprefix* $= ?\,a\,!\,c$ and then finding all the traces in the model that have such a symbolic prefix. Two traces are found: $?\,a\,!\,c\,?\,e\,T\,!\,g$ and $?\,a\,!\,c\,?\,e\,T\,!\,h$. Suppose neither of them are covered yet and both have the same number of trace attempts therefore the array of candidate traces could look like this: $\{?\,a\,!\,c\,?\,e\,T\,!\,g,\ \ ?\,a\,!\,c\,?\,e\,T\,!\,h\}$. Our strategy starts with the first element in the array and inspects whether its *trace-constraint* is solvable with the data value chosen for the label parameter $\_y$ in transition !c. The answer is positive because the constraint solver returns the following solution: {"$\_y\_1$" => 23, "$\_y\_2$" => 2}. $\_y\_2$ refers to the label parameter in transition ?e. This means that if our strategy wants to continue with this trace and be able to reach the end of it, label parameter $\_y$ in ?e should be instantiated to 2. So our strategy does the same by computing the symbolic postfix for the new trace and instantiating it according to the solution. *spostfix* is $?\,e\,T\,!\,g$ and its instantiation which will be used to communicate with the SUT is $?\,e\,(\_y = 2)\,!\,g$. Note that both traces were actually reachable so the reason $?\,a\,!\,c\,?\,e\,T\,!\,g$ was chosen for continuation was simply because of its position in the array of candidate traces. After switching to the new trace test execution continues until we reach state 6 in the model. Here our strategy is expecting to receive !g from the SUT because that is what it has on its current test case, but let us assume that the SUT sends back output !h. Our strategy takes the exact same steps as we just described to resolve the second deviation. Look up in the test suite will fail again because there is no test case prefixed with *iprefix* $= ?\,a\,!\,c\,(\_y = 23)\,?\,e\,(\_y = 2)\,!\,h$. Our strategy then tries to find traces that have a symbolic prefix as *sprefix* $= ?\,a\,!\,c\,?\,e\,!\,h$. One trace is found which is $?\,a\,!\,c\,?\,e\,T\,!\,h$. Its *trace-constraint* is solvable with the valuations chosen so far therefore it can be used. The symbolic postfix is computed for this trace but considering that we are already at the end of the trace, *spostfix* will be empty. Our strategy realizes this and marks $?\,a\,!\,c\,?\,e\,T\,!\,h$ as covered. This example shows that our strategy started test execution with an attempt to cover the left hand side trace in the model but after two deviations it ended up covering the right most trace in the model.

For our last example we are going to take a look back at the model in figure 3.13 where we could deviate from the current trace when we are non-deterministically in more than one state. Suppose state variable x is updated with 2 in transition ?a and our strategy is trying to cover $?\,a\,T_1\,T_3\,!\,b$. Our strategy's expected response is !b while the SUT non-deterministically responds with !c. We mentioned as part of the requirements that our strategy should correctly locate the SUT's position in the model after the deviation has happened. With $x = 2$ the only correct trace is $?\,a\,T_2\,T_4\,!\,c$. To make the example more interesting let's assume that the look up fails to find any test case prefixed with *iprefix* $= ?\,a\,(\_y = 2)\,!\,c$ and our strategy has to go through all the traces in the test suite to find those that have a symbolic prefix like $?\,a\,!\,c$. Two traces are found with this property namely $?\,a\,T_2\,T_4\,!\,c$ and $?\,a\,T_2\,T_5\,!\,c$ but we know when label parameter $\_y$ in transition ?a is instantiated to 2, it is not possible for the SUT to be on $?\,a\,T_2\,T_5\,!\,c$. Our strategy is able to rule out this incorrect trace in its selection because the reachability analysis reveals that the *trace-constraint* for this trace is not solvable with $\{"\_y_0" => 2\}$.

5.2.4 Non-deterministically Covering More than One Trace

The test suite which is created by our strategy contains an instantiation for every trace in the model. This is because during test generation all the possible traces through the model are extracted. Due to non-determinism it is possible to cover the entire model by testing a subset of the traces in the test suite. An example is given in figure 5.11. This is not an interesting model from a modeling perspective but it provides a simple example of the relationship between non-determinism and coverage in TestManager. By sending input ?a to the SUT, all three transitions starting from state 0 will be followed and we will end up in states 1, 2 and 3 at the same time. There is an entity in TestManager that keeps track of coverage information. According to this entity three transitions labeled with ?a are all covered at this moment. If we receive output !b then the transition labeled with !b is covered and if instead we receive output !c then both transitions labeled with !c will be followed in the model and consequently marked as covered. Both traces $(0 \to 2 \to 4)$ and $(0 \to 3 \to 4)$ are non-deterministically covered when the SUT is tested with $?\,a!\,c$. The entire model can be covered by just testing it with two test cases namely $?\,a!\,b$ and $?\,a!\,c$. We want our strategy to realize this and only test this model with these two test cases. In other words, we do not want our strategy to test $?\,a!\,c$ twice. Testing it once is enough for covering both traces. Our strategy on the other hand creates the following test suite: $\{?\,a!\,b,\ ?\,a!\,c,\ ?\,a!\,c\}$. This is because our strategy extracts every symbolic trace from this model: $\{(0 \to 1 \to 4),\ (0 \to 2 \to 4),\ (0 \to 3 \to 4)\}$.

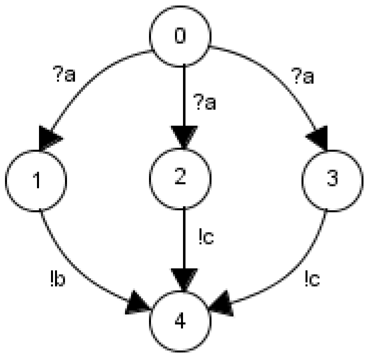

Figure 5.11: Test case covering more than one trace

Calculating every trace through the model becomes more problematic in the presence of non-deterministic unobservable transitions. If there are multiple outgoing unobservable transitions from a state and they are all taken at the same time then the number of traces that are non-deterministically covered together, could grow considerably. The model in figure 5.12 shows a very simplified example of this situation.

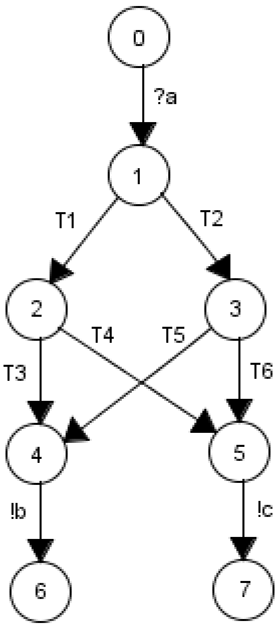

Figure 5.12: Test case covering more than one trace

Unobservable transitions are labeled with T. Suppose they are all taken after sending input $?\,a$ to the SUT. There are four symbolic traces in this model: $\{?\,aT_1T_3!\,b,\ \ ?\,aT_2T_5!\,b,\ \ ?\,aT_1T_4!\,c,\ \ ?\,aT_2T_6!\,c\}$. Our strategy finds and instantiates all of them which results in creating the following test suite: $\{?\,a!\,b,\ \ ?\,a!\,b,\ \ ?\,a!\,c,\ \ ?\,a!\,c\}$. Similar to the previous example this model can be completely covered by just testing a subset of the traces in the test suite. We don't want our strategy to test $?\,a!\,b$ twice, one time for covering $?\,aT_1T_3!\,b$ and another time to cover $?\,aT_2T_5!\,b$. Both traces are non-deterministically covered the first time the SUT is tested with $?\,a!\,b$. The same applies to $?\,a!\,c$.

Our strategy calculates every path through the unobservable part of the model while this part does not really matter in the eyes of testing. When testing the SUT with $?\,a!\,b$ what is important for us is that we observe $!\,b$ after sending input $?\,a$. In general we do not care how we get from one state in the model to another as long as the observable part is observed. The unobservable part of the model is not visible to the SUT's environment after all. If we abstract from the unobservable transitions in the model during test generation and we stop calculating every path through them then $?\,aT_1T_3!\,b$ and $?\,aT_2T_5!\,b$ become the same trace. The same applies to $?\,aT_1T_4!\,c$ and $?\,aT_2T_6!\,c$. Our test suite will then be reduced to $\{?\,a!\,b,\ \ ?\,a!\,c\}$ which contains the minimum amount of test cases for covering this model.

Having said that, considering the way our strategy operates we need to calculate every trace through the model including unobservable transitions. We cannot afford to abstract from the unobservable part of the model because as we have already explained in 3.1.3 unobservable transitions have constraints and/or update mappings on them and they have an impact on test generation and also the reachability algorithm. We cannot consider $?\,aT_1T_3!\,b$ and $?\,aT_2T_5!\,b$ the same traces. Although they have the same sequence of observable transitions but it is possible that they are reachable with completely different valuations due to the constraints on the intervening unobservables. The model in this example does not show this but $?\,aT_2T_4!\,c$ and $?\,aT_2T_5!\,c$ in figure 3.13 are two traces with the same labels but only one of them is reachable when state variable x is 2. We also cannot already know which path the SUT will take through the model during test generation since it depends on the values that are chosen for the label parameters during testing. For these reasons our strategy considers traces like $?\,aT_1T_3!\,b$ and $?\,aT_2T_5!\,b$ separate from each other and generates a test case for each of them.

We saw that due to non-determinism it is possible that when our strategy covers a trace, other traces are also covered in the model. If instead of only marking the current trace as covered, we identify all the other traces in the test suite that have been non-deterministically covered because of the current test case and also mark them as covered, then our strategy will not ever try to cover a trace that has already been covered before. Traces that are covered at the same time because of non-determinism have the same sequence of observable labels and they are all reachable with the same valuation. After successfully covering the current trace and marking it as covered and before starting with the next test case, our strategy finds all the traces in the test suite that have the same sequence of observable labels as the current trace and checks if they are reachable with the valuations used for covering the current trace. If that is the case then they will also be marked as covered. For example consider the model in figure 3.13. After successfully testing the SUT with $?\,a(\_y = 1)\,!\,c$ both traces $?\,aT_2T_4!\,c$ and $?\,aT_2T_5!\,c$ are marked as covered in the test suite while a successful execution of $?\,a(\_y = 2)\,!\,c$ only covers $?\,aT_2T_4!\,c$. Our strategy still has to test the SUT with $?\,a!\,c$ one or more times to finally cover $?\,aT_2T_5!\,c$.

We ignore non-determinism during test generation and we solve it later during test execution by finding the traces that are non-deterministically followed together and marking all of them as covered. With this approach we might not have a minimum test suite at the beginning of the test but we are certainly moving towards it during test execution and by the time we are done, we have tested the SUT with the same number of test cases as in the minimum test suite.

5.3 Experimenting with the *FEI* Model

Table 5.1 shows the results obtained from running *Trace Coverage Strategy* on the *FEI* model. Each row provides information about the execution of a set of test cases T of length n against this model. We start with the definition of *trace coverage* in the sixth column. The *trace coverage* of a set of executed test cases T of length n is the percentage of all the reachable symbolic traces of the bounded model until depth n that were successfully covered by test cases from T. We repeat that the depth to which we limit the model corresponds to the number of observable transitions in a symbolic trace (Unobservable transitions do not count). The total number of symbolic traces that have n observable transitions and their average length are given in the second and third columns respectively. The big difference between the average length of symbolic traces and n in each row shows that this model is mainly comprised of unobservable transitions. The fourth column shows the number of symbolic traces that are reachable and since test cases are an instantiation of these traces then it also gives the size of T. The fifth column represents the total time spent by *Trace Coverage Strategy* to generate and execute the test cases in T. Finally the last column gives the percentage of the transitions that were covered due to the execution of test cases in T.

Length of test cases	Total number of symbolic traces	Average length of symbolic traces	Number of reachable symbolic traces or size of test suite	Total time	Trace coverage	Transition coverage
$n = 1$	15	18	13	0m1.396s	100%	12.9%
$n = 2$	3461	43	21	0m46.228s	100%	18.28%
$n = 3$	27835	45	273	09m18.853s	100%	21.35%
$n = 4$	—	—	—	—	—	—

Table 5.1: *Trace Coverage Strategy* vs the *FEI* model

The table does not contain any entries for test cases of length 4. This is because the total number of symbolic traces that have 4 observable transitions were so many that after 10 hours of running, our strategy was not even able to extract them, let alone perform a reachability analysis on them and execute them. What happens is that after a while the main memory becomes full and the operating system starts swapping pages of data back and forth between the main memory and the hard disk which ultimately leads to thrashing and a massive drop in CPU's utilization. We are confident that the total number of symbolic traces with 4 observable transitions is more than 4884677. The reason is that 4884677 is the maximum number of symbolic traces that our strategy is able to extract from the model without getting blocked on I/O.

We are interested in finding all the reachable symbolic traces with n observable transitions because once they are found and instantiated, we will have created our finite test suite. With our current approach, we extract all the symbolic traces that have n observable transitions and we narrow them down to our desired set by performing a reachability analysis on each of them. The problem is that the initial set of traces is too large that we cannot even pass the extraction phase let alone perform a reachability analysis on them which is far more computationally intensive compared to their extraction.

There is a workaround to this problem. By looking at the second and fourth columns in the table, we can see that the rate at which the reachable symbolic traces grow from one step to the next is a lot less than that of the total number of symbolic traces. Reachable traces of length n can be calculated from the set of reachable traces of length $n-1$. This solution completely changes the way tests are generated by the strategy. Our strategy no longer needs to find all the symbolic traces with n observable transitions which could be in the millions in order to get to a very small subset of reachable traces of length n. In this new approach we start from the set of reachable traces of length one R_1. We find all the symbolic traces of length 2 that are an extension of R_1. Note that we are not extracting all the symbolic traces of length 2 anymore, we are only finding those whose prefix is in R_1. By performing reachability analysis on the resulting set, we end up with the reachable traces of length two R_2. If we carry on like this, then we are eventually able to calculate the reachable traces of length n (R_n) without confronting with the explosion resulted from extracting all the symbolic traces of length n. We implemented this new approach of test generation for our strategy and we tested the *FEI* model again. The results are presented in table 5.2.

Length of test cases	Traces of length n extended from R_{n-1}	R_n Reachable traces of length n or size of test suite	Total time	Trace coverage	Transition coverage
$n = 1$	0	13	0m1.446s	100%	12.9%
$n = 2$	3454	21	0m46.718s	100%	18.28%
$n = 3$	315	273	1m3.905s	100%	21.35%
$n = 4$	72534	441	36m33.956s	100%	22.43%
$n = 5$	6587	5709	47m13.808s	100%	26.11%
$n = 6$	1516323	—	—	—	—

Table 5.2: *Trace Coverage Strategy* with optimized test generation vs the *FEI* model

Although we were successful in extending R_5 with another observable transition but unfortunately we were not able to calculate R_6 and test the model with 6 observable steps. The reason is that after 42 hours of running, our strategy stopped because of a runtime error. However, the optimized solution for test generation seems promising. Once the error is resolved, *Trace Coverage Strategy* should be able to cover a great portion of the model.

Chapter 6

Conclusion

We have designed and implemented a *Trace Coverage Strategy* that checks the input-output conformance of a system under test with respect to a model of its specification. Test cases in the ***ioco***-testing theory are constructed from traces of the model obtained from the specification. The resulting test suite is often infinite because of the high number of combinations of data values and the presence of loops in the specification. Executing such a test suite is not practical since it cannot be carried out in a reasonable time. We have restricted the set of test cases by defining a finite subset of traces of the specification. By limiting the model to a certain depth and unrolling the loops in it we obtain a tree-like structure where the number of traces becomes finite. The resulting tree is symbolic which could still result in an infinite number of test cases since label parameters can be instantiated from an infinite domain of data values. That is why for every trace leading to a leaf only one test case is generated. In summary by limiting the model on the length of its traces and confining to one instantiation for every trace we end up with a finite set of bounded length test cases. The finite test set is created by our strategy.

We have implemented a reachability algorithm that plays an important role in our *Trace Coverage Strategy* both in the generation of test cases and later in their execution. Given a symbolic trace, the algorithm in conjunction with a constraint solver decides if the trace is ever reachable that is whether the label parameters on the transitions can be instantiated in such a way so that all the constraints along the trace can be satisfied simultaneously. If the trace is reachable then the valuations returned by the constraint solver are used to instantiate the trace otherwise the trace is unreachable within its length and thus is removed from the test suite. Our strategy handles unobservable transitions in the models by including them in the reachability analysis and finally removing them from the test cases. Our reachability algorithm can also be used for other purposes like static model checking. It can be used to check if a model contains any deadlock traces.

After having a test case generated for every reachable symbolic trace in the model, test execution is started. Our strategy picks a trace from the test suite and tries to cover it but during testing it is not always possible to remain on the trace that we want to cover because the SUT might deviate from the strategy's intended trace by giving another response. We have explained all the possible scenarios of deviation, both deterministic and non-deterministic variants. We have implemented a single solution that is able to handle all of these deviations efficiently. Using our reachability algorithm we are always able to find the correct trace in the model after a deviation has occurred and continue with test execution. Our strategy is able to handle as many deviations as possible during testing. Our strategy keeps track of all the traces that are non-deterministically covered together in a single test execution and marks all of them as covered. This way our strategy will not ever try to cover a trace that has already been covered because of another test case. Maximum number of trace attempts has been defined to prevent our strategy from getting caught in an infinite loop trying to cover a trace. Every time our strategy switches from a trace, the number of trace attempts for that trace is increased by one and if the upper bound is reached then that trace will not be picked by our strategy anymore.

We validated our strategy by running it on an industrial model. We experienced an explosion of symbolic traces when testing the model with four steps. As a result we provided a solution that optimized our test generation process and we retested the model using the new approach and we presented the new results.

Trace Coverage has grown to become a full fledged strategy that can correctly check for *ioco*-conformance within Axini TestManager. Assuming that the model of the specification is correct, then the test suite that is created from it by our strategy contains only valid test cases. We know they are valid because each of them are an instantiation of one of the reachable traces in the model, i.e., every trace extracted from the model goes through the reachability algorithm. As for test execution if the SUT follows a trace which is different than our strategy's current trace then we can always readjust by finding the new trace even when we are non-deterministically in more than one state in the model. Under these considerations if the SUT is *ioco*-conforming to its specification, then we should be able to observe each and every one of the test cases in the SUT as well. If it becomes evident that the SUT has failed on one of the test cases then it means that the SUT is not in conformance with its specification and that there is an error in the implementation of the SUT. Our strategy is able to detect that during testing. In the end the coverage metric that we are looking to achieve is 100 percent of all the reachable traces in the bounded model and if we do not achieve that, its because of the failing traces in the SUT.

Future Work

A SUT may exhibit parallel behavior. In this case, the SUT's behavior is captured by multiple concurrent processes where each process is modeled with its own symbolic transition system. The way in which processes are combined is called *parallel composition* [AML]. They can be either dependent or independent of each other. When they are independent, every interleaving of their transition systems are allowed. As for dependent processes, they are still fully interleaved except on synchronizing transitions. At the current moment *Trace Coverage Strategy* does not support *parallel composition*. It only operates on systems modeled with a single transition system.

Bibliography

[AML] *Axini Modeling Language Tutorial.*

[Apt03] Krzysztof Apt. *Principles of constraint programming.* Cambridge University Press, 2003.

[BJK+05] Manfred Broy, Bengt Jonsson, Joost-Pieter Katoen, Martin Leucker, and Alexander Pretschner. *Model-based testing of reactive systems: advanced lectures*, volume 3472. Springer, 2005.

[Dia] Daniel Diaz. Prolog compiler with constraint solving over finite domains. Software. URL: http://www.gprolog.org/.

[ESSD08] Steve Easterbrook, Janice Singer, Margaret-Anne Storey, and Daniela Damian. Selecting empirical methods for software engineering research. In *Guide to advanced empirical software engineering*, pages 285–311. Springer, 2008.

[FTW04] Lars Frantzen, Jan Tretmans, and Tim AC Willemse. Test generation based on symbolic specifications. In *International Workshop on Formal Approaches to Software Testing*, pages 1–15. Springer, 2004.

[FTW06] Lars Frantzen, Jan Tretmans, and Tim AC Willemse. A symbolic framework for model-based testing. In *Formal approaches to software testing and runtime verification*, pages 40–54. Springer, 2006.

[McC96] Steve McConnell. Software quality at top speed. *Software Development*, 4(8):38–42, 1996.

[MSB11] Glenford J Myers, Corey Sandler, and Tom Badgett. *The art of software testing.* John Wiley & Sons, 2011.

[Sie09] Floor Sietsma. A Case Study in Formal Testing and an Algorithm for Automatic Test Case Generation with Symbolic Transition Systems. Master's thesis, University of Amsterdam, 2009.

[Tre08] Jan Tretmans. Model based testing with labelled transition systems. In *Formal methods and testing*, pages 1–38. Springer, 2008.

[VT] Michele Volpato and Jan Tretmans. Test selection for the ioco framework.