Test-Based Modelling - Learning a Model From a System's Implementation

Lennart Tange

23rd August 2011

Master Course Software Engineering – Thesis

Thesis Supervisor: Jan van Eijck

Internship Supervisor: Machiel van der Bijl

Company: Axini

Availability: Public Domain

University of Amsterdam

Author	**Lennart Tange**
Address	Leeuwenstraat 68 5645BE Eindhoven The Netherlands
Phone	+31 6 54902637
E-mail	lennarttange@gmail.com

Location of demo application: `http://lennarttange.nl`

Abstract

This thesis presents an algorithm for test-based modelling in order to support the modelling process of model-based testing of an existing software implementation. Other known learning theories were unable to deal with real-world testing situations, such as non-determinism, inputs separated from the outputs or suffer from state space explosion. This thesis describes how to observe an implementation as a black box and how to detect loops using a simulation pre-order relation. The learned labelled transition system respects the properties of the **ioco** relation which is often used in modern model-based testing techniques.

Preface

This thesis is submitted in support of the candidature for a master's degree in Software Engineering at the University of Amsterdam. During the first period of my research, I had the intention to implement an existing learning method in a tool, then do a case study using the tool. Also I hoped I could somehow improve the chosen existing method. During my exploration, I found that none of the existing theories really worked in a setting I intended to use it for. By experimenting and reasoning, I was able to come up with a new algorithm for an upcoming field of activity.

For me this research was the perfect mixture of theoretical and practical challenges. Also, on most of the lectures on model-based testing I attended, the question arose "Can't we do this modelling part automatically?", so I knew of my research questions were useful to the real world.

I am glad I was given the opportunity to work out this algorithm and would therefore like to thank in particular Machiel van der Bijl[1], for spending numerous hours on discussing and sharing his insights and Jan van Eijck[2] for his critical view and helping me in the right directions when I got stuck on the theoretical path.

Also thanks to Tim Willemse[3], Jan Tretmans[4] and Fides Aarts[5] for sharing their knowledge and experience with me.

Thanks to all professors at the UvA involved with the master Software Engineering for putting together this master course which, besides being very interesting and intensive, really made me grow as a person.

Thanks to the *professionals* at Axini for all their help and advises.

At last, I would like to thank my family and friends for showing their interest in my work and now and then providing the necessary distraction from this research.

[1]Axini
[2]Centrum Wiskunde & Informatica, University of Amsterdam
[3]Technical University of Eindhoven
[4]Embedded Systems Institute, Radboud University Nijmegen
[5]Radboud University Nijmegen

Contents

Chapter 1

Introduction

Model-based testing of software systems is an upcoming technique that can completely automate the software testing process by generating tests, performing them and checking the results, all by addressing the system under test as a black box. Axini has proven in different fields of software development, model-based testing results in a better test coverage as well as lower testing costs.

One of the problems at most current software development departments is that models to test software systems against are difficult to construct. Because the people working there are not familiar with making models, so the modelling process takes up a considerable amount of time.

So how could we help this process? Can we learn a model of a system by observing it? Is there a way to fully automate the modelling process when there already is an implementation? What requirements would such a modelling method have and do these methods already exist?

This research aimed at answering these questions. Therefore, this thesis starts by explaining the kind of models used in model-based testing and analyse the existing techniques. After explaining why none of these existing techniques really fit our setting, we present a new algorithm for *model learning*, also known as *test-based modelling*. The correctness of this algorithm has been proved and to demonstrate this algorithm and to support further research, a tool implementing the algorithm has been developed. This tool, called Speculaas, can be found at `http://lennarttange.nl` and is explained in the appendix.

In this thesis, when speaking of the Learner, this is the learning and observing part that obtains the model, the algorithm. The Teacher is the software implementation that needs to be modelled or learned, which can also be referred to as ‘System Under Test’ (SUT) or ‘implementation’. A ‘model’ in the setting of this thesis is a transition system which describes the behaviour of a system.

Chapter 2

Preliminaries - Model-Based Testing

Tretmans[7, 8, 9] defined the **ioco** relation which is often used in modern model-based testing techniques, at Axini for example. This chapter explains and defines this relation based on Tretmans' research.

2.1 Labelled Transition Systems

Labelled Transition Systems ($\mathcal{LTS}$) are the preferred abstraction of systems in modern testing techniques. Compared with finite state machines, $\mathcal{LTS}$ tend to deal better with non-determinism and parallelism. Because it is necessary to separate inputs from outputs, general $\mathcal{LTS}$ do not suffice. Therefore $\mathcal{IOLTS}$ is used. To explain the **ioco** relation, it is necessary to define $\mathcal{IOLTS}$ first:

Definition 1 (IOLTS)**.** *[9]*

An $\mathcal{LTS}$ with separated inputs and outputs, an $\mathcal{IOLTS}$,
is a 5-tuple $\langle Q, L_I, L_O, T, q_0 \rangle$, where:

- *Q is the finite set of states*
- *L_I is the finite set of input labels and L_O the finite set of output labels, where $L_I \cap L_O = \emptyset$*
- *$\tau \notin L$, where $L = L_I \cup L_O$*
- *T is the set of transitions (transition $\subseteq Q \times (L \cup \{\tau\}) \times Q$)*
- *q_0 is the initial state where $q_0 \in Q$*

Transitions between states can be denoted as an arrow with a label on it. Some arrow notations used later are defined in Definition 2.

Definition 2 (arrow notations)**.**

Let q, q' be states of a $\mathcal{LTS}$ $\langle Q, L_I, L_O, T, q_0 \rangle$, $\lambda, \lambda_i \in L$ and $\sigma \in L^*$.

$$\begin{array}{lll}
q \xrightarrow{\lambda} q' & \overset{def}{\iff} & q \textit{ reaches state } q' \textit{ by doing a transition with the label } \lambda,\ q \xrightarrow{\lambda} q' \in T \\
q \xrightarrow{\lambda} & \overset{def}{\iff} & \exists_{q'} \left[q \xrightarrow{\lambda} q' \right] \\
q \xRightarrow{\varepsilon} q' & \overset{def}{\iff} & q = q' \textit{ or } q \xrightarrow{\tau \cdot \ldots \cdot \tau} q' \\
q \xRightarrow{\lambda} q' & \overset{def}{\iff} & \exists_{q_1, q_2} \left[q_1, q_2 \in Q : q \xRightarrow{\varepsilon} q_1 \xrightarrow{\lambda} q_2 \xRightarrow{\varepsilon} q' \right] \\
q \xRightarrow{\lambda_1 \cdot \ldots \cdot \lambda_n} q' & \overset{def}{\iff} & \exists_{q_0 \ldots q_n} \left[q_0 \ldots q_n \in Q : q = q_0 \xRightarrow{\lambda_1} q_1 \xRightarrow{\lambda_2} \ldots \xRightarrow{\lambda_n} q_n = q' \right] \\
q \xRightarrow{\sigma} & \overset{def}{\iff} & \exists_{q'} \left[q \xRightarrow{\sigma} q' \right]
\end{array}$$

When observing the behaviour of a system, it is found useful not only to observe the output actions, but also to observe the absence of output. A state that produces no output is called ‘quiescent’.

Definition 3 (quiescence)**.** *[7]*

A state q in an $\mathcal{LTS}$ $\langle Q, L_I, L_O, T, q_0 \rangle$ is quiescent, denoted as $\delta(q)$, if $\neg \exists_x \left[x \in L_O \cup \{\tau\} : q \xrightarrow{x} \right]$.

$\delta \notin L$

In some cases, it is useful to treat quiescence as a label. In those cases L_δ is used, where $L_\delta = L \cup \{\delta\}$. Traces of such labels are called suspension traces or Straces.

Definition 4 (quiescence-arrow, traces and Straces)**.** *[9]*

A $\sigma \in L^*$ is called a trace and $\sigma \in L_\delta^*$, is called a suspension trace or Strace.

Let q be a state in an $\mathcal{LTS}$ $\langle Q, L_I, L_O, T, q_0 \rangle$ s:

$$\begin{array}{lll}
q \xrightarrow{\delta} & \overset{def}{=} & \delta(q) \\
traces(q) & \overset{def}{=} & \{\sigma \in L^* \mid q \xRightarrow{\sigma}\} \\
Straces(q) & \overset{def}{=} & \{\sigma \in L_\delta^* \mid q \xRightarrow{\sigma}\} \\
Straces(s) & \overset{def}{=} & Straces(q_0)
\end{array}$$

The **ioco** definition uses *out* and *after* as defined in Definition 5

Definition 5 (out and after)**.** *[9]*

Let p be a state in a transition system $\langle Q, L_I, L_O, T, q_0 \rangle$ s, $P \subseteq Q$ and $\sigma \in L^*$. Then:

$$\begin{array}{lll}
out(p) & \overset{def}{=} & \{x \in L_O \mid p \xRightarrow{x}\} \cup \{\delta \mid \delta(p)\} \\
out(P) & \overset{def}{=} & \bigcup \{out(q) \mid q \in P\} \\
p \textit{ after } \sigma & \overset{def}{=} & \{p' \mid p \xRightarrow{\sigma} p'\} \\
s \textit{ after } \sigma & \overset{def}{=} & q_0 \textit{ after } \sigma
\end{array}$$

The set of labels from a state or to a state are called *from* and *to* and the set of reachable states is called *reachable*. Definition 6 shows the exact meaning of *from*, *to* and *reachable*.

Definition 6 (from, to and reachable)**.**

Let q be a state in a transition system $\langle Q, L_I, L_O, T, q_0 \rangle$. Then the sets of outgoing transitions, ingoing transitions and reachable states are defined as:

$$\begin{array}{lll}
from(q) & \overset{def}{=} & \{x \in L \cup \{\tau\} \mid q \xrightarrow{x}\} \cup \{\delta \mid \delta(q)\} \\
to(q) & \overset{def}{=} & \{x \in L \cup \{\tau\} \mid \exists_p \left[p \in Q : p \xrightarrow{x} q \right]\} \\
reachable(q) & \overset{def}{=} & \{q' \in Q \mid \exists_\sigma \left[\sigma \in L_\delta^* : q \xRightarrow{\sigma} q' \right]\}
\end{array}$$

2.2 An $\mathcal{IOLTS}$ Example

Figure 2.1 shows an example $\mathcal{IOLTS}$ of a coffee machine that gives coffee when the button is pressed unless there is not enough water.

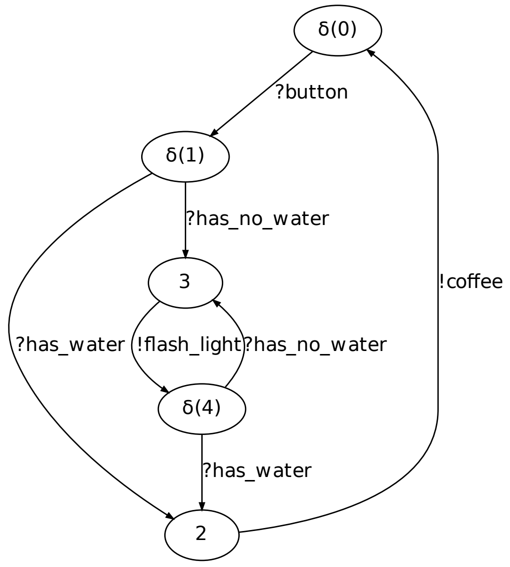

Figure 2.1: An example $\mathcal{IOLTS}$ of a simple coffee machine

Inputs can be recognised in visualisations by a question mark prefix, while output labels have a exclamation mark prefix. So the coffee machine has three types of input (`button`, `has_water` and `has_no_water`) and two types of output (`flash_light` and `coffee`).

State 0, the start state or q_0, is quiescent (denoted by $\delta(0)$). So the coffee machine does nothing until input is given. When the machine gets a `button` (which is an input label), it goes to state 1 (q_1) ($q_0 \xrightarrow{\texttt{button}} q_1$). Again, the machine produces no output but waits until it receives either `has_water` or `has_no_water`. Under normal conditions, when a coffee machine has enough water the trace through the system is $q_0 \overset{\sigma}{\Longrightarrow} q_0$, where σ is the trace `button` $\cdot$ `has_water` $\cdot$ `coffee`. Or, when the coffee machine is called i, $out(i\ after\ \texttt{button} \cdot \texttt{water})$ is $\{\texttt{coffee}\}$.

When instead of `has_water`, `has_no_water` was received, a light flashes `flash_light` until there is enough water.

2.3 The ioco Relation

ioco, which stands for 'input-output conformance', is a relation between the transition systems of a specification s and an implementation i. It is meant to give a notion of when an implementation can be accepted by the specification.

The implementation is assumed to be input enabled, meaning at all times all input is accepted by the implementation. An example of such transition systems are $\mathcal{IOTS}$.

Definition 7 (input-output transition system)**.** *[9]*

An input-output transition system or $\mathcal{IOTS}$ *is an* $\mathcal{IOLTS}$ $\langle Q, L_I, L_O, T, q_0\rangle$ *where all input actions are enabled in any reachable state:*

$$\forall_{q,\lambda}\left[q \in reachable(q0), \lambda \in L_I : q \overset{\lambda}{\Longrightarrow}\right]$$

The assumption an implementation can be modelled into an $\mathcal{IOTS}$ is called the *test assumption.*

Figure 2.2 shows an example $\mathcal{IOTS}$ that may represent the coffee machine of the previous example (Figure 2.1). The $\mathcal{IOTS}$ has defined all input action in each state. For example, when the button is pressed for a second time, nothing happens; the machine remains in the same state.

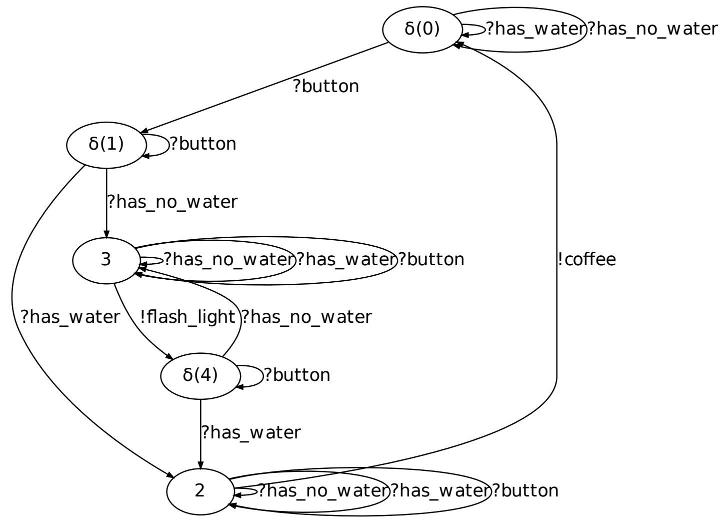

Figure 2.2: An example $\mathcal{IOTS}$ of a simple coffee machine

For specifying the behaviour of a model, $\mathcal{IOLTS}$ are easier to use, since only important traces have to be modelled while $\mathcal{IOTS}$ requires all possible traces in the model. But for testing the behaviour of an implementation (a Teacher) it is useful to test all of its behaviour and see how that is consistent with the specified behaviour.

Definition 8 (**ioco**)**.** *[9]*

Given a set of input labels L_I *and output labels* L_O*, the definition* **ioco** $\subseteq \mathcal{IOTS} \times \mathcal{IOLTS}$ *is defined as follows:*

$$i \ \textbf{\textit{ioco}}\ s \overset{def}{\Longleftrightarrow} \forall_\sigma\left[\sigma \in Straces(s) : out(i \ \ after \ \ \sigma) \subseteq out(s \ \ after \ \ \sigma)\right]$$

Intuitively, the **ioco** relation holds if an implementation does not show more output behaviour for the specified suspension traces than the specification does. For example, using the coffee machine $\mathcal{IOLTS}$ introduced in Figure 2.1 as a specification model, the **ioco** relation would hold if Figure 2.2 is the model of the implementation. The relation would not hold if the implementation would produce coffee in any other state. The relation could still hold, when the implementation is as shown in Figure 2.3, where a second press on the button cancels the process (resetting the coffee machine to its initial state 0), because behaviour of the second press on a button was not defined by the specification.

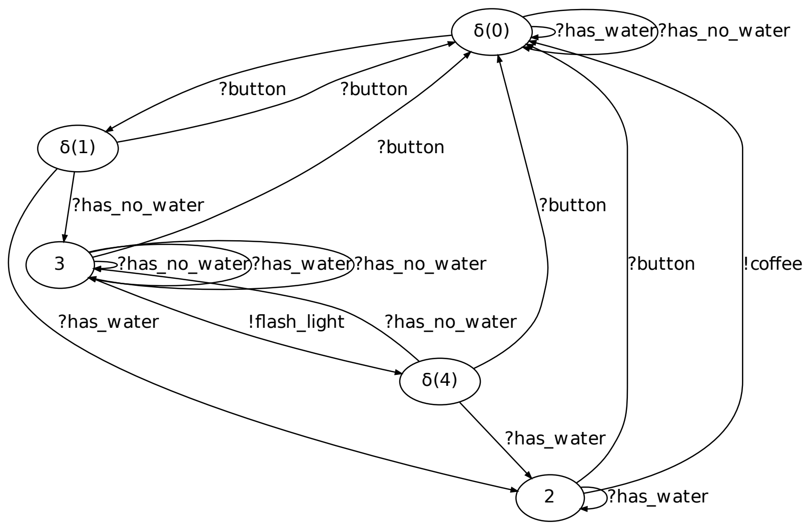

Figure 2.3: Another possible implementation of the coffee machine

Chapter 3

Model Learning

For an automatically learned model to be most useful in the context of model-based testing, the model should be compact to allow designers to decide whether the implementation does what it should do. Also, the learned model should be able to be used for regression testing the implementation. Testing the implementation with the model learned from the same implementation should not result in errors.

At the moment there are several theories to learn a model. This section gives a short description of how a model can be learned using existing methods and what problems a method in the setting of this research should be able to deal with.

3.1 Angluin

Most experiments on model learning are based on the algorithm of Angluin[3]. This algorithm is based on the assumption that the Teacher can respond to two different kinds of questions of the Learner:

- membership queries – Which determines whether a trace is accepted by the Teacher or not (`yes` or `no`);
- equivalence queries – Which determines if the current learned model is equivalent with $\mathcal{I}$. (`yes` or a counterexample trace)

By performing membership queries an observation table is built, which has prefix traces on the rows, suffix traces on the columns and a mapping to whether the concatenated trace is accepted or not in the cells.

Every once in a while a model is extracted and an equivalence query is performed to check if the learned model is correct and, if not, what other membership queries to test. Once the equivalence query passes, the learned model is equivalent with the implementation.

The Angluin algorithm is based on the assumption the implementation can be seen as a deterministic finite-state automaton. Non-deterministic behaviour (when a trace can lead to different states) is therefore not supported. Practical cases however are often not deterministic.

The research of Bollig et al.[5] shows a way to use the Angluin algorithm on non-deterministic finite-state automata, which also further minimises the learned model. However this method does not seem to fit the current testing techniques, since (black-box) Teachers, do not have the ability to handle the described queries and the **ioco** theory cannot be applied directly to finite state machines.

3.1.1 LearnLib

Aarts et al.[1] has done several case studies with the Angluin algorithm implemented (with a few improvements) in the LearnLib tool [6]. LearnLib learns deterministic behaviour modelled in a Mealy Machine, where input and output is put on a single transition. After learning the Mealy Machine, Aarts et al. transformed it to an $\mathcal{LTS}$ to test the learned model with a given model[2]. With this research, Aarts et al. showed it is possible to learn a model that respects testing theories with combining a lot of tools, but does so far only work for deterministic systems.

3.2 Willemse

Willemse learned labelled transition systems by using the **ioco** theory. He showed this theory is not only usable for model-based testing, but also for learning models[10]. By taking an empty model as a starting point, the model expands when it is tested on the **ioco** relation and the counterexamples are added to the model (an empty specification cannot be **ioco** with some implementation that behaves in any way).

Willemse added several heuristics to reduce the trees size and learning time, but loops are not detected, so the model (tree shaped) expands exponentially.

3.3 Why We Need a New Algorithm

The Angluin algorithm is hard to use in practical situations since most software implementations are not able to handle the queries as described. Furthermore, the efficient way of building the observation table would not be relevant, because the system this research intends to learn are prefix-closed: whenever $\sigma \cdot x$ is accepted, where $\sigma \in L^*$ and $x \in L$, σ will also be accepted by the system.

The Willemse approach is pretty straightforward and is able to deal with the fact that the output behaviour of the Teacher cannot be enforced, but the learned models have the problem of expanding exponentially. As for the heuristics added, one of the heuristics is based on more prescience than only in- and output labels of the system. Since this research aims to learn a model with minimal prior knowledge of a system, this heuristic cannot be used.

Both algorithms have their advantages, but none seems to fit the goal. Therefore a new method is necessary.

3.4 Learning a Precise Model

Because a Teacher should in every state be able to do an input action (because the **ioco** definitions requires the implementation to be an $\mathcal{IOTS}$), the visualised transition system of the Teacher contains loops (or would have infinite states, which is in contradiction with its definition). In the setting of this research, it is only possible to learn a specification by observing the input and output behaviour of the Teacher.

	Angluin[3]	Willemse[10]	A New Algorithm
Support for Non-determinism		✓	✓
Inputs and Outputs Separated		✓	✓
Loop Detection	✓		✓
Learned Model	Finite State Machine	$\mathcal{IOLTS}$	$\mathcal{IOLTS}$
Behaviour Observation	Membership Query	Input-Output Behaviour	Input-Output Behaviour
Observations Overview	Observation Table	Tree	(Unknown at this point)
Stop Condition	Equivalence Query	Maximum Depth	implementation **obsioco** learned model

Table 3.1: Comparison summary of learning algorithms

One of the problems of having this setting is that it is impossible to observe traces with an infinite length (it would require infinite time, which is, obviously, very impractical). Therefore, it is not possible to validate there is a loop in the system.

Another problem arises when the Teacher behaves non-deterministically, which implies that it is impossible to conclude all possible behaviour of a system is observed. Imagine an implementation of a coin that can be tossed to be the Teacher. When observing the behaviour of the coin after tossing, it is possible that the tossing results every time in a `tail`. It is likely for a normal coin toss to result in a `head` after a while, but unless it is observed, it is impossible to be sure. Also, it is impossible to conclude at any time the tossing will never result in a `head`, only that it becomes very unlikely.

To deal with these uncertainties and to keep the learned model compact, it is useful to predict loops and the possibility of having non-determinism in the implementation. To make these predictions, it is necessary to add extra information to the observations.

Because it is always possible this added information will not get falsified by further observing (for the same reasons as described above), it will always be necessary for complex implementations to get the learned model validated by a person. On top of this, an implementation may behave different than expected. Hence, the model learned based on this behaviour also differs from expectation.

3.5 The obsioco Relation

In Section 2.3, the definition of **ioco** was given as defined by Tretmans[7, 8, 9]. However, because the previous section showed that it is impossible to observe all behaviour and the learned model will probably be able to generate longer traces (since it may contain loops), the observations, will never be **ioco** with the learned model. Therefore a new relation **obsioco** is defined:

Definition 9 (**obsioco**). *Given a set of input labels L_I and output labels L_O, **obsioco** $\subseteq \mathcal{IOLTS} \times \mathcal{IOLTS}$ is defined as follows:*

*Obs **obsioco** $m \overset{def}{\iff} \forall_\sigma\, [\sigma \in Straces(Obs) : out(Obs \text{ after } \sigma) \subseteq out(m \text{ after } \sigma)]$, where Obs are the observations and m is the learned model.*

Note that Obs, the observations, are merely a representation of the real implementation i; While the implementation is still assumed to be an $\mathcal{IOTS}$ (the *test assumption*), the observations may not know all input behaviour and therefore are a $\mathcal{IOLTS}$.

When observations are **obsioco** with the learned model m, the implementation may still be **ioco** with the learned model if $Straces(Obs) \subseteq Straces(m)$.

Chapter 4

Presenting an Algorithm for Test-Based Modelling of Non-Deterministic Systems

As we have seen in previous sections, current methods for model learning can not always be used in practical situations. Therefore a new algorithm is presented in the following section that provides a method of learning a model by testing, recognises patterns to reduce the model, guarantees the learned model is **obsioco** with the Teacher (and have the the observation traces as a subset of its own traces) and is easy to integrate with model-based testing techniques.

The learning itself is an *iterative* process: The Teacher is being observed; from these observations a model is generated and the Teacher is again being observed to check that model; if it is not correct, a new model is generated, etcetera.

4.1 Observing

In order to generate and verify a model that is learned by the Learner, it is necessary to observe the Teachers behaviour. To learn the behaviour, it is required to know L_I and L_O of the Teacher to be able to generate test-cases. These test-cases are traces of input labels. Observing a test-case sent to the Teacher results in a suspension trace.

It is required that the Teacher has a *Reset*-label to make sure that at least the state in which the Teacher starts, is the same. Note that $Reset \notin L$, so it cannot be used other than the first label of a test-trace; *Reset* will not be observed.

The set of all observed suspension traces is called $Straces(Obs)$. `observe`(i) is the set of suspension traces obtained by observing test-traces at iteration i.

The first test-trace is at iteration $i = 0$, where the test-traces are $\{Reset\}$. At `observe`$(i+1)$, $\{Reset \cdot \sigma_{inObs(i)} \cdot x\}$ are tested, where $x \in L_I$ and $\sigma_{inObs(i)}$ is a traces from `observe`(i) with only input labels in it. For its formal definition, projection is used[4]:

Definition 10 (projection)**.** *Let $\lambda \in L_\delta \cup \{\tau\}$ and $\Sigma \subseteq L_\delta \cup \{\tau\}$.*

$$\lambda\lceil\Sigma \begin{cases} \varepsilon & if \lambda \notin \Sigma \\ \lambda & otherwise \end{cases}$$

The definition of projection to traces is extended the following way, where $\sigma = \lambda_1 \cdots \lambda_n$ *for some* $n \geq 1$ *with* $\forall_i \, [1 \leq i \leq n : \lambda_i \in L_\delta \cup \{\tau\}]$:

$(\lambda_1 \cdots \lambda_n) \restriction \Sigma = (\lambda_1 \restriction \Sigma \cdots \lambda_n \restriction \Sigma)$

$\sigma_{inObs(i)}$ can now be defined as: $\sigma_{inObs(i)} = \sigma \restriction L_I$ where $\sigma \in \texttt{observe}(i)$.

4.1.1 Observing Quiescence

When sending an input label to an implementation, it may take a while before it responds. Therefore the Learner waits a moment before continuing. This response time of the Teacher must therefore be known and determines the waiting time. When, after waiting for this timeout, the Teacher did not give any output, quiescence (absence of output) is observed and $\tau \cdot \delta$ is added to the observed *Strace*. When quiescence is observed, it cannot directly be followed by another quiescence observation (observing $\tau \cdot \delta \cdot \tau \cdot \delta$ does not add relevant information).

A silent step, τ, is introduced to avoid any conflict with the definition of quiescence for systems that behave non-deterministically in giving output after a trace or not. When modelling the observations, the added silent step makes sure the state that is quiescent never shows output. Since τ itself cannot be observed, all τ-actions are followed by a δ label.

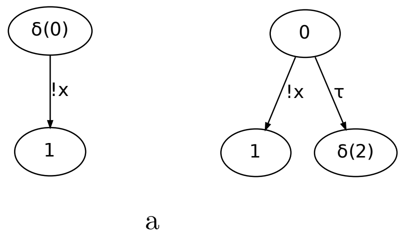

Figure 4.1: the left-hand picture (a) is in conflict with the definition of quiescence, so a silent step is introduced when quiescence is observed (b).

4.1.2 Building the Tree

Since *Straces*(*Obs*) is a set of suspension traces, each label in each Strace in *Straces*(*Obs*) gets its own transition. Except when two transition are coming from the same state and have the same label, it is assumed the state it goes to is also the same in order to make the model deterministic. For example, when *Straces*(*Obs*) $= \{a \cdot x, a \cdot y\}$, the tree will be modelled as in Figure 4.2.b. Since each test-case begins with a *Reset* it is safe to make the first label of a suspension trace in *Straces*(*Obs*) start from q_0.

Because of these properties and the observed traces being finite, the transition system generated has a tree structure with q_0 as the root.

Definition 11 (Obs)**.**

Obs is the deterministic $\mathcal{IOLTS}$ *with a tree structure generated from the observed traces,* *Straces*(*Obs*)*. Obs built from observe(i) is denoted as* Obs_i*.*

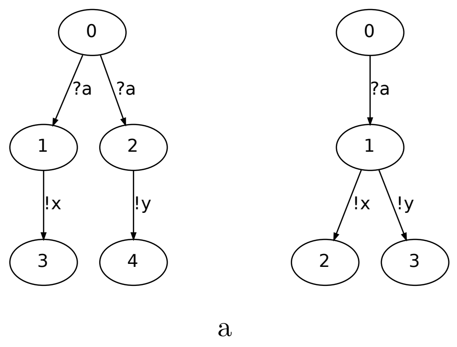

Figure 4.2: To make the model compact and deterministic, observation traces $\{a \cdot x, a \cdot y\}$ are modelled as the right hand picture (b) shows.

4.2 Guessing the Completer model

After each iteration i an $\mathcal{IOLTS}$ m is learned. This starts by generating *Obs* from *Straces*(*Obs*).

The states in *Obs* that do not have outgoing transitions, should still allow input labels at the Teachers side (since the Teacher is supposed to be input enabled by the definition of **ioco**). They just are not observed yet. A transition system with a tree structure will always have such 'leaf'-states and therefore cannot be the real representation of a system (the Teacher).

By trying to simulate branches on another, hypothetical loops can be introduced using the simulation relation:

Definition 12 (Simulation Relation)**.** *p can simulate q, where $p, q \in Q$ in a labelled transition system $\langle Q, L_I, L_O, T, q_0\rangle$, when there is a relation $R \subseteq state \times state$ such that:*

$$pRq \implies q \xrightarrow{a} q' \implies \exists_{p'} \left[p' \in Q : p \xrightarrow{a} p' \text{ and } p'Rq'\right]$$

Theorem 1. *There is a **simulation preorder**, $p \underline{\rightarrow} q$, where $p, q \in Q$ in a labelled transition system $\langle Q, L_I, L_O, T, q_0\rangle$, when there exist a simulation relation R such that pRq*

Proof. To show this relation is a preorder, it is necessary to show it is both reflexive ($\forall_q [q \in Q : qRq]$) and transitive ($\forall_{p,q,r} [p, q, r \in Q : pRq \wedge qRr \implies pRr]$). A state can always simulate itself, since they have the same traces, and therefore R is reflexive. When pRq and qRr, $p \xrightarrow{a} p' \implies \exists_{q'} \left[q \xrightarrow{q}{}' \implies \exists_{r'} \left[r \xrightarrow{a} r'\right]\right]$ implies $p \xrightarrow{a} p' \implies \exists_{r'} \left[r \xrightarrow{a} r'\right]$. The relation is therefore also transitive. $pRq \implies p \underline{\rightarrow} q$ □

It is easy to show $p \underline{\rightarrow} q$ is not always symmetrical (and therefore is not an equivalence relation) by looking at Figure 4.3, where $q_0 \underline{\rightarrow} q_2$ but not $q_2 \underline{\rightarrow} q_0$, because q_2 cannot do a `x`.

Whenever the loop-detection part of the algorithm detects a p and a q where $p \underline{\rightarrow} q$, it thinks q and p may be the same state, so it redirects all incoming transitions from p ($from(p)$) to q and removes all states and transitions that have become unreachable (which includes p). Because states and transitions are getting removed, the model gets smaller.

Leaf-nodes, defined as $\{q \in Q \mid from(q) = \emptyset\}$, do not get minimised by simulation directly: a leaf-node can always be simulated by all other states, which is considered to be too broad.

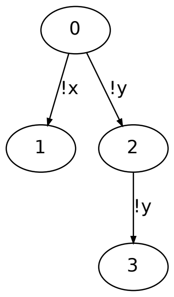

Figure 4.3: $q_0 \underline{\rightarrow} q_2$ but not $q_2 \underline{\rightarrow} q_0$

Other, stronger relations, such as a simulation equivalence relation would not be able to add information to incomplete observed branches, which is crucial since a system will never be completely observed (Section 3.4 "Learning a Precise Model").

Note that for the implementation of this simulation-relation τ, σ and labels $\in L$ are equally handled: If a state p can do either τ, σ and labels $\in L$, state q must also be able to do so. This, together with properties of observing quiescence (Section 4.1.1 "Observing Quiescence"), ensures to raise no conflict with the definition of quiescence: A state can not be quiescent and have output (or silent-steps) at the same time.

Because the simulation preorder is transitive, minimising a tree Obs in multiple steps, for example $m_0 \underline{\rightarrow} Obs, m_1 \underline{\rightarrow} m_0$ and $m_2 \underline{\rightarrow} m_1$, m_2 will still be able to simulate Obs. Therefore, it allows the Learner to do multiple loop-detection steps without losing the property $m \underline{\rightarrow} Obs$.

These steps are used to find a fixpoint, which means no state simulates another state. When there is a fixpoint, the loop-detection stops.

But where to start? Since the simulation preorder is not symmetrical per definition, the order in which these minimisation steps take place, becomes important to the fixpoint's model.

Because the fixpoint's model needs to be as compact as possible, the minimisation is done top-down. It starts from the top down to check if states can be simulated by states closer to the root. The advantages of starting to check for simulations at the top of the tree over bottom-up minimisations are:

- Minimisations are as close as possible to the root;
- A minimisation at the bottom may obstruct minimisations higher in the tree as shown in Figure 4.4;
- It is more likely traces for traces close to the root to have been covered more often and therefore it is more likely that all possible output is observed;
- Since there are more states at the bottom of the tree than at the top, more states are removed every step.

Figure 4.4 shows an example where top-down minimisation performs better than bottom-up minimisation: Because a larger pattern was recognised ($[a \cdot x \cdot x]^*$ instead of $[x]^*$), more states and transitions were removed resulting in a smaller model.

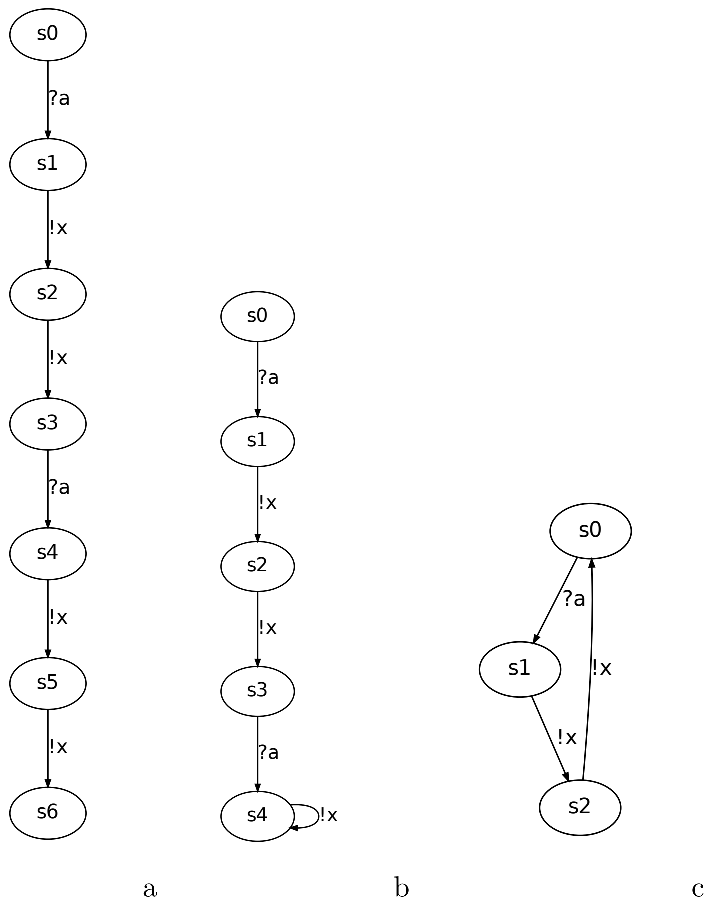

Figure 4.4: a. the original observation trace; b. bottom-up minimisation by $s4 \underline{\rightarrow} s5$ (fixpoint); c. top-down minimisation by $s0 \underline{\rightarrow} s3$ (fixpoint).

A minimisation step looks like the following pseudo-code, where the depth of a state is defined as the minimum number of labels needed to reach that state from q_0:

Ruby-style Pseudocode 1 (update_model_by_simulation).

```
def update_model_by_simulation(m)
  0..highest_depth.each do |n|
    $Q_L$ = m.get_states_at_depth(0..n)
    $Q_R$ = m.get_states_at_depth(n)
    $Q_R$.each do |$q_R$|
      $Q_L$.each do |$q_L$|
        next if $from(q_R) == \emptyset$
        next if $q_L == q_R$
        if $q_L$.can_simulate $q_R$
          $to(q_R)$.each do |cur_transition|
            cur_transition.to_state = $q_L$
          end
          clean_up_states_unreachable_from_q_0
          return
        end
      end
    end
  end
end
```

The implementation of `p.can_simulate q` looks like the following:

Ruby-style Pseudocode 2 (can_simulate).

```
def can_simulate
  $from(q)$.each do |$t_q$|
    $q'$ = $t_q$[to_state]
    return false if $\neg(p \xrightarrow{t_q[label]})$
    $p'$ = $p' \in Q \mid p \xrightarrow{t_q[label]} p'$
    return false unless $p'$.can_simulate $q'$
  end
  return true
end
```

When one state simulates another, the model changes, so the completing algorithm starts over again by trying to simulate the branches close to the initial state. When no more simulations are possible, the hypothesis is formed and ready to be tested.

4.3 Testing the Speculated Model with New Observations

After a model m is formed at an iteration i, `observe`$(i+1)$ is done and it is checked if these new observations can still be simulated by m. If not, a new model is made with the new *Obs* and another iteration is added to put the new model under test. When $Obs_{i+1} \underset{\rightarrow}{\rightarrow} m$, the learned model m is found plausible and the Learner will stop.

At the base step $i = 0$, m will always fail since $m = \emptyset$ and `observe`(0) is not empty (either δ or output is observed). The learned model will therefore contain only δ and/or output labels. So, if $L_I \neq \emptyset$, `observe`(1) will not be consistent with m.

Having explained how to observe, make a tree from the observations and how to add information to the specification model, the overview of the algorithm looks as follows, where m is the learned model:

Ruby-style Pseudocode 3 (learn)**.**

```
def learn
  $m$, $previous\_m = \emptyset$
  $i = 0$
  $Obs =$ observe(i)
  until $m \underline{\rightarrow} Obs$
    $m = Obs$
    until $m == previous\_m$
      $previous\_m = m$
      $m =$update_model_by_simulation(m)
    end
    $i = i + 1$
    $Straces(Obs) =$ observe(i)
  end
end
```

4.4 Proof of the ioco Correctness of the Learned Model

The previous chapter showed the characteristics of the learned $\mathcal{IOLTS}$ m and the finite observations Obs. In order to be able to use the learned model for model-based testing purposes, it needs to respect the **ioco** relation. Therefore, it is necessary to show that Obs **obsioco** m for all observations, where Obs is an $\mathcal{IOLTS}$ representation of the implementation and $m \underline{\to} Obs \implies Straces(Obs) \subseteq Straces(m)$.

Theorem 2. $m \underline{\to} Obs \implies Straces(Obs) \subseteq Straces(m)$

Proof. by induction on the length of the observation traces, denoted by $|\sigma|$

Induction Hypothesis $\mathcal{H}$: $\forall_\sigma \left[\sigma \in Straces(Obs) : Obs \xRightarrow{\sigma} q \implies \exists_p \left[p \in Q_m : m \xRightarrow{\sigma} p \wedge p \underline{\to} q\right]\right]$

Base Step: $|\sigma| = 0$

$\sigma = \varepsilon$

$Obs \xRightarrow{\varepsilon} q$

$\implies$ {$*$ by definition of $\xRightarrow{\varepsilon}$ $*$}

$Obs = p'$ or $Obs \xrightarrow{\tau\cdot\ldots\cdot\tau} p'$

$\implies$ {$*$ By the definition of simulation $*$}

When Obs can do a τ implies that m always can do a τ and reach a state that simulates the state in Obs. So $Obs = q \implies m = q'$ and $Obs \xrightarrow{\tau} q \implies m \xrightarrow{\tau} q'$ and $q' \underline{\to} q$.

(Note: a single silent step cannot exist when it is not followed by a quiescence-label (Section 4.1.1), so a $\tau \cdot \tau$ is not a possible part of a trace in either m or Obs).

Base Step holds.

Induction Step: $|\sigma| = n + 1$

where $x \in \tau \cup L_\delta : p \xRightarrow{\sigma\cdot x} p'$

$\implies$ {$*$ by definition of $\xRightarrow{\varepsilon}$ $*$}

$p \xRightarrow{\sigma} p_1 \xrightarrow{x} p_2 \xRightarrow{\varepsilon} p$

Following from the implementation of the simulation pre-order relation, where all labels $\in L$, τ and δ are simulated; if p_2 not equals p' on Obs (implying there is a τ-action), means m also has this τ-action:

$\forall_{\sigma,x} \left[\sigma \in Straces(Obs) x \in (L_\delta \cup \tau) : Obs \xRightarrow{\sigma\cdot x} q \implies \exists_{q'} \left[q' \in Q_m : m \xRightarrow{\sigma\cdot x} q' \wedge q' \underline{\to} q\right]\right]$

Induction Step holds.

□

Theorem 3. *Obs* ***obsioco*** *m, where Obs is the observation tree* $\langle Q(Obs), L_I, L_O, T(Obs), q_0(Obs)\rangle$ *and m the learned model* $\langle Q(m), L_I, L_O, T(m), q_0(m)\rangle$*, with* $L(Obs) = L(m)$.

Proof.

$\implies$ {$*$ by properties described in this chapter $*$}

$m \underline{\to} Obs$

$\implies$ {$*$ by Theorem 2 $*$}

$Straces(Obs) \subseteq Straces(m)$.

$\implies$ {$*$ by definition of $\subseteq$ $*$}

$\forall_\sigma [\sigma \in Straces(Obs) \cup Straces(m) : \sigma \in Straces(Obs) \implies \sigma \in Straces(m)]$

$$\begin{aligned}
\Longrightarrow\ & \{* \text{ because } \sigma \in \mathit{Straces}(m) \text{ always results in } \mathit{true} *\} \\
& \forall_\sigma \left[\sigma \in \mathit{Straces}(\mathit{Obs}) : \sigma \in \mathit{Straces}(\mathit{Obs}) \implies \sigma \in \mathit{Straces}(m)\right] \\
\Longrightarrow\ & \{* \text{ by definition of Straces } *\} \\
& \forall_\sigma \left[\sigma \in \mathit{Straces}(\mathit{Obs}) : \mathit{Obs} \overset{\sigma}{\Longrightarrow} \implies \exists_{q'} \left[q' \in Q(m) : m \overset{\sigma}{\Longrightarrow} q'\right]\right] \\
\Longrightarrow\ & \{* \text{ by definition of } \sigma *\} \\
& \forall_{\sigma,x} \left[\sigma \in \mathit{Straces}(\mathit{Obs}), x \in L_\delta \cup \{\tau\} : \mathit{Obs} \overset{\sigma\cdot x}{\Longrightarrow} \implies \exists_{q'} \left[q' \in Q(m) : m \overset{\sigma\cdot x}{\Longrightarrow} q'\right]\right] \\
\Longrightarrow\ & \{* \text{ by } L_O \subseteq L_\delta *\} \\
& \forall_{\sigma,x} \left[\sigma \in \mathit{Straces}(\mathit{Obs}), x \in L_O \cup \{\delta\} : \mathit{Obs} \overset{\sigma\cdot x}{\Longrightarrow} \implies \exists_{q'} \left[q' \in Q(m) : m \overset{\sigma\cdot x}{\Longrightarrow} q'\right]\right] \\
\Longrightarrow\ & \{* \text{ by definition of } \overset{\sigma\cdot x}{\Longrightarrow} *\} \\
& \forall_{\sigma,x} \left[\sigma \in \mathit{Straces}(\mathit{Obs}), x \in L_O \cup \{\delta\} : \mathit{Obs} \overset{\sigma}{\Longrightarrow} p \overset{x}{\Longrightarrow} \implies \exists_{q'} \left[q' \in Q(m) : m \overset{\sigma}{\Longrightarrow} q \overset{x}{\Longrightarrow} q'\right]\right] \\
\Longrightarrow\ & \{* \text{ because } m \underline{\rightarrow} \mathit{Obs} *\} \\
& \forall_{\sigma,x} \left[\sigma \in \mathit{Straces}(\mathit{Obs}), x \in L_O \cup \{\delta\} : \mathit{Obs} \overset{\sigma}{\Longrightarrow} p \xrightarrow{x} \implies \exists_{q'} \left[q' \in Q(m) : m \overset{\sigma}{\Longrightarrow} q \xrightarrow{x} q'\right]\right] \\
\Longrightarrow\ & \{* \text{ by definition of out and } \mathit{after} *\} \\
& \forall_{\sigma,x} \left[\sigma \in \mathit{Straces}(\mathit{Obs}), x \in L_O \cup \{\delta\} : x \in \mathit{out}(\mathit{Obs}\ \mathit{after}\ \sigma) \implies x \in \mathit{out}(m\ \mathit{after}\ \sigma)\right] \\
\Longrightarrow\ & \{* \text{ by definition of } \subseteq *\} \\
& \forall_\sigma \left[\sigma \in \mathit{Straces}(\mathit{Obs}) : \mathit{out}(\mathit{Obs}\ \mathit{after}\ \sigma) \subseteq \mathit{out}(m\ \mathit{after}\ \sigma)\right] \\
\Longrightarrow\ & \{* \text{ by definition of } \textbf{obsioco} *\} \\
& \mathit{Obs}\ \textbf{obsioco}\ m
\end{aligned}$$

□

Chapter 5

A Learning Example - The Algorithm In Action

Imagine a game with a coin where we can keep tossing a coin once, twice or not at all. The outcome of a toss with the coin is either Head or Tail (non-deterministically).

This game would be an excellent example to see the algorithm in action, since it has non-determinism and the possibility to have no output (quiescence), one output or two outputs after doing an input label.

Let $L_I = \{TossOnce, TossTwice, Pass\}$ and $L_O = \{Head, Tail\}$. Suppose we observe an implementation of such a game and we get the following observation traces at `observe(2)`:

$\tau \cdot \delta \cdot$`TossOnce`$\cdot$`Tail`$\cdot$`TossTwice`$\cdot$`Tail`$\cdot$`Head`
$\tau \cdot \delta \cdot$`TossOnce`$\cdot$`Head`$\cdot$`Pass`$\cdot \tau \cdot \delta$
$\tau \cdot \delta \cdot$`TossOnce`$\cdot$`Tail`$\cdot$`TossOnce`$\cdot$`Head`
$\tau \cdot \delta \cdot$`Pass`$\cdot \tau \cdot \delta \cdot$`TossTwice`$\cdot$`Tail`$\cdot$`Tail`
$\tau \cdot \delta \cdot$`Pass`$\cdot \tau \cdot \delta \cdot$`Pass`$\cdot \tau \cdot \delta$
$\tau \cdot \delta \cdot$`Pass`$\cdot \tau \cdot \delta \cdot$`TossOnce`$\cdot$`Tail`
$\tau \cdot \delta \cdot$`TossTwice`$\cdot$`Tail`$\cdot$`Tail`$\cdot$`TossTwice`$\cdot$`Head`$\cdot$`Head`
$\tau \cdot \delta \cdot$`TossTwice`$\cdot$`Head`$\cdot$`Tail`$\cdot$`Pass`$\cdot \tau \cdot \delta$
$\tau \cdot \delta \cdot$`TossTwice`$\cdot$`Tail`$\cdot$`Head`$\cdot$`TossOnce`$\cdot$`Tail`

When we visualise these $Straces(Obs)$, we get the Obs shown in Figure 5.1. The algorithm starts to check top-down for one state to be simulated by a state closer to the root. The first hit is state 0 simulating state 7, so all ingoing transitions from state 7 are routed to state 0 and states that are not reachable from state 0 are removed (state 7 and its branches).

Then again, by starting to check at the states that are the closest to state 0, state 1 simulates state 25, so again, the transitions to state 25 are routed to state 1 and the unreachable states are removed as shown in Figure 5.2.

This step is repeated until no more simulations are possible which results in Figure 5.6.

From the final model can be seen that it is a correct model: it is consistent with the description of the game; adding an observation iteration would therefore result in the same model.

If the implementation behaves as expected, adding any new observations to *Obs* would still result in *Obs* **obsioco** the learned model (Figure 5.6) and therefore, the implementation would be **ioco** with the learned model.

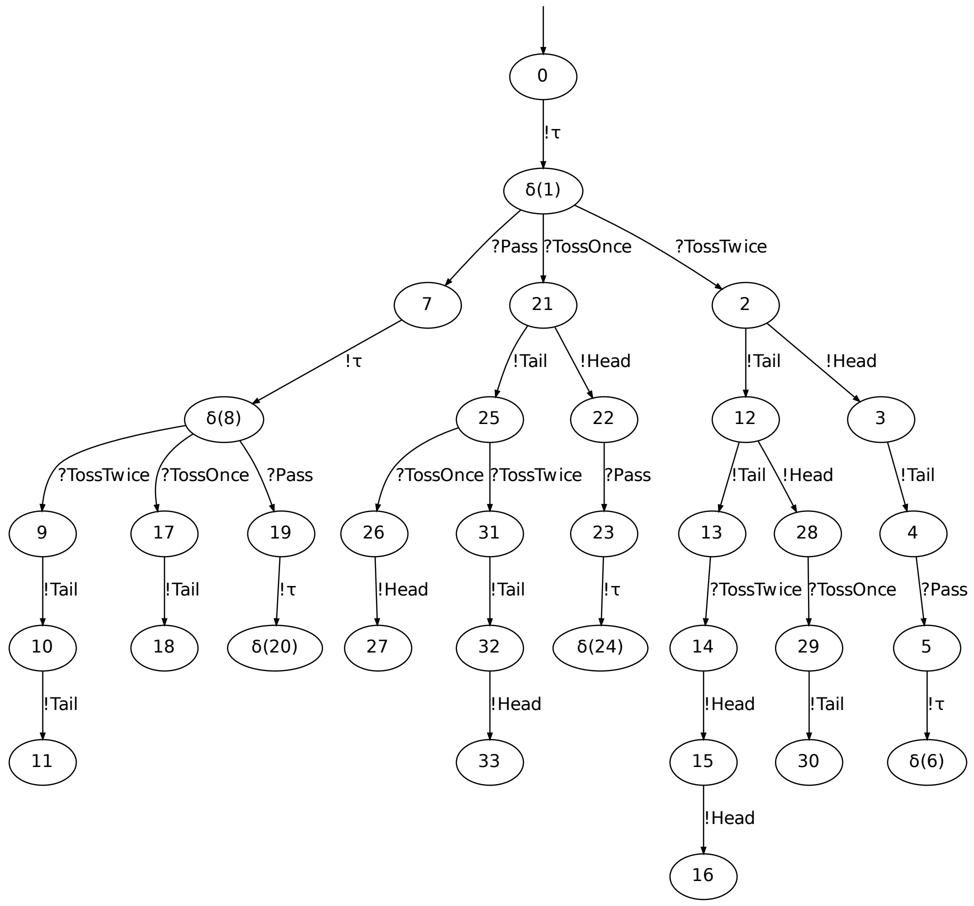

Figure 5.1: *Obs*, where state 0 simulates state 7, so state 7 will be removed

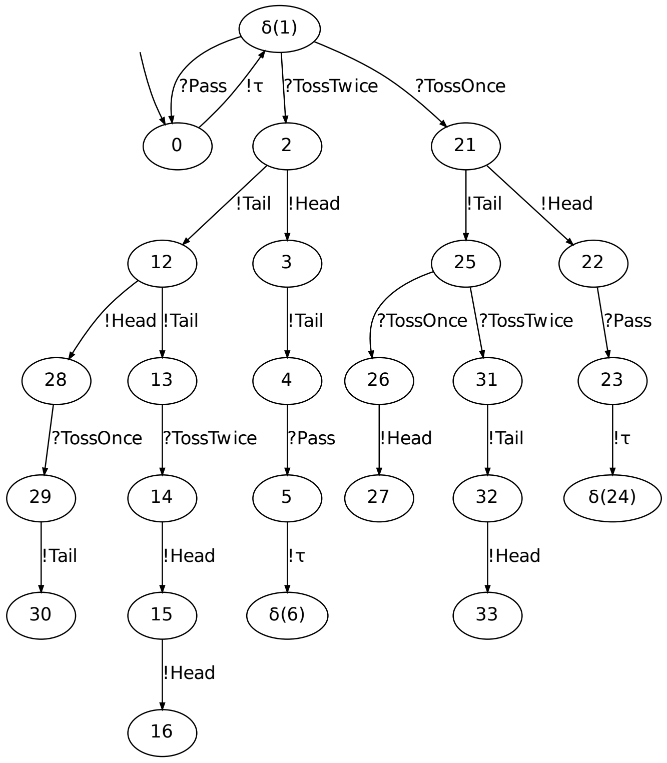

Figure 5.2: State 1 simulates State 25, State 25 will be removed

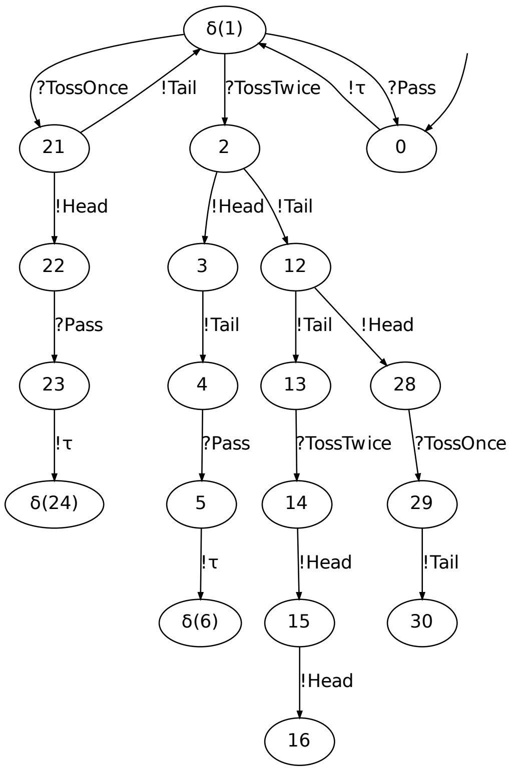

Figure 5.3: State 21 simulates State 3, State 3 will be removed

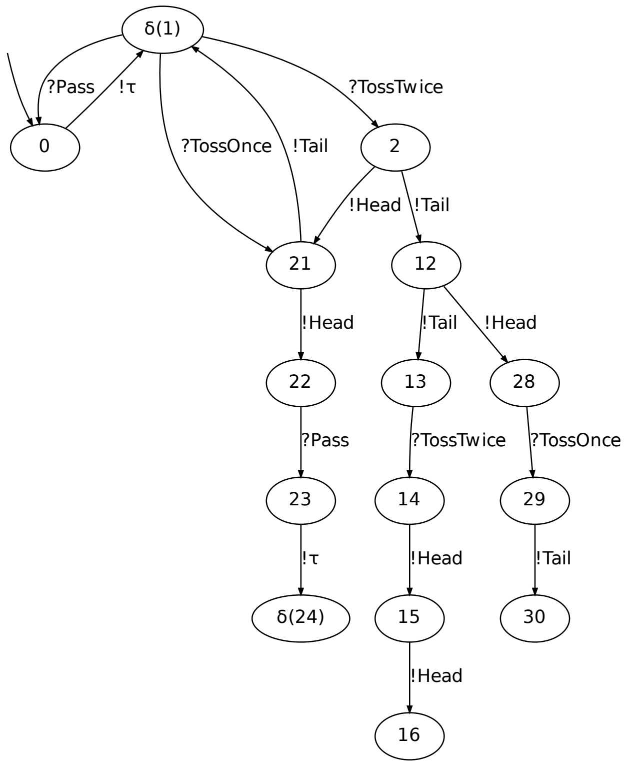

Figure 5.4: State 1 simulates State 22, State 22 will be removed

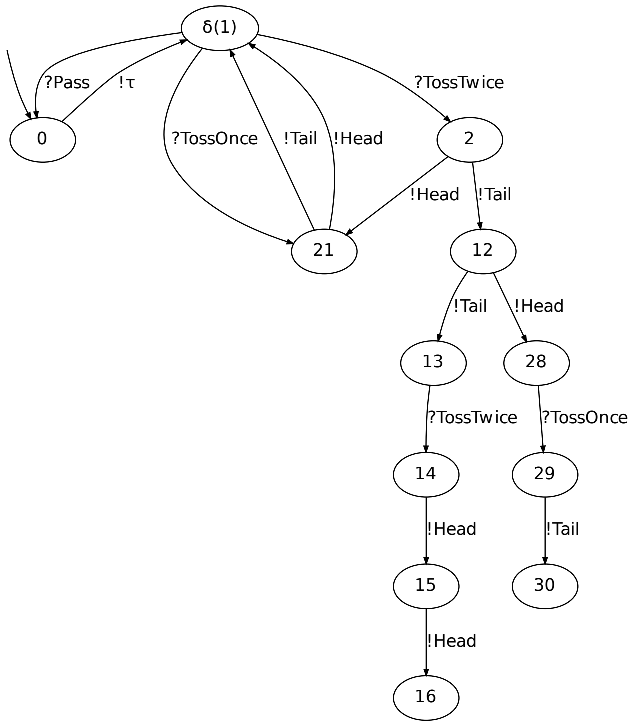

Figure 5.5: State 21 simulates State 12, State 12 will be removed

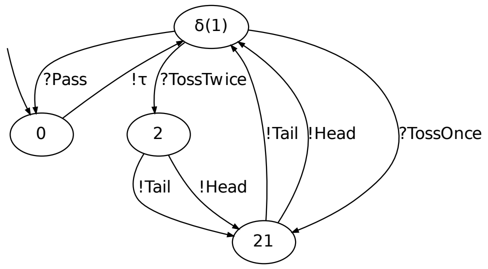

Figure 5.6: The final model, no more states can be simulated by another state

Chapter 6

Evaluation

In this research an algorithm was developed, which enables learning a model from their implementation to speed up the modelling process when starting with model-based testing. To demonstrate the algorithm, the tool 'Speculaas' was also developed. The models in the learning example from the previous chapter (Chapter 5 "A Learning Example - The Algorithm In Action") were generated by this tool. More information on Speculaas and how to use it is described in Appendix A and on `http://lennarttange.nl`.

From the simple tests that are done together with the theory from Chapter 4, it is possible to conclude the algorithm is able to:

- learn a model by knowing only the set of input labels, output labels, a reset label and the quiescence time-out;
- observe deterministic as well as non-deterministic behaviour of a system;
- model observations into a more compact model by adding information to it;
- test the added information;
- deliver a model that respects the properties of the **ioco** relation.

The observation table described by Angluin is a very powerful tool to learn the behaviour of a system by minimal effort. Because the Teachers in this research are always prefix-closed (Chapter 3) and Teachers cannot respond if a trace is not accepted, the observation table of Angluin was simplified to $Straces(Obs)$. Both methods strive after having complete tables.

Because a Teacher can only deal with traces of input labels and may be non-deterministic, a different way of observing was necessary. The way observations are done is comparable with the algorithm described by Willemse without his heuristics. Because the tree-structured models learned by Willemse would theoretically become of infinite depth, the new algorithm added loop detection based on a simulation pre-order relation. This loop detection allowed the Learner, as implemented in Speculaas, to find a correct model (including observing) for the coin game (Chapter 5) within three minutes.

6.1 Accepting a Model

The algorithm is able to come up with a transition system that represents all observations done so far. As explained in Section 3.4 "Learning a Precise Model", a person always needs to validate the model. It is always possible that not all behaviour was observed and therefore, the person may add known traces to $Straces(Obs)$ by hand in order to get a better model. This feature is implemented in Speculaas. Observations can also be added this way for 'passive learning', learning by observations from a log file when the implementation operates in its natural environment (with real users for example).

It is also possible to learn parts of a system by giving a subset of the input labels to the Learner.

When the behaviour specified by the model is found acceptable it might contain redundant traces: traces that are observed, but unnecessary to specify a correctly working implementation. In order to keep the model maintainable, the model might need to be stripped of these traces.

6.2 Weaknesses

The algorithm has some scenarios in which it cannot not perform efficiently.

6.2.1 Hindering Non-Determinism

Though it is not likely, it is possible for non-deterministic systems to have non-determinism detected at a higher depth, but not at the lower depth. For example, when the coin-exmple would observe:

```
TossTwice,Tail,Tail,TossTwice,Head,Head ...
TossTwice,Tail,Tail,TossTwice,Tail,Head ...
TossTwice,Tail,Tail,TossTwice,Head,Tail ...
TossTwice,Tail,Tail,TossTwice,Tail,Tail ...
```

The loop-detection would not recognise doing a second `TossTwice` comes from the same state as the first `TossTwice`. But these observations the loop to the state doing a `TossTwice` would be recognised by the loop-detection:

```
TossTwice,Head,Head,TossTwice,Tail,Tail
TossTwice,Tail,Head,TossTwice,Tail,Tail
TossTwice,Head,Tail,TossTwice,Tail,Tail
TossTwice,Tail,Tail,TossTwice,Tail,Tail
```

When the implementation of an unfair coin is being tested, the first case may be wanted. But when the coin is fair as in the learning example, it is unwanted. The problem here is that it uses another iteration when the model is falsified, which requires exponentially more time than the previous one.

6.2.2 Variable Quiescence

Imagine the testing of a bridge. It takes at maximum 60 minutes for the bridge to open or to close. That would imply, when learning the system, the quiescence time-out needs to be set to 60 minutes. This means the Learner waits after every input label 60 minutes. Suppose there are input action that lead to output different from opening or closing of the bridge (outputs that do not require 60 minutes), it could require a huge amount of time to learn the model, while the 60 minutes waiting time is not always necessary.

6.2.3 Data as a Label

Next case is a system that accepts data as a parameter and/or as output, for example a simple calculator. The set of input labels is almost infinite (every number) so observing all those labels takes a huge amount of time, while testing all numbers might not even be necessary.

6.3 Future Research

The cases described in the previous section are cases the other described methods also can't deal with efficiently or not at all. For that reason it may still be useful to research the new algorithm further.

While the performance of the algorithm is not measured, the time needed for observations expands exponentially. Observing is the most time-consuming part of the algorithm, so performance optimisations could be best done in the observing part. Take in consideration that it might be acceptable to have an automated learning method that takes a few days (or nights) since making a model by hand can easily take considerable more time. However, there are still some opportunities to make the algorithm perform better and faster.

Statistics can be used to conclude it is unlikely to have more output behaviour as observed, for example in the case given in Section 6.2.1 "Hindering Non-Determinism" or for learning the quiescence time-out in the case described in Section 6.2.2 "Variable Quiescence". Since all output labels are known, it would be possible to calculate the probability of an output label when and where to occur. Furthermore, when observing, all input labels are added to the previous observations. For complex systems it might be more efficient to test just a subset of those traces. The observed test-traces may be determined by the information added to the model: a smarter way for trying to verify/falsify the loops.

An easy way to decrease the observation model in size would be the removal of the unnecessary introduced τ-steps: τ-steps that do not prevent the conflict in definition between output and δ.

Also it might be useful to add more features to support passive learning of a system. The advantage of this passive learning is that the Learner can see what the most used traces are (which may be more important than other traces). This can be used to further minimise the transition system. In the example of the coffee machine (Chapter 2 "Preliminaries - Model-Based Testing"), the behaviour of normal usage (Figure 2.1) can be learned instead of all behaviour (as an $\mathcal{IOTS}$).

As for the case described in Section 6.2.3 "Data as a Label", it would be useful to add support for data in the transition systems, for example by using the Symbolic Transition System instead of $\mathcal{IOLTS}$, that are able to put conditions on the labels. Also, it might be necessary to use an abstraction layer for the data parameters that learns which abstractions can be made in the data so not all combinations of data have to be observed.

Bibliography

[1] Aarts, F., Schmaltz, J., and Vaandrager, F. Inference and abstraction of the biometric passport. In *ISoLA 2010, 4th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation* (2010), pp. 673–686.

[2] Aarts, F., and Vaandrager, F. Learning i/o automata. In *CONCUR 2010, 21th International Conference on Concurrency Theory* (2010), pp. 71–85.

[3] Angluin, D. Learning regular sets from queries and counterexamples. *Inf. Comput. 75* (November 1987), 87–106.

[4] Bijl, van der, H. M. A. *On changing models in model-based testing.* PhD thesis, Enschede, May 2011. IPA Dissertation Series No. 2011-07.

[5] Bollig, B., Habermehl, P., Kern, C., and Leucker, M. Angluin-style learning of nfa. In *Proceedings of the 21st international jont conference on Artifical intelligence* (2009), Morgan Kaufmann Publishers Inc., pp. 1004–1009.

[6] Raffelt, H., Steffen, B., and Berg, T. Learnlib: a library for automata learning and experimentation. In *FMICS 2005, 10th international workshop on Formal methods for industrial critical systems* (2005), pp. 62–71.

[7] Tretmans, J. Test generation with inputs, outputs and repetitive quiescence. Tech. rep., Univerity of Twente, 1996.

[8] Tretmans, J. Testing techniques. Tech. rep., Univerity of Twente, 2002.

[9] Tretmans, J. Formal methods and testing. Springer-Verlag, Berlin, Heidelberg, 2008, ch. Model based testing with labelled transition systems, pp. 1–38.

[10] Willemse, T. A. C. Heuristics for ioco-based test-based modelling. In *FMICS 2006, 11th international workshop and PDMC 2006, 5th international workshop on Formal methods: Applications and technology* (2006), Springer-Verlag, pp. 132–147.

Appendix A

Speculaas - The Tool Implementing The Algorithm

A lot of effort was put in creating a software tool to demonstrate how the algorithm works and to support future work.

The name of the tool, Speculaas, refers to a typical Dutch biscuit as well as the Latin word ‘speculum’ for mirror of which the English word ‘speculation’ is derived.

This chapter explains how Speculaas works, what it does and a few important design decisions.

A.1 Getting Started

Speculaas is a web application and can be found at `http://lennarttange.nl`. It is recommended to watch the demonstration video first.

At the homepage, some predefined configurations are shown. Two demonstration applications are running on the Speculaas server:

- **The Coin Teacher** - This application is the same application as described in Chapter 5 “A Learning Example - The Algorithm In Action” (the example described there is based on the output of Speculaas): A coin can be tossed once, twice or not at all. The outcome of tossing is either `Head` or `Tail` (non-deterministic);
- **The StartStop Teacher** - This is an even simpler application, where the Teacher responds to input `Start` with output `Started` and to `Stop` with `Stopped`.

Note that these applications can be communicated with by one user at a time, which means only one user can ‘learn’ at a time. To start learning, click the ‘Learn’ button next to a configuration. Learning can take some time because it needs to observe the behaviour of the Teacher. It should not take longer as 15 minutes for a predefined configuration to learn. When the learning is finished, the user is redirected to a new page showing the results.

A.2 Configuration

Each configuration has variables that can be shown by clicking ‘View Config’ next to a configuration on the home page.

A.2.1 General Information

General Information is a collection of variables that are supposed to give a user a description of Learner:

- **Application Name** - The name of the application; in the predefined configurations this is either `Coin` or `StartStop`;
- **Configuration Name** - A summarising title of how the Learner is configured;
- **Description** - A long description of how the Learner is configured;
- **User Name** - The name of the person configured the Learner;
- **Status** - Shows whether this configuration is running or not.

A.2.2 Maximum Iterations

Maximum Iteration is a natural number that limits the number of iterations the Learned is allowed to do. It is introduced to be able to control the maximum learning time and enables a very useful feature for demonstration purposes: Learning without observing the Teacher (which will be discussed in Section A.2.5 “Observations”).

A.2.3 Labels

Labels are input actions the Teacher accepts (L_I) and actions it can output (L_O). A separate Reset label is needed for reasons described in Section 4.1 “Observing”. When the given set of input labels is just a subset of all actions the Teacher accepts, the Learner will only learn the behaviour of the system regarding to that subset, which can be useful.

A.2.4 Quiescence Timeout

Quiescence Timeout is the timeout described in Section 4.1.1 “Observing Quiescence”.

A.2.5 Observations

The set of observations is a set of consecutive labels. In a configuration, observations can be changed or added manually. A configuration also saves whether the Learner should save new observations to this variable and whether the Learner should load the observations from this set at iteration 0. This allows the following scenarios:

- Normal learning: start with an empty set of observations and save the observations at the end of the learning process;

- Learning strictly by the given observations, without needing to connect to a Teacher (the 'maximum iteration' variable needs to be set to 0);
- Seeing what happens each iteration by setting 'maximum iterations' at 1 and let the Learner load and save the observations. Each time the 'Learn' button is pressed, another iteration is added.

A.2.6 Host Information

The Learner connects to the Teacher by a XML-RPC interface. Therefore, it needs the IP address and the port number of the Teacher. This enables the learning of a remote application. More information on the XML-RPC interface is described in Section A.4 "Teacher Interface".

A.3 Other Features

The output Speculaas gives is a image file of a transition system rendered by Graphviz. Also, Speculaas outputs the learned model as in Aldebaran format. This .aut-file can be used by other modelling tools such as mCRL2[1] and TorX[2]. The Learner also saves the steps to a compacter model (as described in Section 4.2 "Guessing the Completer model") of the last iteration. These can be seen at the result page.

All features to add, delete or edit configurations have been removed from the webpage to keep the demonstration clean and intact.

A.4 Teacher Interface

As mentioned before, the Learner connects to the Teacher by a XML-RPC interface. Because the Teacher may take a while to respond to an input label or not at all, the communication is set up the following: The Learner sends an input label and the Teacher responds with an acknowledgement; the Learner waits for the defined 'Quiescence Timeout' and then asks the Teacher is there is any Output: if there is any, this output is added to the observation trace, if not, a silent step and quiescence is added. When the Teacher does not respond with an acknowledgement it means it has either still an output ready or an error message that showed up (for example when an unknown label was given as input).

A.5 Runtime Environment

Speculaas is developed in Ruby (v1.8.7). To make it easily accessible and demonstrable, it has been put into the Ruby-on-Rails (v3.0.7) web application framework. The Speculaas server runs Ubuntu 10.04 64-bit, Apache 2, SQLite 3, Phusion[3] Passenger together with Phusion Ruby Enterprise (v1.8.7-2011.03).

[1] `http://www.mcrl2.org`
[2] `http://fmt.cs.utwente.nl/tools/torx/`
[3] `http://www.phusion.nl`

A.6 Further Development

Speculaas is released under the GNU GPL (v3)[4] licence. Source code is available upon the request from the author.

It should be easy to set up the development environment when familiar with Ruby. The application can also run by starting a local server by using the default command `rails server`. All needed gems (Ruby libraries) are bundled in a bundler-file. All core features of Speculaas are implemented in a single folder containing approximately 750 lines of documented source code.

[4]`http://www.gnu.org/licenses/gpl-3.0.txt`