Using Decision Tables in Model-Based Testing

Johanneke T. Lamberink

May 12, 2014

Preface

This paper is the result of a graduation project for the master Software Engineering at the University of Amsterdam. This research started out as a mixture of formal reasoning, empirical studies, and implementation. In the end, it turned out to be mostly formal reasoning, while the empirical studies were dropped completely.

I would like to thank Jan van Eijck[1] for his help with the formalisms encountered and the theory in general, but mostly for his guidance in work methodologies. He seems to have a vast amount of tips and advices on how to tackle problems and get to work. I would also like to thank Machiel van der Bijl[2] for giving me the opportunity to do my research at Axini, and for his advice and knowledge when I needed it.

Thanks to the professionals both at Axini and CWI for their help and advice. A big thank you to my close friends and my family who kept pushing me, and were available for motivation and advice. And lastly I am thankful for the opportunity I was given by my employer Richard Smit[3] to take time off and finish my graduation project.

[1]Centrum Wiskunde & Informatica, University of Amsterdam.
[2]Axini.
[3]Onior B.V.

Contents

Introduction

Fully automated testing of software systems can be done through the use of models, called model-based testing [7]. In this paper, we assume we're dealing with black-box testing, where only the outside of the system under test (SUT) is known, as opposed to white-box testing, where the internal structure of the SUT is also known by the tester. The constructed models need to be specified by means of a formal method if we want to be able to reason about them without ambiguity. The use of formal methods in software projects is increasing, but there are still people who do not see the advantages, or simply dont know how to use it [6].

Fortunately, thanks to companies such as Axini, people do not need knowledge of formal methods themselves to make use of automated testing. Axini is a company that specialises in the black-box testing of the functional behaviour of software systems, using symbolic transition systems. Symbolic transition systems are a formal method based on labelled transition systems, with the explicit notion of data and data-dependent control flow added. Both labelled and symbolic transition systems are further discussed in Section 1. The experience at Axini is that their clients have a hard time understanding the models that are produced for their real-life systems, such as pension administration, tax administration, and banking.

Because they want to help their clients understand their models, and let them make models themselves, Axini is interested in other formalisms that are able to express the same behaviour as their symbolic transition systems. It is their hope that decision tables might help them communicate with their clients. In the past, research has shown [4, 3, 5] that decision tables, further discussed in Section 2, are a formalism easily understood, regardless of one's background.

When I came across this topic as a possible thesis subject, I was immediately drawn to the idea that something as simple as a decision table might be used to model a software system. Although I had never done anything with model-based testing, the practice of formalizing specifications was not new to me. In combination with the feeling that research on decision tables would not be for a purely theoretical purpose, but could actually benefit Axini and their clients, my interest in the subject was raised enough to pick it for my master's thesis.

In this paper, I will first discuss labelled transition systems in Section 1 and give examples. In Section, 2, I define decision tables and explain their semantics. In Section 3 I give equivalence relations for labelled transition systems and decision tables. These relations are later used for proving the correctness of transformation algorithms I introduce in Section 3.2. The transformation algorithms are for transforming sets of decision tables into labelled transition systems, and the other way around. Section 4 shows an example implementation of the transformation algorithms in Haskell. My findings can be found in Section 5. Lastly, in Section 6 I discuss symbolic transition systems and suggestions for future research.

Chapter 1

Model-Based Testing

Software testing is aimed at finding differences between the actual and the intended behaviour of a system, which is done by experimenting with the system, observing the results, and forming a verdict. Writing test cases is a difficult process which takes up a lot of resources, and the resulting tests are bound to contain errors as well. This is where model-based testing comes in. It is a testing technique where the system under test (SUT) is modelled, and test cases are generated based on this model. To guarantee correct and meaningful test cases, the model itself should be validated, which implies that it has to be simpler than the actual SUT. Formal specifications and models make it possible to reason about the models in a precise and unambiguous manner. Another advantage is that formal models are more suitable for automated testing. A formal method often used for describing reactive systems are labelled transition systems. Reactive systems are event-driven systems in which stimulus/response behaviour is important [6].

1.1 Labelled Transition Systems

A labelled transition system (LTS) consists of states and transitions between the states. Transitions are labelled with actions that the system can perform.

Definition 1.1 (LTS [6]).
A labelled transition system is a tuple $\langle q_0, Q, L, \rightarrow\rangle$ where

- $q_0 \in Q$ is the initial state;
- Q is a countable, non-empty set of *states*;
- L is a countable set of observable *action labels.* The special action label τ represents unobservable actions, we write $L_\tau = L \cup \{\tau\}$;
- $\rightarrow \subseteq Q \times (L_\tau) \times Q$ is the *transition relation.* When $(q, \lambda, q') \in \rightarrow$ we write $q \xrightarrow{\lambda} q'$.

To make a distinction between input and output, we can divide the label set in input labels and output labels, $L = L_I \cup L_O$, with $L_I \cap L_O = \emptyset$. Every state in an LTS has either only transitions with input labels, or only transitions with output labels. We call them input states, respectively output states. A state without output labels is called *quiescent*, the system can not continue without input. Observing that a state is quiescent can be considered an event [7], which we will call δ ($\delta \notin L_\tau$). The δ-transition is left out of the visualisation of models for clarity purposes. Non-determinism is modeled by a state with multiple transitions with output labels.

Transitions can be composed, such that the transition $q \xrightarrow{\lambda_1 \cdot \lambda_2 \ldots \lambda_n} q'$ describes the sequence of actions $\lambda_1 \cdot \lambda_2 \ldots \lambda_n$ that can be executed to go from state q and end in state q'. A *trace* is the set of observable actions in a sequence. The term observable refers to the system's environment. Unobservable actions, such as internal actions, can not be observed by the sytem's environment. Observable actions on the other hand, model the system's interaction with its environment. When abstracting from unobservable actions in a sequence $q \xrightarrow{\lambda \cdot \tau \cdot \mu} q'$ we write $q \overset{\lambda \cdot \mu}{\Longrightarrow} q'$.

Decision tables, Section 2, have a notion of valuation, in the form of propositions that are either true or false. For the transformation from one model to the other in Section 3, LTSs need a valuation as well. This is done by assigning a valuation to every state.

Definition 1.2 (State)**.**
A state is a tuple $q = \langle i, \theta \rangle$ where

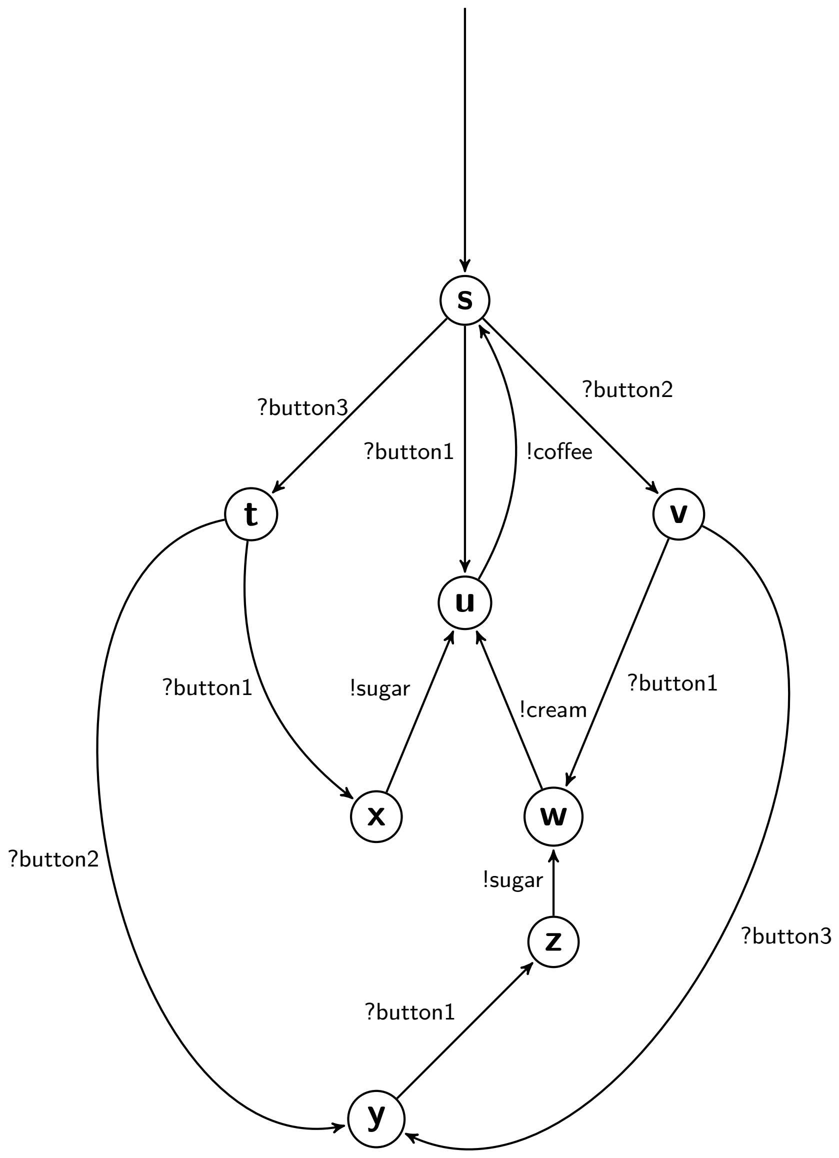
- i is the index of the state;
- θ is the valuation of state variables. We write $\text{val}(q) = \theta$.

A typical way of representing an LTS is by a tree or graph. Nodes and (labelled) edges represent states and transitions respectively, see Diagram 1.1. The set of action labels is $L = \{$?button1, ?button2, ?button3, !coffee, !sugar, !cream$\}$ and the set of states $Q = \{\langle s, \emptyset \rangle, \langle t, \emptyset \rangle, \langle u, \emptyset \rangle, \langle v, \emptyset \rangle, \langle w, \emptyset \rangle, \langle x, \emptyset \rangle, \langle y, \emptyset \rangle, \langle z, \emptyset \rangle\}$. What this LTS shows is the behaviour of a coffee machine with three buttons. When button1 is pressed, it gives coffee. When first button2 and then button1 is pressed, it gives coffee with cream. When first button3 and then button1 is pressed, it gives coffee with sugar. When first button2 and button3 are pressed, in either order, and then button1 is pressed, is gives coffee with cream and sugar.

An LTS lacks the possibility to model variables and data values. For data values to be in the model, they have to be modelled explicitly in the action labels as names. This representation of the data no longer contains the structure and information of the original data definitions and constraints [2]. It also leads to the state space explosion problem [1]. Frantzen [1] added the concepts of location variables and data, the latter of which can be communicated over gates, to labelled transition systems, calling the result symbolic transition systems (STSs). By treating the data symbolically, STSs can be combined with a multitude of formalisms for the specificiation and manipulation of data. In STSs, data types are treated as sets of values and first-order formulas specify values or predicates [1]. STSs are further discussed in Section 6.

This section introduced the concepts of both labelled and symbolic transition systems. Labelled transition systems are often used for black-box testing of reactive systems, because of their ability to model functional specifications. Symbolic transition systems are an extension of labelled transition systems, to solve the state space explosion problem with the introduction of data and data control flow. In the next section I will explain what decision tables are and give definitions and examples.

Diagram 1.1 Coffee Machine



Chapter 2

Decision Tables

Decision rules are a part of everyday life. When saying something along the lines "*if it rains, I will take the bus, otherwise I'll bike*", you have defined a decision rule, where the *action* to be taken (riding the bus or a bike) is dependent on a *condition* (whether it rains or not). There are different methods for describing decision rules, such as flowcharts [3], but I will only discuss decision tables here.

Decision tables are, as the name suggests, a tabular way of describing decision rules. A typical decision table is divided into four quadrants, which are usually separated by double lines in the visual representation of the table, see Table 2.1. The upper left quadrant is called the *condition stub*, this is where the conditions are listed. The outcomes of the conditions are defined in the upper right quadrant, the *condition entry*. Entries on one row define the different outcomes of the condition that is on that same row in the stub. The lower part of the table is divided similarly. The lower left quadrant lists the actions, and is called the *action stub*. The lower right quadrant, the *action entry*, defines which actions are to be executed if the corresponding column of condition entries holds.

2.1 Definitions

Most literature about decision tables doesn't give a formal definition of the tables, but instead focuses on a description of the components and their semantics. While such an informal definition in a natural language is sufficient for using decision tables, it is not enough for formal reasoning. The latter requires a formal, precise definition. I started out with a simple definition that could be extended if need be.

Definition 2.1 (Decision Table).
Let P be a set of *propositions*, A a set of *actions.*
A decision table over P and A is a tuple $t = \langle i, \to_i \rangle$ where

- i is the *index* of the table;
- $\to_i$ is a set of *decision rules* $(\Phi, \alpha^\star)$ where $\Phi \subseteq P$, $\alpha^\star$ a sequence of $\alpha \in A$.

To mimic transitions in transition systems, I introduce a special action where a decision table can reference another decision table by index. A decision rule then is extended with a reference to another table, $(\Phi, \alpha^\star, k)$ with k an index of a decision table. When $(\Phi, \alpha^\star, k) \in \to_i$ we write $i \xrightarrow{\Phi,\alpha^\star} k$. For every $(\Phi, \alpha^\star, k) \in \to_i$ it holds that if all propositions $\phi \in \Phi$ hold, then the actions in $\alpha^\star$ are executed in order, and we proceed with the table with index k. In the tabular visualisation of a decision table, these extra reference actions are separated from the other actions by a double line.

Condition Stub	Condition Entry
Action Stub	Action Entry

Table 2.1: Decision Table

1: $\emptyset$				
?button1	T	T	T	T
?button2	F	T	F	T
?button3	F	F	T	T
coffee	X	X	X	X
cream		X		X
sugar			X	X
goto 1	X	X	X	X

Table 2.2: Coffee Machine

Since every decision rule in a decision table references another table, we risk getting endless loops. To prevent this from happening, we use the symbol $\surd$ to denote a special decision table without decision rules, so with no conditions, no actions, and no references. This table can be used as the end of a sequence of decisions.

Tables with references to each other form a set which I call a *decision structure*. A simple example decision structure can be found in Table 2.2. The structure contains only one table, with index 1. The set of actions is {coffee, cream, sugar}, the set of propositions is {?button1, ?button2, ?button3}, and the decision rules are described by the columns of the table. For execution purposes, we also need an initial set of propositions that hold, which can be found next to the index of the start table, in this case the empty set, $\emptyset$. Although this structure might seem to model the same machine as the LTS from Diagram 1.1, it does not. In this structure, there is no difference in first pressing button1 and then button2 (or button3, or both), or the other way around. The LTS clearly modelled the effects of first pressing button2 and/or button3, *followed* by pressing button1. We will see later how to model that LTS as a decision structure.

Definition 2.2 (Decision Structure)**.**
Let I be a set of *indices.*
A decision structure over I is a tuple $\langle i, \iota, T\rangle$ where

- i is the starting index with $i \in I$;
- ι is the initial set of conditions that hold, with $\iota \subseteq P$;
- T is the set of decision tables, with for all $t = \langle i_t, \rightarrow_t\rangle \in T$ and for all $i_t \xrightarrow{\Phi, a^\star}_t k$, then $k \in I$;
- $\forall j \in I \;.\; \exists\langle j, \rightarrow\rangle \in T$;

This section explained the concept of decision rules, and how they can be expressed in decision tables. It also showed that decision tables can be used to express the black-box functionality of a system. Because of their tabular form, decision tables are comprehensible for non-technicians as well as technicians. In the next section decision tables are compared with transition systems, to explore whether both formalisms can express the same behaviour.

Chapter 3

Formal Comparison of decision structures and transition systems

Now that we have formal definitions, how do we compare decision tables and transition systems? If we are able to transform one model into the other, and then inverse the transformation, the resulting model should be equivalent to the original. Thus, we need a transformation algorithm, an inverse transformation algorithm, and an equivalence relation.

3.1 Bisimilarity

For labelled transition systems there is an equivalence relation, named the bisimulation relation, or bisimilarity. The idea is that if two states are bisimilar, then it is not possible to distinguish them based on their behaviour. This means that two states s and t are bisimilar if for every state s' reachable from s by executing action λ, there is a state t' reachable from t by executing the same action λ such that s' and t' are bisimilar.

Definition 3.1 (Bisimilarity)**.**
Given a labelled transition system $\langle s_0, Q, L, \rightarrow\rangle$, a bisimulation is a binary relation $R \subseteq Q \times Q$ such that $\forall (s,t) \in R \,.\, \forall \lambda \in L \cup \{\tau\}$:

$$\forall s' \;.\; \text{ if } (s \xrightarrow{\lambda} s') \text{ then } \exists t' \,.\, (t \xrightarrow{\lambda} t' \wedge (s', t') \in R) \tag{Zig}$$

$$\forall t' \;.\; \text{ if } (t \xrightarrow{\lambda} t') \text{ then } \exists s' \,.\, (s \xrightarrow{\lambda} s' \wedge (s', t') \in R) \tag{Zag}$$

$$\begin{tikzcd}
s \arrow[r, dash] \arrow[d, "\lambda"'] & t \arrow[d, dashed, "\lambda"] \\
s' \arrow[r, dash, dashed] & t'
\end{tikzcd}$$

Zig

$$\begin{tikzcd}
s \arrow[r, dash] \arrow[d, dashed, "\lambda"'] & t \arrow[d, "\lambda"] \\
s' \arrow[r, dash, dashed] & t'
\end{tikzcd}$$

Zag

We can extend this definition of bisimulation to work with the definition of states (Definition 1.2) that we use for our LTSs, which includes a valuation. In addition to the rules in Definition 3.1, the bisimulation should also uphold the following invariant:

$$\forall (s,t) \in R \,.\, val(s) = val(t) \tag{Invariance}$$

If there is a bisimulation relation R such that $(s,t) \in R$, we write $s \sim t$. To prove that bisimilarity is an equivalence relation, we need to show that it is reflexive, symmetric and transitive. For this purpose, we consider bisimilarity within a single LTS.

Reflexive Assume $s \sim s$, then there is a bisimulation $R : s \sim s$. The identity relation is a bisimulation, $I : s \sim s$. Thus $s \sim s$.

Symmetric Assume $s \sim t$, then there is a bisimulation $R : s \sim t$. Note that (Invariance), (Zig), and (Zag) are symmetric. Thus, $R : s \sim t$ implies $R^{-1} : t \sim s$. Thus $t \sim s$.

Transitive Assume $s \sim t$ and $t \sim u$. Then there exist bisimulations $R : s \sim t$ and $R' : t \sim u$. It follows that $R \circ R' : s \sim u$. Thus $s \sim u$.

Given two labelled transition systems $s_1 = \langle s_{0_1}, Q_1, L, \rightarrow_1 \rangle$ and $s_2 = \langle s_{0_2}, Q_2, L, \rightarrow_2 \rangle$, we say that $s_1 \sim s_2$ if there is a bisimulation R between s_1 and s_2 such that $(s_{0_1}, s_{0_2}) \in R$.

3.1.1 Bisimilarity for decision structures

With bisimilarity defined for LTSs, let's try to define it for decision structures as well. For LTSs the bisimulation relation is focused on the transition relation over states, so the set of decision rules of decision tables seems a good starting point for bisimilarity on decision structures. Two tables s and t are bisimilar if for every table s' that is reachable from s by executing action a under propositions Φ, there is a table t' reachable from t by executing the same action a and under the same set of propositions Φ, such that s' and t' are bisimilar.

Definition 3.2 (Bisimilarity of decision tables)**.**
Given a decision structure $\langle i, \iota, T \rangle$, a bisimulation is a binary relation $R \subseteq T \times T$ such that $\forall (s,t) \in R \,.\, \forall \Phi \subseteq P \,.\, \forall a \in A$:

$$\forall s' \,.\; \text{if } (s \xrightarrow{\Phi, a^\star}_s s') \text{ then } \exists t' \,.\, (t \xrightarrow{\Phi, a^\star}_t t' \wedge (s', t') \in R) \tag{Zig'}$$

$$\forall t' \,.\; \text{if } (t \xrightarrow{\Phi, a^\star}_t t') \text{ then } \exists s' \,.\, (s \xrightarrow{\Phi, a^\star}_s s' \wedge (s', t') \in R) \tag{Zag'}$$

If there is a bisimulation R such that $(s,t) \in R$, we write $s \sim t$. Again, to prove that it is an equivalence relation, we need to show that bisimulation on decision structures is reflexive, symmetric and transitive. This is very similar to what was done for bisimulation on LTSs. We consider bisimilarity within a single decision structure.

Reflexive Assume $s \sim s$, then there is a bisimulation $R : s \sim s$. The identity relation is a bisimulation, $I : s \sim s$. Thus $s \sim s$.

Symmetric Assume $s \sim t$, then there is a bisimulation $R : s \sim t$. Note that (Zig') and (Zag') are symmetric. Thus, $R : s \sim t$ implies $R^{-1} : t \sim s$. Thus $t \sim s$.

Transitive Assume $s \sim t$ and $t \sim u$, then there are bisimulations $R : s \sim t$ and $R' : t \sim u$. It follows that $R \circ R' : s \sim u$. Thus $s \sim u$.

Given two decision structures $s_1 = \langle i_1, \iota_1, T_1 \rangle$ and $s_2 = \langle i_2, \iota_2, T_2 \rangle$, we say that $s_1 \sim s_2$ if there is a bisimulation R between s_1 and s_2 such that $(t_1, t_2) \in R$ with $t_1 \in T_1$ the decision table with index i_1 and $t_2 \in T_2$ the decision table with index i_2.

3.2 Transforming an LTS to a decision structure and vice versa

Recall the definition for LTSs, Definition 1.1 on page 1, and the definition for decision structures, Definition 2.2 on page 5. In this section I will introduce an algorithm for transforming LTSs into decision structures, the inverse algorithm, and proof of their correctness.

3.2.1 LTS to decision structure

Given an LTS $s = \langle q_0, Q, L, \rightarrow_s \rangle$ we have a transition relation $\rightarrow_s$. An element $q \xrightarrow{\lambda}_s q'$ describes how to go from state q to state q' by executing action λ. Transitions with input labels together with the state valuations provide the conditions of tables, while output labels provide us with the actions. See Algorithm 3.1. Applying the transformation algorithm to an LTS s is denoted with a $\circ$, such that s° is the resulting decision structure.

Algorithm 3.1 Transforming an LTS (pseudo-code)

```
Require: An LTS $\langle\langle i,\theta\rangle, Q, L, \rightarrow_s\rangle$.
 1: new set $C$
 2: new set $M$
 3: new set $D$
 4: $M = M \cup \{\langle i,\theta\rangle\}$
 5: $C = C \cup \{\langle i,\theta\rangle\}$
 6: while $C \neq \emptyset$ do {Breadth-first traversal}
 7:    select $s \in C$
 8:    new set $I_s = \{(a,t)|(a,s',t) \in\rightarrow_s, s == s', a \in L_I\}$
 9:    new set $O_s = \{(a,t)|(a,s',t) \in\rightarrow_s, s == s', a \in L_O \cup \{\tau\}\}$
10:    new set $E_s = I_s \cup O_s$
11:    new set $E'_s = \{t|(a,t,s') \in\rightarrow_s, s == s', t \notin M\}$
12:    new set $N = \{t|(a,t) \in E_s, t \notin M\} \cup E'_s$
13:    $M = M \cup N$
14:    $C = (C - \{s\}) \cup N$
15:    $D = D \cup \{(s, O_s, I_s)\}$
16: end while
17: new set $T$
18: for all $(\langle i,\theta\rangle, O_s, I_s) \in D$ do
19:    if $O_s = \emptyset$ then {quiescent state}
20:       new set $\rightarrow_{O_s} = \{\theta, \{\delta\}, i)\}$
21:    else
22:       new set $\rightarrow_{O_s} = \{(\theta, \{a\}, j)|(a, \langle j,\rho\rangle) \in O_s\}$
23:    end if
24:    new set $\rightarrow_{I_s} = \{(\theta \cup \{\pi\}, \emptyset, j)|(\pi, \langle j,\rho\rangle) \in I_s\}$
25:    $T = T \cup \{\langle i, \rightarrow_{O_s} \cup \rightarrow_{I_s}\rangle\}$
26: end for
27: return $\langle i, \theta, T\rangle$
```

The algorithm starts by performing a breadth-first traversal on the LTS, to collect the in- and outgoing transitions per state (line 6). Because a state may not be reachable from the start state, the traversal also visits states from which a current state can be reached. The result of the breadth-first traversal is a set of tuples of the transitions per state, divided in transitions with in- and output labels. The algorithm then proceeds to loop over the tuples, turning every tuple into a decision table (line 18). If the state has no transitions with output labels (thus is quiescent), a decision rule with the special δ-action is added (line 19). The set of transitions with output labels become decision rules with the valuation of the 'from'-state as conditions, the output label as only action in the action sequence and the 'to'-state index as table reference. The transitions with an input label become decision rules where the label is added to the conditions, and the action sequence is empty. The set of decision rules that is the result of one iteration of the loop form the decision rules of one decision table. After all tuples have been processed, the algorithm returns a decision structure with the created decision tables, and the valuation of the start state as initial set of conditions that hold.

For an example of the algorithm, we take the example LTS 1.1 on page 3. The breadth-first traversal starts with state s as the current state. After visiting s the set of tuples D now has one tuple with transitions, $D = \{\langle s, \emptyset\rangle, \{(\delta, s)\}, \{(?\text{button1}, u), (?\text{button2}, v), (?\text{button3}, t)\}\}$. The traversal continues for the other states, after which the set of tuples is used for creating decision tables. The tuple for state s results in a table with 4 columns, one for each of the transitions in the tuple. The result is a set of decision tables, see the tables in Table 3.1.

3.2.2 Decision structure to LTS

Given a decision table $t = \langle i, \rightarrow_t\rangle$ over A, P, and I, we have a set of decision rules $\rightarrow_t$. An element $(\Phi, a^\star, k) \in \rightarrow_t$ describes that when the set of propositions Φ holds, the system can execute the actions in $a^\star$ in order and go to the table with index k. We can annotate conditions in the table to indicate which conditions should become input labels, and which conditions should become state valuations, Φ_{label} and Φ_{val} respectively. Since we are dealing with LTSs in this transformation, we assume all actions are output labels and do not allow for proposition updates. See Algorithm 3.2 for a pseudo-code algorithm for the transformation. Applying the transformation algorithm to a decision table t is denoted with a $\bullet$, such that $t^\bullet$ is the resulting LTS.

The algorithm starts by setting up some variables for the fields of an LTS. After this initialisation it goes into a loop over all tables in the decision structure, on line 5, in order to collect all valuations in the tables. The loop adds a state to Q for every valuation in every table. The next step in the algorithm is to visit all tables (line 10) and for every table visit all decision rules (line 12). The in- and output labels encountered in a rule are added to the sets L_I and L_O respectively, and the algorithm creates a new state r with the valuation conditions of the rule as state valuation and the current table's index as state index. The loop that collects valuations will already have added an instance of this state to Q. In case the decision rule has no labels at all (line 17) the algorithm adds τ-transitions to all states in Q with the same index as the one referenced by the rule.

Because the conditions in a decision table do not have an order, the input labels in the decision rule can be put in the LTS in different orders. The output labels should occur after every possible order of the input labels, thus the algorithm keeps a set R of states from which the output transitions should start. If there are no input labels, this set consists only of the state r (line 19). Otherwise, the algorithm loops over all permutations of the input labels (line 22) and visits the labels in every permutation in order of the permutation (line 23). If the decision rule contains no output labels, and the current input label is the last occurring label in the current permutation, the algorithm adds transitions with the current label to all states in Q with the same index as the one referenced by the rule (line 24). Otherwise the algorithm adds a transition with the current label from state r to the newly created state r' with the same valuation as r and a unique index. The new state is also added to the set of states Q, and r is updated to be r', thus creating a sequence of transitions. After all labels in a permutation are processed, the last created state is added to the set R, to ensure that the sequence of input transitions is followed by a sequence of output labels.

s				
?button1	F	T	F	F
?button2	F	F	T	F
?button3	F	F	F	T
δ	X			
goto s	X			
goto t		X		
goto u			X	
goto y				X

(a)

t	
(no conditions)	
coffee	X
goto s	X

(b)

u			
?button1	F	T	F
?button2	F	F	T
delta	X		
goto u	X		
goto v		X	
goto w			X

(c)

v	
(no conditions)	
cream	X
goto t	X

(d)

w		
?button1	F	X
delta	X	
goto w	X	
goto x		X

(e)

x	
(no conditions)	
sugar	X
goto v	X

(f)

y			
?button1	F	T	F
?button2	F	F	T
delta	X		
goto y	X		
goto z		X	
goto w			X

(g)

z	
(no conditions)	
sugar	X
goto t	X

(h)

Table 3.1: Coffee Machine

Algorithm 3.2 Transforming a decision structure (pseudo-code)

```
Require: A decision structure $ds = \langle i, \iota, T\rangle$ over $A$, $P$, and $I$.
 1: new set of states $Q$
 2: new set of input labels $L_I$
 3: new set of output labels $L_O$
 4: new relation $\rightarrow_s$
 5: for all $\langle q_n, \rightarrow_n\rangle \in T$ do {collect valuations}
 6:   for all $(\Phi, a^\star, k) \in \rightarrow_n$ do
 7:     $Q = Q \cup \{\langle q_n, \Phi_{\text{val}}\rangle\}$
 8:   end for
 9: end for
10: for all $\langle q_n, \rightarrow_n\rangle \in T$ do
11:   new integer $j = 0$
12:   for all $(\Phi, a^\star, k) \in \rightarrow_n$ do
13:     $L_O = L_O \cup \{a \in a^\star\}$
14:     $L_I = L_I \cup \Phi_{\text{label}}$
15:     new state $r = \langle q_n, \Phi_{\text{val}}\rangle$
16:     new set $R = \emptyset$
17:     if $\Phi_{\text{label}} \cup a^\star = \emptyset$ then {no labels}
18:       $\rightarrow_s = \rightarrow_s \cup \{(r, \tau, \langle l, \theta\rangle) | (l, \theta) \in Q \wedge l == k\}$
19:     else if $\Phi_{\text{label}} = \emptyset$ then {no input labels}
20:       $R = \{r\}$
21:     else
22:       for all permutations $\Pi$ of $\Phi_{\text{label}}$ do
23:         for all $\pi \in \Pi$ do {visit labels in order}
24:           if $\pi$ is the last proposition in $\Pi$ and $a^\star = \emptyset$ then {no output labels}
25:             $\rightarrow_s = \rightarrow_s \cup \{(r, \pi, \langle l, \theta\rangle) | (l, \theta) \in Q \wedge l == k\}$
26:           else
27:             new state $r' = \langle q_{n_j}, \Phi_{\text{val}}\rangle$
28:             $Q = Q \cup \{r'\}$
29:             $\rightarrow_s = \rightarrow_s \cup \{(r, \pi, r')\}$
30:             $j = j + 1$
31:             $r = r'$
32:           end if
33:         end for
34:         $R = R \cup \{r\}$
35:         $r = \langle q_n, \Phi_{\text{val}}\rangle$
36:       end for
37:     end if
38:     for all $a \in a^\star$ do {visit actions in order}
39:       if $a$ is the last action in $a^\star$ then
40:         for all $r \in R$ do
41:           $\rightarrow_s = \rightarrow_s \cup \{(r, a, \langle l, \theta\rangle) | (l, \theta) \in Q \wedge l == k\}$
42:         end for
43:       else
44:         new state $r' = \langle q_{n_j}, \Phi_{\text{val}}\rangle$
45:         $j = j + 1$
46:         $Q = Q \cup \{r'\}$
47:         for all $r \in R$ do
48:           $\rightarrow_s = \rightarrow_s \cup \{(r, a, \langle l, \theta\rangle) | (l, \theta) \in Q \wedge l == k\}$
49:         end for
50:         $R = \{r'\}$
51:       end if
52:     end for
53:   end for
54: end for
55: return $\langle\langle i, \iota\rangle, Q, L_I \cup L_O, \rightarrow_s\rangle$
```

When all permutations have been visited the algorithm proceeds with the actions in the decision rule (line 38), which are visited in order (remember that a decision rule contains a *sequence* of actions). If the algorithm encounters the last action in the sequence it adds transitions with that action as output label from each state in R to all states in Q with the same index as the one referenced by the rule (line 39). If the action is not the last in the sequence, the procedure is similar as for the input labels. The algorithm adds transitions with the current output label from the states in R to a newly created state r' with the same valuation as r and a unique index (line 43). The new state is added to the set of states Q, and afterwards the set R is replaced by the singleton set containing r'. This way transition sequences with input label permutations converge to one sequence with output labels. After all tables have been processed the algorithm returns an LTS with the created set of states Q, transitions $\rightarrow_s$, labels $L_O \cup L_I$, and initial state $\langle i, \iota \rangle$.

For an example execution of the algorithm, we look at the decision structure from the previous section, that was the result of the LTS to decision structure algorithm, Algorithm 3.1. All conditions occurring in the decision structure should be used for input labels, so the loop that collects valuations results in a set of states with empty valuations, one state for every decision table. After that, for every rule in every table the label conditions are added to the set of input labels, and the actions are added to the set of output labels. This results in the set $L_I = \{$?button1, ?button2, ?button3$\}$ and $L_O = \{$coffee, cream, sugar$\}$. The LTS that is returned by the algorithm is shown in Diagram 3.1. We can already see that this LTS is very similar to the original LTS from page 3. Using bisimulation we can check that the LTSs are equivalent. Where the new LTS has δ-transitions, we need to remember from Section 1.1 that an input state always has a δ-transition to itself, but that those transitions were left out of the diagram.

3.2.3 Proof of correctness

So how do we know the algorithms correctly transform decision structures and LTSs? A reasonable demand is that we should be able to reverse the transformation, and end up with a model equivalent to the original. We need to prove that after performing a transformation on LTS S to decision structure S°, and from S° to an LTS $S^{\circ\bullet}$, the LTSs S and $S^{\circ\bullet}$ are equivalent. For this purpose, we have defined the bisimulation relation in Section 3.1. Our proof of bisimulation between S and $S^{\circ\bullet}$ has to consist of four subproofs, one for the (Invariance), one for (Zig), one for (Zag), and one for proving the set of labels for S and $S^{\circ\bullet}$ are equal. For the decision structure we write $S^\circ = \langle s_0^\circ, \theta^\circ, T \rangle$. For the LTS after transformation we write $S^{\circ\bullet} = \langle \langle s_0, \theta \rangle, Q, L, \rightarrow \rangle^{\circ\bullet} = \langle \langle s_0^{\circ\bullet}, \theta^{\circ\bullet} \rangle, Q^{\circ\bullet}, L^{\circ\bullet}, \rightarrow^{\circ\bullet} \rangle$.

Theorem 3.1 (Equal label sets)**.**
For any given LTS $S = \langle \langle s_0, \theta \rangle, Q, L, \rightarrow \rangle$ and it's corresponding $S^{\circ\bullet}$, the label sets of both LTSs are equal, that is, $L = L^{\circ\bullet}$.

Proof. Given LTSs $S = \langle \langle s_0, \theta \rangle, Q, L, \rightarrow \rangle$ and $S^{\circ\bullet}$.

$$\begin{aligned}
&\forall \lambda \in L && \text{then } \exists ((\theta, \lambda, j) \text{ in } S^\circ \vee (\theta \cup \{\lambda\}, \emptyset, j) \text{ in } S^\circ) \\
&\forall (\Phi \cup \{\lambda\}, a^\star, j) \text{ in } S^\circ && \text{then } \exists (s, \lambda, t) \in \rightarrow^{\circ\bullet} \\
&\forall (\Phi, a^\star \cup \{\lambda\}, j) \text{ in } S^\circ && \text{then } \exists (s, \lambda, t) \in \rightarrow^{\circ\bullet}
\end{aligned}$$

Thus, $L \subseteq L^{\circ\bullet}$.

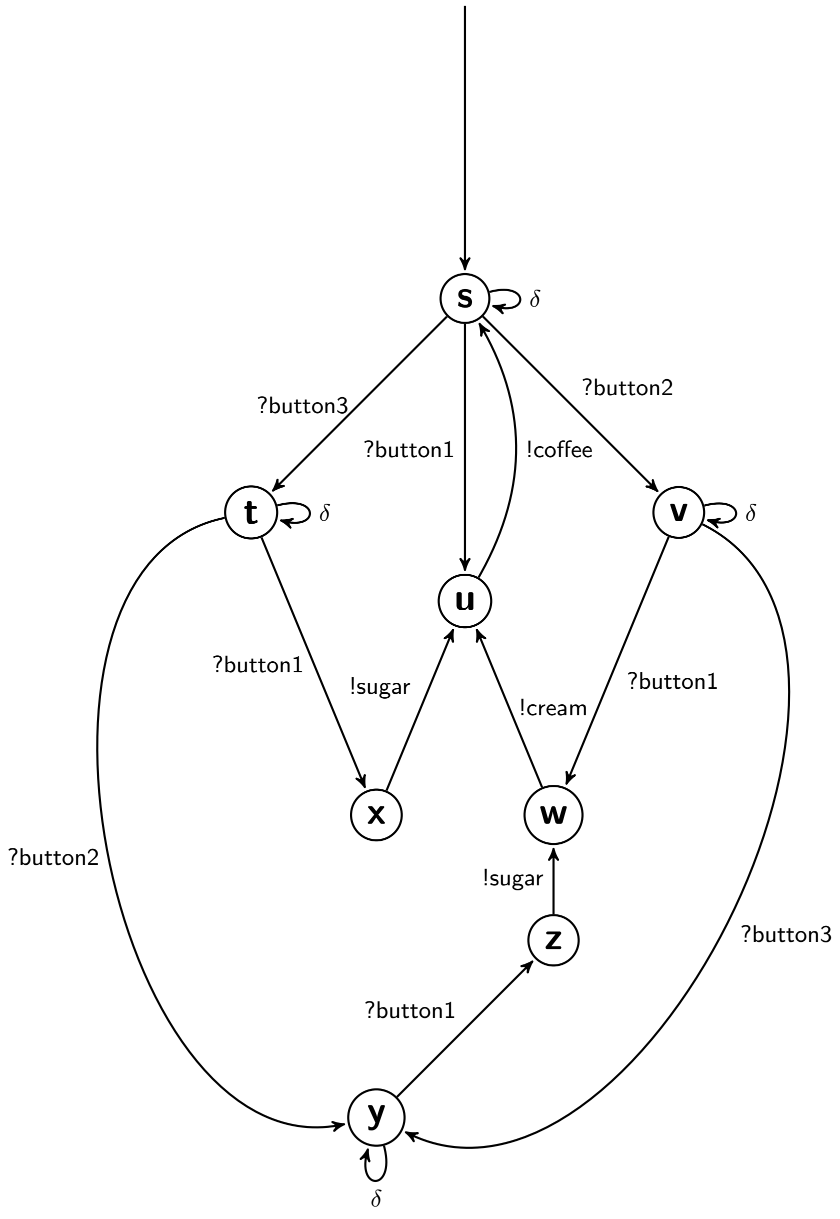
$$\begin{aligned}
\forall \lambda \in L^{\circ\bullet} &\text{ then } \exists ((\Phi, a^\star \cup \{\lambda\}, j) \in S^\circ \vee (\Phi \cup \{\lambda\}, a^\star, j) \text{ in } S^\circ) \\
\forall (\Phi \cup \{\lambda\}, a^\star, j) \text{ in } S^\circ &\text{ then } \exists (s, \lambda, t) \in \rightarrow \\
\forall (\Phi, a^\star \cup \{\lambda\}, j) \text{ in } S^\circ &\text{ then } \exists (s, \lambda, t) \in \rightarrow
\end{aligned}$$

Thus, $L^{\circ\bullet} \subseteq L$. And, $L = L^{\circ\bullet}$. □

Now that we know the label set stays the same during the transformation, let's move on to proving that the valuation of a state is equal to the valuation of its counterpart after the transformation.

Theorem 3.2 (Invariance)**.**
For any given LTS $S = \langle \langle s_0, \theta \rangle, Q, L, \rightarrow \rangle$ and it's corresponding $S^{\circ\bullet}$, there exists a relation R such that $\forall (s, s^{\circ\bullet}) \in R \,.\, \mathrm{val}(s) = \mathrm{val}(s^{\circ\bullet})$.

Diagram 3.1 Coffee Machine LTS from DS



Proof. Given LTSs $S = \langle\langle s_0, \theta\rangle, Q, L, \rightarrow\rangle$ and $S^{\circ\bullet}$.
According to the definition of lts2ds, a state $\langle i, \theta\rangle \in Q$ transforms to a decision tree $\langle i, \rightarrow_i\rangle$ and $\forall(\Phi, a^\star, k) \in\rightarrow_i \ . \ \Phi_{\text{val}} = \theta$
The definition of ds2lts states that for every rule $(\Phi, a^\star, k)$ in the set of decision rules of a decision table with index i, there exists a state $\langle i, \Phi_{\text{val}}\rangle$ in the output LTS. Thus, $\forall\langle i, \theta\rangle \in Q \ . \ \exists\langle i, \theta\rangle \in Q^{\circ\bullet}$. □

Next, we'll proof (Zig).

Theorem 3.3 (Zig)**.**
For any given LTS $S = \langle\langle s_0, \theta\rangle, Q, L, \rightarrow\rangle$ and it's corresponding $S^{\circ\bullet}$, there exists a relation R such that $\forall(s, s^{\circ\bullet}) \in R \ . \ \forall\lambda \in L$

$$\forall t \text{ if } s \xrightarrow{\lambda} t \text{ then } \exists t^{\circ\bullet} \ . \ (s^{\circ\bullet} \xrightarrow{\lambda}^{\circ\bullet} t^{\circ\bullet} \wedge (t, t^{\circ\bullet}) \in R)$$

Proof. Given LTSs $S = \langle\langle s_0, \theta\rangle, Q, L, \rightarrow\rangle$ and $S^{\circ\bullet}$.
Case of $\lambda \in L_I$:
The definition of lts2ds states that a transition $\langle i, \theta\rangle \xrightarrow{\lambda} \langle j, \rho\rangle$ translates to a decision rule $(\theta \cup \{\lambda\}, \emptyset, j)$ in the decision table with index i.
The definition of ds2lts states that a decision rule with exactly one input label and no output labels, $(\theta \cup \{\lambda\}, \emptyset, j)$, translates to a set of transitions with label λ, one for each found valuation for j. From Theorem 3.2 it follows that there exists a state $\langle j, \rho\rangle \in Q^{\circ\bullet}$, thus there will be a transition $\langle i, \theta\rangle \xrightarrow{\lambda}^{\circ\bullet} \langle j, \rho\rangle$.
Case of $\lambda \in L_O$:
The definition of lts2ds states that a transition $\langle i, \theta\rangle \xrightarrow{\lambda} \langle j, \rho\rangle$ translates to a decision rule (θ, λ, j) in the decision table with index i.
The definition of ds2lts states that a decision rule with exactly one output label and no input labels, (θ, λ, j), translates to a set of transitions with label λ, one for each found valuation for j. From Theorem 3.2 it follows that there exists a state $\langle j, \rho\rangle \in Q^{\circ\bullet}$, thus there will be a transition $\langle i, \theta, \rangle \xrightarrow{\lambda}^{\circ\bullet} \langle j, \rho\rangle$.
Thus, $\forall s \xrightarrow{\lambda} t \ . \ \exists s^{\circ\bullet} \xrightarrow{\lambda}^{\circ\bullet} t^{\circ\bullet}$. □

Lastly, let's prove (Zag).

Theorem 3.4 (Zag)**.**
For any given LTS $S = \langle\langle s_0, \theta\rangle, Q, L, \rightarrow\rangle$ and it's corresponding $S^{\circ\bullet}$, there exists a relation R such that $\forall(s, s^{\circ\bullet}) \in R \ . \ \forall\lambda \in L$

$$\forall t^{\circ\bullet} \text{ if } s^{\circ\bullet} \xrightarrow{\lambda}^{\circ\bullet} t^{\circ\bullet} \text{ then } \exists t \ . \ (s \xrightarrow{\lambda} t \wedge (t, t^{\circ\bullet}) \in R)$$

Proof. Given LTSs $S = \langle\langle s_0, \theta\rangle, Q, L, \rightarrow\rangle$ and $S^{\circ\bullet}$.
Case of $\lambda \in L_I$:
From the definition of ds2lts it follows that a transition of the form $\langle i, \theta\rangle \xrightarrow{\lambda}^{\circ\bullet} \langle j, \rho\rangle$ is the result of the transformation of a decision rule $(\theta \cup \{\lambda\} \cup \Phi, a^\star, k)$ in the decision table with index i, where Φ is a set of label conditions. According to the definition of lts2ds, in a decision structure that is the result of the transformation of an LTS, a decision rule can have at most one label condition. Thus, the decision rule in S° with an input label could only have the form $(\theta \cup \{\lambda\}, \emptyset, k)$.
The definition of lts2ds states that a decision rule of the form $(\theta \cup \{\lambda\}, \emptyset, k)$ in the decision table with index i comes from a transition $\langle i, \theta\rangle \xrightarrow{\lambda} \langle k, \rho\rangle$.
Case of $\lambda \in L_O$:
From the definition of lts2ds it follows that in a decision structure that is the result of the transformation of an LTS, decision rules can have at most one action, combined with only valuation conditions. Knowing this, the definition of ds2lts tells us that a transition of the form $\langle i, \theta\rangle \xrightarrow{\lambda}^{\circ\bullet} \langle j, \rho\rangle$ is the result of the transformation of a decision rule of the form $(\theta, \{\lambda\}, j)$.
According to the definition of lts2ds, such a decision rule is the result of a transition of the form $\langle i, \theta\rangle \xrightarrow{\lambda} \langle j, \rho\rangle$.
Thus, $\forall s^{\circ\bullet} \xrightarrow{\lambda}^{\circ\bullet} t^{\circ\bullet} \ . \ \exists s \xrightarrow{\lambda} t$. □

Chapter 4

Implementation in Haskell

The Haskell programming language allows for the creation of custom data types that are very similar to formal definitions. Thus, I was able to implement the definitions of decision tables and transition systems in Haskell in a straightforward way.

4.1 Data types

Recall the definition for LTSs on page 1, decision tables on page 5, and decision structures on page 6. In Listing 4.1 the Haskell data types for the different formalisms are displayed. Compared to the definition of an LTS, the `LTS` data type is missing a field for the set of labels, but they can be propagated by traversing the LTS. The other elements of an LTS are all part of the data type as well. There is the initial state, the non-empty set of states, and the transition relation. The distinction between input and output labels is made by the `Action` data type.

The data types for decision tables and decision structures are structured similar to the definitions for both. The `DT` data type for decision tables has two elements, an index and set of decision rules. The decision structure data type `DS` has three elements, a starting index, set of conditions, and a set of tables. For the transformation from decision structure to LTS we need to know which conditions should be input labels and which should be valuation conditions. The `Property` data type makes that distinction. In the transformation from decision structure to LTS it is also necessary to create multiple unique indices from one table, for which we need the `Label` data type.

The coffee machine example LTS, Diagram 1.1, can be implemented as in Listing 4.2. Here we first define the different labels to be used, before we start defining the LTS itself. For practical reasons I've used integers for the indices, but for clarity the names of the states as used in the diagram are next to the states in a comment.

4.2 LTS transformation

Listing 4.3 shows the Haskell implementation of the transformation algorithm from LTS to decision structure. Recall the algorithm on page 9. Similar to the algorithm, the implementation first collects a set of tuples in a breadth-first traversal of the LTS. The breadth-first traversal is implemented in the functions `bfs` and `bfs'`, where `bfs` is the set up for the actual traversing done in `bfs'`. Like in the algorithm, the result of the breadth-first traversal is a set of tuples with the input and output transitions per state. The arguments of the `bfs'` function are the transition relation, the set of states yet to be visited (C), the set of states already discovered (M), and a set of tuples which is expanded on every iteration. Eventually, this set of tuples is the result of the traversal. The traversal uses the function `while3`, which is a Haskell implementation of a while loop with three variables. It takes a function from the three variables' types to a `Bool`, a function from the three variables' types to a tuple of the three variables' types, and the three variables. When the first function argument (condition) evaluates to true, the second function argument is applied to the other three arguments and the result is used for

Listing 4.1 Data types

```
-- example properties/actions
data Prop = P1 | P2 | P3 | P4 deriving (Eq,Ord,Enum,Bounded,Show)
data Act = Delta | Tau | A1 | A2 | A3 | A4 deriving (Eq,Ord,Enum,Bounded,Show)

-- an LTS is a tuple (q0, Q, L, ->) where we can propagate the set L from ->
data LTS state prop act =
    LTS
    state
    [(state,[prop])]
    [(act,state,state)] deriving (Eq,Ord,Show)

-- distinguishing input labels from output labels
data Action = In Prop | Out Act deriving (Eq,Ord,Show)

-- a DT is a tuple (i, ->)
data DT index prop act =
  DT
  index
  [([prop],[act],index)] deriving (Eq,Show)

-- a DS is a tuple (i, I, T)
data DS index prop act =
  DS
  index
  [prop]
  [DT index prop act] deriving (Eq,Show)

-- distinguishing properties for input labels and properties for valuation conditions
data Property = Label Prop | Cond Prop deriving (Eq,Ord,Show)

-- used for creating unique indices while transforming
data Label = P Prop | A Act deriving (Eq,Ord,Show)
type Index = Int
```

Listing 4.2 Labelled Transition System - Coffee Machine

```
-- the labels of the LTS
button1, button2, button3, coffee , cream, sugar :: Action
button1 = In P1
button2 = In P2
button3 = In P3
coffee  = Out A1
cream = Out A2
sugar = Out A3

coffeeMachineLTS :: LTS Index Prop Action
coffeeMachineLTS = LTS 1
                        -- the set of states
                         [(1,[]) , -- s
                        (2,[]) ,   -- u
                        (3,[]) ,   -- v
                        (4,[]) ,   -- w
                        (5,[]) ,   -- y
                        (6,[]) ,   -- z
                        (7,[]) ,   -- t
                        (8,[]) ]   -- x
                        -- the transition relation
                        [(button1,1,2)
                        ,( coffee ,2,1)
                        ,(button2,1,3)
                        ,(button1,3,4)
                        ,(cream,4,2)
                        ,(button3,3,5)
                        ,(button1,5,6)
                        ,(sugar,6,4)
                        ,(button3,1,7)
                        ,(button1,7,8)
                        ,(sugar,8,2)
                        ,(button2,7,5)]
```

a new iteration. When the first function argument evaluates to false, the last argument (the third of the three variables) is returned as result of the function. In the breadth-first traversal, the first function argument tests whether there are still states to visit. The second function argument selects a state to visit, collects the input and output transitions, adds newly discovered nodes to the set of states to visit and marks them as discovered, and adds the collected transitions in a tuple to the result set. The third argument is the set of states to visit, and the fourth argument the set of discovered/marked states. The fifth argument is a set of tuples with input and output transitions per state, initially empty and used to append new tuples during iterations.

After the breadth-first traversal the algorithm goes into a loop over the result set of the traversal. The tuples resulting from the traversal have the input and output transitions sorted by state, but no information on the state valuation. So before the implementation continues with the loop over the tuples, it first adds the corresponding state. Next, the loop over the result set is done with a `map` of `tuples2table`, which creates a `DT` for every tuple. In `tuple2rules` the state valuation of the tuple is turned into conditions for the decision rules. The input labels are added as conditions in rules without action, while the output labels are added as actions to rules. When applying the implementation on the LTS implementation from Listing 4.2, we get the decision structure in Listing 4.4.

4.3 Decision Structure transformation

The transformation algorithm from a decision structure to an LTS is implemented as shown in Listing 4.5. First of all, for every decision rule the valuation conditions are collected and tupled with the table index, in `collectValuations`. After that, `ds2lts'` takes the set of valuations, the set of decision tables, and an empty list, and returns the transition relation. In order to loop over all tables and collect transition tuples for every table, `ds2lts'` uses the function `while2`, which is a Haskell implementation of a while loop with two variables. It works similar to `while3` from the previous section, but with two variables instead of three, and the function arguments also take two instead of three arguments. In this implementation, the first function argument tests whether there are still tables to transform. The second function argument selects a table, extracts the transition tuples from it, and adds those to the result set. The third argument is the set of tables, and the fourth argument the result set of transition tuples, initially empty and used to append new tuples to. The loop over the decision rules from a table is implemented in `dt2tuples`. Function `makeTuples` distinguishes the cases where a decision rule has no labels at all, has no input labels, and does have input labels. It's arguments are, in order, a set of input labels, a set of output labels, the referenced index, and the set of valuations that was collected.

If there are input labels the function `makeTuples'` is called for every permutation of the input labels. For every input label in a permutation a new state is made. To ensure that the state is unique states have an ordered list of labels to which the input label is added. After the input labels in the permutation have been processed, or if there were no input labels to begin with, the resulting set of tuples is appended with tuples for the action labels. The action labels are processed in order in `actionTuples`. Again, to ensure state uniqueness, the label is added to the ordered list of labels of the state to create a new state. When all input and all but one output labels are used to create transition tuples, or when the decision rule had no labels to begin with, the `endOfTrace` function is called to make transitions from the end of the created trace to all states with the referenced index. The states of the resulting LTS are of the type `(a,[Label]),[Prop])` where the list of `Labels` is only to ensure uniqueness and has no semantic value. To make the resulting LTS a bit more readable, there is a function `cleanLTS` that transforms the index of states from any type to an `Index`. When applying the implementation on the DS from Listing 4.4 we get the LTS in Listing 4.6.

Listing 4.3 Transformation algorithm LTS to DS

```haskell
-- implementation of LTS to decision structure algorithm
lts2ds :: Eq a => LTS a Prop Action -> DS a Property Act
lts2ds lts@(LTS i vals _) = let
    tuples = bfs lts
    sorted = [ (v, filter (\(j,_,_) -> j==i) tuples) | v@(i,_) <- vals ]
    tables = map tuples2table sorted
    maybeVal = lookup i vals
    init = case maybeVal of
             Nothing -> []
             Just ps -> map Cond ps
  in DS i init tables

-- Breadth-first traversal of an LTS
-- resulting in a set of tuples with the input and output transitions per state
bfs :: Eq a =>
       LTS a b Action ->
       [(a,[(Act,a)],[(Prop,a)])]
bfs (LTS s0 _ rel) = bfs' rel [s0] [s0] []

-- function to do the actual traversing
bfs' :: Eq a =>
        [(Action,a,a)] ->
        [a] ->
        [a] ->
        [(a,[(Act,a)],[(Prop,a)])] ->
        [(a,[(Act,a)],[(Prop,a)])]
bfs' rel = while3
            (\current _ _ -> not (null current))
            (\current marked result -> let
                (y,rest) = (head current, tail current)
                output = [ e | e@(Out _,u,_) <- rel, u == y ]
                input = [ e | e@(In _,u,_) <- rel, u == y ]
                newnodes = [ z | (_,_,z) <- output ++ input, notElem z marked ]
                newnodes' = [ z | (_,z,u) <- rel, u == y, notElem z marked ]
                new = nub $ newnodes ++ newnodes'
                current' = rest ++ new
                marked' = marked ++ new
                result' = (y, [ (a,z) | (Out a,_,z) <- output ]
                            , [ (a,z) | (In a,_,z) <- input ]) : result
            in (current',marked',result'))

-- Haskell implementation of a while loop with three variables
while3 :: (a -> b -> c -> Bool) ->
          (a -> b -> c -> (a,b,c)) ->
          a -> b -> c -> c
while3 p f x y z | p x y z   = let (x',y',z') = f x y z
                               in while3 p f x' y' z'
                 | otherwise = z

-- implementation of the loop over the tuples set from the breadth-first traversal
tuples2table :: (Eq a) =>
                ((a,[Prop]),[(a,[(Act,a)],[(Prop,a)])]) ->
                DT a Property Act
tuples2table ((i,v),xs) = let
    rules = nub . concat $ map (tuple2rules v) xs
  in DT i rules

-- function to turn a tuple into a set of decision rules
tuple2rules :: Eq a =>
               [Prop] ->
               (a,[(Act,a)],[(Prop,a)]) ->
               [([Property],[Act],a)]
tuple2rules v (i,acts,props) = let
    conditions = map Cond v
    actions | acts == [] = [ (conditions, [Delta], i) ]
            | otherwise  = [ (conditions, [a], k) | (a,k) <- acts ]
    labels = [ (Label p : conditions, [], k) | (p,k) <- props ]
  in actions ++ labels
```

Listing 4.4 DS from LTS - Coffee Machine

```
DS 1 []
    [DT 1  [([],[ Delta],1) ,([ Label P1 ],[],2)  ,([ Label P2 ],[],3)  ,([ Label P3 ],[],7) ]
    ,DT 2  [([],[ A1],1)]
    ,DT 3  [([],[ Delta],3) ,([ Label P1 ],[],4)  ,([ Label P3 ],[],5) ]
    ,DT 4  [([],[ A2],2)]
    ,DT 5  [([],[ Delta],5) ,([ Label P1 ],[],6) ]
    ,DT 6  [([],[ A3],4)]
    ,DT 7  [([],[ Delta],7) ,([ Label P1 ],[],8)  ,([ Label P2 ],[],5) ]
    ,DT 8  [([],[ A3],2)]]
```

Listing 4.5 Transformation algorithm DS to LTS

```haskell
-- implementation of decision structure to LTS algorithm
ds2lts  :: Eq a =>
           DS a Property Act ->
           LTS ((a,[Label]),[Prop]) Prop Action
ds2lts  (DS i qs  [])  = let  v = valuation qs
                              s = ((i ,[]) ,  v)
                         in   LTS s [(s,v)]   []
ds2lts  (DS i qs  ts)  = let
    vals  = concatMap collectValuations ts
    tuples  = ds2lts'  vals  ts  []
    states  = (nub . concat)  [  [s,t]  |  (_,s,t)  <- tuples ]
    valtable  = map (\(i,ps) -> ((i,ps),ps))  states
  in LTS ((i ,[]) ,valuation  qs)  valtable  tuples

-- tuple the table index with the valuation conditions  of  the  decision  rules
collectValuations  :: DT a Property b -> [(a,[Prop])]
collectValuations  (DT i  [])  = [(i ,[]) ]
collectValuations  (DT i ts)  = [(i,  valuation  qs)  |  (qs,_,_)  <- ts ]

-- filter  out the  label  conditions
valuation  ::  [Property] -> [Prop]
valuation  = sort  .  valuation'
  where valuation'  []  = []
        valuation'  ((Cond p):ps) = p : valuation'  ps
        valuation'  (_:ps)  = valuation'  ps

-- implementation of the big loop from the algorithm
-- resulting in the  transition  relation
ds2lts'  ::  Eq a =>
             [(a,[Prop])]  ->
             [DT a Property Act] ->
             [(Action,((a,[Label]),[Prop]),((a,[Label]),[Prop]))]  ->
             [(Action,((a,[Label]),[Prop]),((a,[Label]),[Prop]))]
ds2lts'  vals  = while2
                  (\tables  _  -> not (null tables))
                  (\tables  tuples  -> let
                      (table,rest)  = (head tables,  tail  tables)
                      tuples'  = dt2tuples table  vals
                    in  (rest,nub (tuples ++ tuples')))

-- Haskell implementation of a while loop with two variables
while2  ::  (a -> b -> Bool) -> (a -> b -> (a,b)) -> a -> b -> b
while2 p f  x y  |  p x y = let  (x',y')  = f x y
                            in  while2 p  f  x'  y'
                          |  otherwise = y

-- implementation of the loop over decision rules  of  a  table
dt2tuples  ::  Eq a =>
               DT a Property Act -> [(a,[Prop])] ->
               [(Action,((a,[Label]),[Prop]),((a,[Label]),[Prop]))]
dt2tuples  (DT i  [])  _ = []
dt2tuples  (DT i ts)  vals  = let
    tuples  = (nub . concat)
                [  makeTuples ((i,[]),valuation  qs)  (label  qs)  acts  k  vals
                |  (qs,acts,k)  <- ts ]
  in  tuples

-- filter  out the  label  conditions
label  ::  [Property] -> [Prop]
label  []  = []
label  ((Label p):ps)  = p :  label  ps
label  (_:ps)  = label ps
```

```haskell
-- decide whether the rule has no labels,  no input  labels ,  or  has input  labels
makeTuples :: (Eq a, Eq b) =>
                ((a,  [Label]),  b) -> [Prop] -> [Act] -> a -> [(a, b)] ->
                [(Action,  ((a,  [Label]),  b),  ((a,  [Label]),  b))]
makeTuples s []  []  k  vals  = endOfTrace s (Out Tau) k vals
makeTuples s [] acts  k  vals  = actionTuples s acts  k  vals
```

Listing 4.6 LTS from DS - Coffee Machine

```
-- Remember we defined the following labels
-- button1, button2, button3, coffee, cream, sugar :: Action
-- button1 = In P1
-- button2 = In P2
-- button3 = In P3
-- coffee = Out A1
-- cream = Out A2
-- sugar = Out A3

LTS 1
    -- states
     [(1,[])    -- s
     ,(2,[])    -- u
     ,(3,[])    -- v
     ,(4,[])    -- t
     ,(5,[])    -- w
     ,(6,[])    -- y
     ,(7,[])    -- z
     ,(8,[]) ]  -- x
    -- transition relation
    [(Out Delta,1,1)
    ,(In P1,1,2)
    ,(In P2,1,3)
    ,(In P3,1,4)
    ,(Out A1,2,1)
    ,(Out Delta,3,3)
    ,(In P1,3,5)
    ,(In P3,3,6)
    ,(Out A2,5,2)
    ,(Out Delta,6,6)
    ,(In P1,6,7)
    ,(Out A3,7,5)
    ,(Out Delta,4,4)
    ,(In P1,4,8)
    ,(In P2,4,6)
    ,(Out A3,8,2)]
```

Chapter 5

Conclusion

The aim of this paper was to provide an answer to the question whether decision tables can express the same behaviour as an LTS or STS can. My approach was to prove that it is possible to express one formalism in terms of the other, and vice versa. As a result of my research, this paper presents algorithms to do exactly that, although limited to LTSs only. Algorithm 3.2 describes how to go from a decision structure to a labelled transition system, while Algorithm 3.1 defines the inverse process. Proving that the transformation is both correct and lossless of information in both directions, I used the concept of bisimilarity. By showing that the result of a transformation and it's inverse is bisimilar, and thus equivalent, to the original model, I proved the correctness of the transformations, and showed that no information is lost in the process.

Chapter 6

Future work

Unfortunately I wasn't able to include transformations between decision structures and symbolic transition systems. This could be an area of future research, for I strongly believe that such transformations do exist. There is a strong correlation between labelled and symbolic transition systems (since the latter is derived from the former). My research showed that there is also a correlation between labelled transition systems and decision structures, which suggests that there should exist a correlation between decision structures and symbolic transition systems. Research in this direction could lift the use of decision tables for model-based testing from a theoretical idea to an applied method.

Another point for future research would be the actual value of using decision tables over symbolic transition systems. For my thesis I assumed that decision tables would be valuable because of their claimed usefulness for non-technicians [3, 4, 5]. However, this assumption is solely based on the claims of others that decision tables provide understanding through their tabular form. In order to verify my assumption an empirical research is needed, where the practice of decision tables in model-based testing is evaluated, and compared to the usage of symbolic transition systems. A start for such research could be a case study, where decision tables are used to model a real-world system.

In the following section I will discuss STSs, and give my views on how the transformation from and to decision structures could be handled.

6.1 Symbolic Transition Systems

Symbolic transition systems allow for the notion of data added to transitions through label parameters, or interaction variables, and provide a way of manipulating the data through update mappings. In addition to being state oriented, STSs make it possible to specify data-dependent behaviour.

Definition 6.1 (STS [1]).
A symbolic transition system is a tuple $\langle L, l_0, V, I, \iota, \Lambda, \rightarrow \rangle$ where

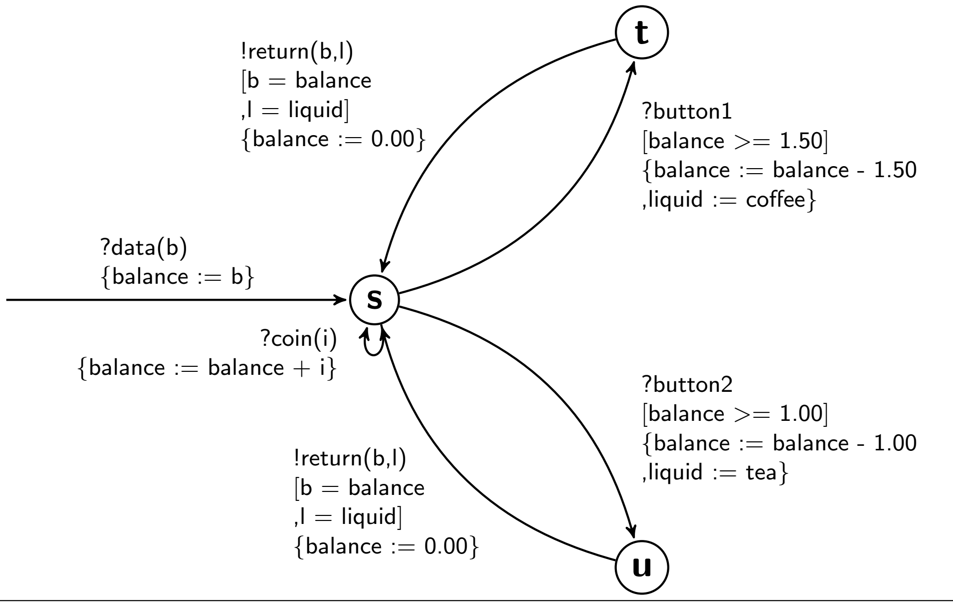
- L is a countable set of states, called *locations*;
- $l_0 \in L$ is the initial location;
- V is a countable set of *location variables*;
- I is a set of *interaction variables*, $V \cap I = \emptyset$;
- ι is an *initialisation* of the location variables;
- Λ is a countable set of observable transitions, called *gates*. The special gate τ represents unobservable gates, we write $\Lambda_\tau = \Lambda \cup \{\tau\}$. The *arity* of a gate $\lambda \in \Lambda_\tau$, denoted arity(λ), is a natural number. The *type* of a gate $\lambda \in \Lambda_\tau$, denoted type(λ), is a tuple of length arity(λ) of distinct interaction variables. The unobservable gate has no interaction variables, arity(τ) = 0;
- $\rightarrow \subseteq L \times \Lambda_\tau \times \mathfrak{F}(V \cup I) \times \mathfrak{T}(V \cup I)^V \times L$. $\mathfrak{F}(V \cup I)$ is the set of first-order formulas over the variables in $V \cup I$. $\mathfrak{T}(V \cup I)^V$ is the set of term-mappings from variables in V to terms over $V \cup I$.

$$\frac{l \xrightarrow{\lambda,\phi,\rho} l' \quad \text{type}(\lambda) = \langle v_1, \dots, v_n \rangle \quad \varsigma \in U^{\text{type}(\lambda)} \quad \theta \cup \varsigma \models \phi \quad \theta' = (\theta \cup \varsigma)_{\text{eval}} \circ \rho}{(l, \theta) \xrightarrow{(\lambda, \langle \varsigma(v_1), \dots, \varsigma(v_2) \rangle)} (l', \theta')}$$

Table 6.1: Deduction rule for transitions

To differentiate between input and output, the gate set is divided in input gates and output gates, $\Lambda = \Lambda_I \cup \Lambda_O$, with $\Lambda_I \cap \Lambda_O = \emptyset$. For a transition $(l, \lambda, \phi, \rho, l') \in \rightarrow$ we write $l \xrightarrow{\lambda,\phi,\rho} l'$, where ϕ is the transition guard and ρ is the update mapping. Similar to an LTS, an STS can be represented by a tree or graph. In addition to the action label, or gate name, edges are annotated with the *transition guards* (in square brackets) and *update mappings* (in curly braces), see Diagram 6.1. The set of locations is given by $L = \{s, t, u\}$, the set of location variables $V = \{balance, liquid\}$, the set of interaction variables $I = \{b, i, l\}$. The gates are $\Lambda = \{!return, ?coin, ?button1, ?button2\}$, with $!return$ in Λ_O and all other gates in Λ_I. This STS models a coffee machine as well, although slightly more complicated. In this model, the machine expects money as input, and checks whether enough is put in for either coffee (button1) or tea (button2) before transitioning to a next location. When the guards hold, the update mapping is executed, so for example the payment for a coffee is subtracted from the balance in the transition from s to t.

Diagram 6.1 Coffee Machine 2



The semantics of an STS can be defined by interpreting it in terms of an LTS. When a valuation θ *satisfies* a formula ϕ we write $\theta \models \phi$. The evaluation of a term based on a valuation θ is denoted θ_{eval}, or eval in the case of a term with no variables.

Definition 6.2 (Interpretation of STS [1]).
Let $S = \langle L, l_0, V, I, \iota, \Lambda, \rightarrow \rangle$ be an STS. The interpretation of S is given by the LTS $\|S\| = \langle q_0, Q, M, \rightarrow \rangle$ where

- $q_0 = (l_0, \text{eval} \circ \iota)$ is the initial state;
- $Q = L \times U^V$ is the set of states, with U a non-empty set
- $M = \bigcup_{\lambda \in \Lambda_\tau} (\{\lambda\} \times U^{\text{arity}(\lambda)})$ is the set of actions;
- $\rightarrow \subseteq Q \times M \times Q$ is the transition relation, defined by the rule in Table 6.1.

6.1.1 Bisimilarity for STSs

Two symbolic transitions systems s and t are bisimilar, if for every location s' reachable from s through gate λ under guard ϕ and after update-mapping π, there is a location t' reachable from t through the same gate λ, under the same guard ϕ and after the same update-mapping π, such that s' and t' are bisimilar.

Definition 6.1 (Bisimilarity of STSs).
Given a symbolic transition system $\langle L, l_0, V, I, \iota, \Lambda, \rightarrow\rangle$, a bisimulation is a binary relation $R \subseteq L \times L$ such that $\forall(s,t) \in R \,.\, \forall\lambda \in \Lambda \,.\, \forall\phi \in \mathfrak{F}(V \cup I) \,.\, \forall\rho \in \mathfrak{T}(V \cup I)^V$:

$$\forall s' \;.\; \text{if } (s \xrightarrow{\lambda,\phi,\rho} s') \text{ then } \exists t' \;.\; (t \xrightarrow{\lambda,\phi,\rho} t') \wedge (s',t') \in R \qquad \text{(Zig")}$$

$$\forall t' \;.\; \text{if } (t \xrightarrow{\lambda,\phi,\rho} t') \text{ then } \exists s' \;.\; (s \xrightarrow{\lambda,\phi,\rho} s') \wedge (s',t') \in R \qquad \text{(Zag")}$$

If there is a bisimulation R such that $(s,t) \in R$ we write $s \sim t$. To prove that bisimilarity for STSs is also an equivalence relation, we need to show that it is reflexive, symmetric and transitive. Again, this is very similar to what is done for bisimulation LTSs. We consider bisimilarity within a single STS.

Reflexive Assume $s \sim s$, then there is a bisimulation $R : s \sim s$. The identity relation is a bisimulation, $I : s \sim s$. Thus $s \sim s$.

Symmetric Assume $s \sim t$, then there is a bisimulation $R : s \sim t$. Note that (Zig") and (Zag") are symmetric. Thus, $R : s \sim t$ implies $R^{-1} : t \sim s$. Thus $t \sim s$.

Transitive Assume $s \sim t$ and $t \sim u$, then there are bisimulations $R : s \sim t$ and $R' : t \sim u$. It follows that $R \circ R' : s \sim u$. Thus $s \sim u$.

Given two symbolic transition systems $s_1 = \langle L_1, l_{0_1}, V_1, I_1, \iota_1, \Lambda, \rightarrow_1\rangle$ and $s_2 = \langle L_2, l_{0_2}, V_2, I_2, \iota_2, \Lambda, \rightarrow_2\rangle$, we say that $s_1 \sim s_2$ if there is a bisimulation R between s_1 and s_2 such that $(l_{0_1}, l_{0_2}) \in R$.

6.2 Decision structures and STSs

Where LTSs only have labels on transitions, transitions in STSs also have guards, update mappings, and different sets of gates for input and output. To translate these concepts to decision tables, we extend the definition of an action in a decision table to include the concept of proposition updates. In an STS guards are evaluated before the output label is processed, after which the update mapping is executed. Thus, proposition updates occur after their corresponding action label in the action sequence of a decision rule.

Definition 6.1 (Action).
An action is one of the following;

- $\lambda \in \Lambda_O$ is an *action label*, with Λ_O the set of output labels;

- ρ is a *proposition update*.

With the addition of proposition updates, it is now possible to make a decision table model that corresponds with the STS modelled in Diagram 6.1. See Table 6.2. This example demonstrates the use of a *dash* ('-') in the condition entry part. The meaning of this is that the valuation of the condition does not influence the valuation of the column. Possible occurrences are when two conditions do not exclude each other, such as '$x > 5$' and '$x < 3$', or when a condition simply has nothing to do with that particular column and the actions to be taken. A dash is used to signal that that condition can be skipped in the evaluation.

1					
?data(balance)	T	F	F	F	F
?coin(money)	F	T	F	F	F
?button1	F	F	T	F	F
?button2	F	F	F	T	F
b $\geq$ 1.50	-	-	T	-	F
b $\geq$ 1.00	-	-	-	T	F
b = balance	-	-	-	-	T
l = liquid	-	-	-	-	T
b := balance	X				
l := liquid					
b := b + money		X			
b := b - 1.50			X		
b := b - 1.00				X	
l := coffee			X		
l := tea				X	
b := 0.00					X
l := null					X
!return(balance,liquid)					X
goto 1	X	X	X	X	X

Table 6.2: Coffee Machine 2

6.2.1 Transforming an STS to a decision structure and vice versa

With the transformation of a decision structure to a labelled transition system and vice versa, we have a starting point for transforming to and from a symbolic transition system. Recall the definition for STSs, Definition 6.1 on page 27. Being based on labelled transition systems, the resemblance is clear. Instead of a simple label, a transition in a symbolic transition system is annotated with a tuple of a guard, an action label, and an update mapping. When the guard holds, the update mapping is applied and the action label is executed.

Similar to the transformation from a decision structure to an LTS, we can turn a reference from one table to another into a transition from one state to the other. Instead of using the valuation conditions as state valuations, we can make use them as the guard of the transition. Label conditions are still turned into input labels on transitions, but the treatment of actions changes.

We extended the definition of actions in decision tables to match the update mappings of transitions in STSs, see Definition 6.1 on page 29. Using this definition of actions, the update mapping of an action in a decision table can be used as the update mapping of a transition in an STS.

For the inverse transformation, we can let every state in an STS correspond with a decision table. If an outgoing transition has an input label, the label is added as a label condition, while the guard on the transition becomes valuation conditions. A transition with an output label adds the output label to the action of a decision tuple. The update mapping of a transition in the STS can be used as the update mapping of the action of a decision tuple in the resulting decision table.

Bibliography

[1] L. Frantzen, J. Tretmans, and T. Willemse. Test generation based on symbolic specifications. *Formal Approaches to Software Testing*, pages 1–15, 2005.

[2] L. Frantzen, J. Tretmans, and T. Willemse. A symbolic framework for model-based testing. *Formal approaches to software testing and runtime verification*, pages 40–54, 2006.

[3] P.J.H. King. Decision tables. *The Computer Journal*, 10(2):135–142, 1967.

[4] A. Lew and D. Tamanaha. Decision table programming and reliability. In *Proceedings of the 2nd international conference on Software engineering*, pages 345–349. IEEE Computer Society Press, 1976.

[5] U.W. Pooch. Translation of decision tables. *ACM Computing Surveys (CSUR)*, 6(2):125–151, 1974.

[6] J. Tretmans. Testing techniques. *Lecture notes, University of Twente, The Netherlands*, 1:1, 2002.

[7] J. Tretmans. Model based testing with labelled transition systems. *Formal methods and testing*, pages 1–38, 2008.