A Case Study in Formal Testing and an Algorithm for Automatic Test Case Generation with Symbolic Transition Systems

MSc Thesis (*Afstudeerscriptie*)

written by

Floor Sietsma
(born February 11, 1992 in Amstelveen)

under the supervision of **Dr Inge Bethke**, and submitted to the Board of Examiners in partial fulfillment of the requirements for the degree of

MSc in Logic

at the *Universiteit van Amsterdam.*

Date of the public defense:	**Members of the Thesis Committee:**
September 25, 2009	Dr Inge Bethke
	Dr Leen Torenvliet
	Prof Dr Frank Veltman

INSTITUTE FOR LOGIC, LANGUAGE AND COMPUTATION

Abstract

We present a case study of two black-box testing techniques. We compare constraint logic programming with symbolic transition systems. Both techniques generate automatically test cases out of the specification of an algorithm. After our case study we decided to design an algorithm to improve the technique using symbolic transition systems with a way to generate test cases automatically based on a coverage criterion. Because we do black-box testing and we have no access to the implementation of the algorithm, we apply this coverage criterion to the specification. Our algorithm attempts to cover all states of the symbolic transition system that models the specification of the algorithm. This task is complicated due to non-determinism and symbolic constraints in the model. We use an external constraint solver to solve these constraints. We try to simplify them as much as possible because constraint solvers only have limited power and often use a great amount of computational resources.

Contents

1 Introduction

The goal of automated testing is to rule out errors in the *implementation* of an algorithm by automatically generating and executing *test cases* for the algorithm. A test case consists of a certain input that should be given to the implementation and the corresponding output and behavior that conforms to the *specification* of the algorithm. If we execute the implementation on the input and the output does not correspond to the output given in the test case, we found an error.

In order to do automated testing we first of all need a way to generate this input. For example, it can be random, or it can be generated to match some user-defined restrictions.

We also need to know what output would be correct on this input, conform the specifications. We call a tool to generate these outputs a *test oracle*.

Finally, even though it is practically impossible to say when we have found enough test cases to rule out all errors, we still would like to have some idea of what part of the errors our test cases "covered". For this we need a *coverage criterion*. Having this coverage criterion could also give us information about which additional test cases we should choose. We then can use this information while generating new test cases.

There are a lot of different approaches to automated software testing. They can be divided in white-box techniques, that generate test cases based on the details of the implementation, and black-box techniques, that treat the implementation as a black box and generate test cases based on the specification of the algorithm. In this thesis we focus on black-box techniques.

2 Symbolic Transition Systems

One quite formal black-box testing technique uses Symbolic Transition Systems (STS) [8]. In this section we explain the basics of STSs which will be used intensively in the sequel.

2.1 First Order Logic

We use some basic concepts from first order logic. We assume a first order structure is given by:

- A logical signature $\mathfrak{S} = (F, P)$ where
  - F is a set of function symbols and each $f \in F$ has a corresponding arity $n \in \mathbb{N}$. If $n = 0$ we call f a constant symbol.
  - P is a set of predicate symbols. Each $p \in P$ has a corresponding arity $n > 0$.
- A model $\mathfrak{M} = (\mathfrak{U}, (f_{\mathfrak{M}})_{f \in F}, (p_{\mathfrak{M}})_{p \in P})$ where
  - $\mathfrak{U}$ is a non-empty set called the universe,
  - $f_{\mathfrak{M}} : \mathfrak{U}^n \mapsto \mathfrak{U}$ is a function on elements of the universe, where n is the arity of f,
  - $p_{\mathfrak{M}} \subseteq \mathfrak{U}^n$ is a predicate on elements of the universe,where n is the arity of p.

Let $\mathfrak{X}$ be a set of variables. $\mathfrak{T}(\mathfrak{X})$ is the set of terms and is inductively defined as the smallest set containing $\mathfrak{X}$ and for every $f \in F$ with arity n and for all $t_1, ..., t_n \in \mathfrak{T}(\mathfrak{X})$, $f(t_1, ..., t_n)$. Let $\mathfrak{F}(\mathfrak{X})$, the set of first order formulas over $\mathfrak{X}$, be inductively defined as the smallest set such that

- $\top, \bot \in \mathfrak{F}(\mathfrak{X})$,
- for every $p \in P$ with arity n, and every $t_1, ..., t_n \in \mathfrak{T}(\mathfrak{X})$, $p(t_1, ..., t_n) \in \mathfrak{F}(\mathfrak{X})$,
- if $\varphi, \psi \in \mathfrak{F}(\mathfrak{X})$ then also $\neg\varphi, \varphi \wedge \psi, \varphi \vee \psi \in \mathfrak{F}(\mathfrak{X})$,
- if $\varphi \in \mathfrak{F}(\mathfrak{X} \cup \{x\})$ and $x \notin \mathfrak{X}$ then $\forall x \varphi, \exists x \varphi \in \mathfrak{F}(\mathfrak{X})$.

We denote the free variables of a formula by $\mathbf{free}(\varphi)$.

2.2 Symbolic Transition Systems

STSs give a representation of the specification of an algorithm. They specify which input and output messages the implementation of the algorithm should accept and give. We can use them to test an implementation by checking whether the implementation gives the specified output on the specified input. STSs consist of locations, modeling a control state of the algorithm, and transitions between these locations, modeling the algorithm's input and output. An example is given in Figure 1. It is a simple STS representing the specification of a coffee machine that takes a coin as input (represented by a question mark) and gives a cup of coffee as output (represented by an exclamation mark).

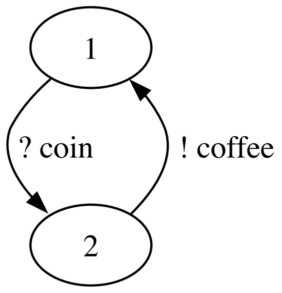


Figure 1: A simple STS modeling the specification of a coffeemachine

Symbolic transition systems are an extension of Labeled Transition Systems (LTS) [12]. The difference is that in an STS the input and output messages are labeled with a tuple of input or output variables called interaction variables. For example, `coin` could be an input variable with a value between 5 and 200 cent. The tuple of interaction variables can also be empty, in which case there is no input or output but only a transition to another location. We call such a message *unobservable* and we denote this by τ. These unobservable messages are comparable to the 'silent step' from process algebra.

Except for input and output variables, an STS also has location variables. These variables can help to specify complex relations between input and output variables and their value can change during execution of the implementation. At every transition the location variables can be updated. This is done with the update mapping, which is a mapping from the location variables to new values that can depend on other location variables or interaction variables.

Additionally, every transition has a guard. This is a logical formula over both the location variables and the interaction variables. The purpose of these guards is that during testing, we may only take the transition if the guard evaluates to $\top$.

Definition 1. *An STS is a tuple $\mathcal{S} = (L, l_0, \mathcal{V}, \mathcal{I}, \Lambda, \rightarrow)$, where*

- *L is a set of locations and l_0 is the initial location*
- *$\mathcal{V}$ is the set of location variables and $\mathcal{I}$ is the set of interaction variables*
- *We abbreviate $Var = \mathcal{V} \cup \mathcal{I}$*
- *$\Lambda \in \bigcup_{n \in \mathbb{N}} \mathcal{I}^n$ is a set of tuples of interaction variables (called* gates*). $\Lambda_\tau = \Lambda \cup \{\tau\}$*
- *$\rightarrow \subseteq L \times \Lambda_\tau \times \mathfrak{F}(Var) \times \mathfrak{T}(Var)^{\mathcal{V}} \times L$ is the transition relation. $l \xrightarrow{\lambda,\varphi,\rho} l'$ abbreviates $(l, \lambda, \varphi, \rho, l') \in \rightarrow$. Here λ is the gate, φ is the transition guard and $\rho : \mathcal{V} \mapsto \mathfrak{T}(Var)$ is the update mapping. This is a mapping from location variables to terms, which can also be seen as a substitution for location variables. In this substitution, all location variables for which no mapping is given in ρ are mapped to themselves. We denote with $\rho \circ \pi$ the concatenation of two update mappings ρ and π, and with $\varphi[\rho]$ the result of applying the substitution associated with update mapping ρ to formula φ.*
- $\mathbf{arity} : \Lambda_\tau \mapsto \mathbb{N}$ *gives the arity of each gate,* $\mathbf{arity}(\tau) = 0$.

As we can see in above definition, an STS can be non-deterministic. We come back to this fact in Section 7.

Definition 2. *An STS $\mathcal{S}$ is well-defined iff $l \xrightarrow{\lambda,\varphi,\rho} l'$ implies that all interaction variables in λ are distinct, $\mathbf{free}(\varphi) \subseteq \mathcal{V} \cup \lambda$ and $\rho : \mathcal{V} \mapsto \mathfrak{T}(\mathcal{V} \cup \lambda)$, so the update mapping maps all location variables to terms over location variables and only the interaction variables found in λ. We only consider well-defined STSs in this paper.*

Definition 3. *$\mathcal{S}(\iota)$ is an initialized STS, where $\iota : \mathcal{V} \mapsto \mathfrak{U}$ initializes all variables from $\mathcal{V}$ in l_0.*

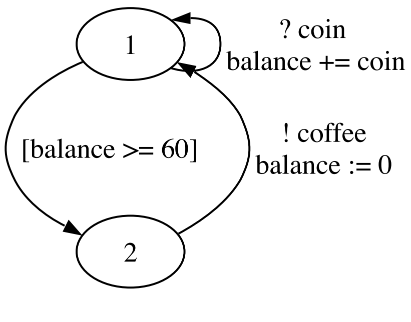


Figure 2: A coffee machine that keeps a balance

In Figure 2 we give an STS modeling the specification of a more complex coffee machine. It has a location variable `balance`. In location 1, every time a coin is given as input, the amount of the coin is added to the balance. If the balance is at least 60 cent we move to location 2 in the STS, the machine should give coffee, we reset the balance to 0 and return to location 1 again. Formally, the STS given in Figure 2 is

$$\begin{aligned}\mathcal{S} \;=\; & (\{1,2\}, 1, \{\texttt{balance}\}, \{\texttt{coin}, \texttt{coffee}\}, \{(\texttt{coin}), (\texttt{coffee})\},\\ & \{(1, (\texttt{coin}), \top, \{\texttt{balance} \mapsto \texttt{balance} + \texttt{coin}\}, 1),\\ & (1, \tau, \texttt{balance} \geq 60, \emptyset, 2),\\ & (2, (\texttt{coffee}), \top, \{\texttt{balance} \mapsto 0\}, 1)\}).\end{aligned}$$

We should keep in mind that STSs are meant to model the *specification* of an implementation, and not the implementation itself. Even though the above example may look a lot like the control flow graph of a coffee machine, it is only a representation of the specification of what the machine's input and output should look like. The implementation is correct if its input and output matches the input and output given in the STS, and because of this we can use the STS for software testing.

3 Constraints

We give a short introduction to Constraint Satisfaction Problems (CSP) [2]. First we give a definition of a constraint. A constraint over a set of variables restricts the possible values these variables may have. A formal definition is below.

Definition 4. *A constraint C over variables $x_1, ..., x_n$ with domains $D_1, ..., D_n$ is an n-ary relation on their domains, so $C \subseteq D_1 \times ... \times D_n$. An assignment of values $d_1 \in D_1, ..., d_n \in D_n$ to the variables $x_1, ..., x_n$ is said to solve the constraint C (or to be a solution to C) iff $(d_1, ..., d_n) \in C$.*

A CSP consists of a set of variables with their domains, and multiple constraints over subsets of these variables. The intention is to solve all constraints simultaneously with one assignment of all variables to elements of their domains.

Definition 5. *A CSP is a tuple $\langle\{x_1, ..., x_n\}, \{D_1, ..., D_n\}, \{C_1, ..., C_k\}\rangle$ where $x_1, ..., x_n$ are variables, $D_1, .., D_n$ are their respective domains, and $C_1, ..., C_k$ are constraints over subsets of $\{x_1, ..., x_n\}$. An assignment $d_1 \in D_1, ..., d_n \in D_n$ of all variables to elements in their domain solves a CSP (is a solution to the CSP) iff it solves all constraints $C_1, ..., C_k$ in the CSP. We call the CSP* consistent *if it has a solution and* inconsistent *otherwise.*

The reason we are interested in these constraints is that we can see the transition guards of an STS as constraints. For example, in the STS from Figure 2 the guard of the transition from location 1 to location 2 is a constraint on one variable, `balance`, with domain $\mathbb{N}$. The set of solutions to the constraint is $\{n \in \mathbb{N} \mid n \geq 60\}$.

If we solve the CSP that is given by the transition guards of a transition in an STS, we get a valuation of all interaction variables that gives a way to take the transition. We can use this to generate test cases that follow the transition.

In general, solving a CSP is an NP-complete problem, which is one of the reasons that automatic test case generation is a complex issue. Often CSPs can be solved using a form of search. We discuss some typically used techniques.

In backtracking techniques, we assign a value to all variables of the CSP in turn. For each variable we search all possible values for a value that is consistent with the assignments of earlier variables. If such a value cannot be found, we do backtracking and change the value of a previously assigned variable. There are a number of improvements on this basic backtracking method that are more efficient but all based on the same principle.

Another technique for solving CSPs tries to change the CSP in order to enforce a form of local consistency. This way, we turn the CSP into a simpler equivalent problem and we may even be able to show the unsatisfiability of the CSP. *Arc consistency* is a form of local consistency that is often used. Below we give a formal definition of arc consistency, which is taken from [4].

Definition 6. *In a CSP $\langle\{x_1, ..., x_n\}, \{D_1, ..., D_n\}, \{C_1, ..., C_k\}\rangle$, a value $d_i \in D_i$ is consistent with $C_1, ..., C_k$ iff there exists an assignment $d_1 \in D_1, ..., d_{i-1} \in D_{i-1}, d_{i+1} \in D_{i+1}, ..., d_n \in D_n$ such that $d_1, ..., d_n$ is a solution to the CSP. The entire CSP is arc consistent if every value of every member of $\{D_1, ..., D_k\}$ is consistent with $C_1, ..., C_k$ and no member of $\{D_1, ..., D_k\}$ is empty.*

We can enforce arc consistency by removing values from the domains of the variables. If we have removed values from the domain of one variable we have to check the domains of the other variables again because there might be values that are not consistent anymore after removing the value from the first domain. However, if we keep on checking all domains and removing non-consistent values, eventually all values will be consistent with the constraints. Then either the domain of some variable is empty and we have showed that the CSP is unsatisfiable, or the CSP is arc consistent and has been simplified by removing values that can never be part of a solution.

A lot of algorithms for solving CSPs have been implemented in constraint solvers. They often simplify the CSP to enforce some form of local consistency like arc consistency, and then use search techniques like backtracking to generate a solution of the CSP. We will use a constraint solver to generate solutions for the transition guards of STSs, as we discuss in Section 7 and 8.

4 A Case Study: The Blockbuster Case

We first present an optimization problem related to a video rental store. Our goal is to test an implementation of an optimization algorithm for this problem. We present the specifications such an algorithm should conform to and we show two different approaches to automatically generating test cases based on these specifications. The blockbuster case is based on an existing software system. This case is interesting because operations on internal data are important in the problem. Formal testing with STSs has so far concentrated on applications with a lot of input and output and few internal steps, and we want to investigate how STSs can be used for testing data-oriented algorithms.

4.1 The Optimization Problem

We assume there is a video store that rents out videos. These videos are distributed over different categories, for example Action, Drama and Documentary. Each of these categories can be subdivided into popularity classes, for example Top 40 and New Release. We will write “a type” of movies if we mean categories as well as popularity classes. Abstracting from the specific movies within each type, we can characterize a video store by the distribution of its movies over the categories and, within each category, over the popularity classes. An example of such a distribution is given in Table 1.

A formal definition is given in Definition 7. For each category and each popularity class within each category, we give the number and percentage of movies it contains. For the categories this is the percentage of the total number of movies, for the popularity classes within the categories it is the percentage of the number of movies in the category.

Definition 7. *A video store is a pair $\mathbb{S} = (C, P)$ such that C is a set of categories and P is a set of popularity types. A distribution for $\mathbb{S}$ is an $\eta : C \times P \mapsto \mathbb{N}$ that assigns to each combination of a category and a popularity class a number of movies. We define $\eta : C \mapsto \mathbb{N}$ as $\eta(c) = \sum_{p \in P} \eta(c, p)$.*

Definition 8. *For each video store $\mathbb{S}$ with distribution η we define a relative distribution ϱ such that it gives for each combination of a category and a popularity class the percentage of movies within the category in that popularity class, so $\varrho(c, p) = \frac{\eta(c,p)}{\eta(c)} \cdot 100$. We define $\varrho : C \mapsto \mathbb{N}$ such that it gives for each category the percentage of the total number of movies it contains, so $\varrho(c) = \frac{\eta(c)}{\sum_{d \in C} \eta(d)} \cdot 100$.*

There is some optimal distribution, based on the customers’ likings. For every type there is a minimal, optimal and maximal value for the percentage of movies it contains. An example is given in Table 2. Optimally, each category contains 25% of the movies, and within each category, 40% of the movies are from the top 40, 30% are new releases and 30% is in the remaining popularity class. For each type there is also a minimal and a maximal value. The point of these values is that the percentage of movies in the type should certainly not come below the minimum or above the maximum. Unlike the optimal values, the minimal and maximal values of the categories and popularity classes do not add up to 100% because they are independent for each type.

Definition 9. *An optimality profile for a video store $\mathbb{S} = (C, P)$ is a triple $\mathcal{P} = (min, opt, max)$ such that $min : C \cup P \mapsto [0, 100]$, $opt : C \cup P \mapsto [0, 100]$ and $max : C \cup P \mapsto [0, 100]$ assign to each category and popularity class a percentage such that for any $t \in C \cup P$, $min(t) \leq opt(t) \leq max(t)$. Additionally we require that $\sum_{c \in C} opt(c) = 100$ and $\sum_{p \in P} opt(p) = 100$.*

Definition 10. *A video store optimization problem is a tuple $\mathcal{O} = (C, P, \eta, min, opt, max)$ such that $\mathbb{S} = (C, P)$ is a video store, η is a distribution for $\mathbb{S}$ (denoting the current distribution), and (min, opt, max) is an optimality profile for $\mathbb{S}$.*

In Definition 11 we give a formal definition of what it means for a category or a popularity class to be optimal and we define the notions of a major or minor shortage or excess. A

Category	Popularity class	Number	Percentage
Action		**200**	**10%**
	Top 40	60	30%
	New Release	40	20%
	Other	100	50%
Drama		**500**	**25%**
	Top 40	250	50%
	New Release	200	40%
	Other	50	10%
Documentary		**500**	**25%**
	Top 40	175	35%
	New Release	175	35%
	Other	150	30%
Music		**800**	**40%**
	Top 40	160	20%
	New Release	160	20%
	Other	480	60%

Table 1: A distribution of the videos in a rental store. For each category and popularity type we display the number and percentage of movies it contains.

		minimal	optimal	maximal
Category	Action	15%	25%	70%
	Drama	5%	25%	35%
	Documentary	5%	25%	45%
	Music	20%	25%	35%
Popularity class	Top 40	10%	40%	50%
	New Release	20%	30%	45%
	Other	5%	30%	40%

Table 2: An optimality profile for the categories and popularity classes.

shortage or excess occurs when the percentage of movies in a type is below or above the optimum; it is a major shortage or excess if it is below the minimum or above the maximum.

We also define what it means for a shop's distribution to be optimal. For this it is sufficient if just the percentage of movies in each category is optimal, and not necessarily also the popularity classes.

Definition 11. *A category $c \in C$ is optimal in a video store optimization problem $\mathcal{O} = (C, P, \eta, min, opt, max)$ with relative distribution ϱ iff $\varrho(c) = opt(c)$. It has a major excess iff $\varrho(c) > max(c)$, a minor excess iff $max(c) \geq \varrho(c) > opt(c)$, a major shortage iff $\varrho(c) < min(c)$ and a minor shortage iff $min(c) \leq \varrho(c) < opt(c)$. Similarly, a popularity class p is optimal within a category c iff $\varrho(c, p) = opt(p)$, and we define major and minor shortages and excesses for popularity classes in a similar fashion. $\mathcal{O}$ is optimal iff every category in C is.*

The shop's distribution changes as movies are rented out, and to maintain the optimal distribution, once in a while the shop buys and sells movies from and to the distribution center. The optimization problem we consider is the question which movies should be bought and sold in order to optimize the shop's distribution.

4.2 The Specification

We want to test the implementation of an algorithm that solves this problem. Since we do black-box testing, we have no access to the details of the implementation and we generate our test cases based on the specifications. The algorithm should not only solve the problem, but it should do so in a certain manner. The specifications give restrictions on the output of the algorithm, and through the output, it restricts the manner in which the problem should be solved. We will explain these restriction in this section.

The input of the algorithm is a video store optimization problem: the current distribution of the shop and an optimality profile. The output of the algorithm should be a succession of transactions, where each transaction consists of a set of movies to buy and a set of movies to sell.

Definition 12. *A transaction for a video store optimization problem $\mathcal{O} = (C, P, \eta, min, opt, max)$ is a pair $\mathcal{T} = (\beta, \sigma)$ where $\beta : C \times P \mapsto \mathbb{N}$ denotes for each category and popularity type the number of movies to buy and $\sigma : C \times P \mapsto \mathbb{N}$ denotes the number of movies to sell in a similar fashion. The result of updating $\mathcal{O}$ with $\mathcal{T}$ is $\mathcal{O}' = (C, P, \eta', min, opt, max)$ where $\eta'(c, p) = \eta(c, p) + \beta(c, p) - \sigma(c, p)$.*

One obvious restriction is that the transactions should be such that the shop's distribution is optimal after executing all transactions. Another restriction is that within one transaction, we buy from one single category and sell from another single category. Also, the sets of movies to buy and movies to sell should be equally large so the total number of movies in the store stays the same after every transaction. The size of these sets should be the minimum of the shortage in the category we buy from and the excess in the category we sell from, so at least one of the two categories will be optimal after updating with the transaction.

There is a specific order in which the different categories should be chosen. First we have to handle all major excesses and shortages. If there are multiple major excesses or shortages, we handle the biggest ones first. After that we should handle the remaining minor shortages and excesses, again the biggest ones first.

Even though the percentages of movies in the popularity classes within the categories do not have to be optimal for the shop's distribution to be so, there are still restrictions on the popularity classes to buy and sell from. Unlike the categories, we are allowed to buy and sell from multiple popularity classes within one transaction. The total number of movies to buy and sell is already determined by the shortage and excess in the buying and selling category. We should choose several popularity classes from which we buy within the buying category until they become optimal, to a total maximum of the number of movies we want to buy overall. Also, we should do the same for the popularity classes within the selling category.

This means there might be one popularity class in the buying category and one in the selling category from which we buy less than is needed to make the percentage of movies in that popularity class optimal, because we have reached the total maximum.

Also with the popularity classes, we should handle the major shortages and excesses first, and if there are multiple major shortages or excesses we handle the biggest ones first. Then we handle the minor shortages and excesses, again the biggest ones first.

Example 1. *For the shop distribution in Table 1 and the optimality profile in Table 2, the first transaction in the output conform the specification of the Blockbuster case has the buying category Action, because Action has a major shortage and there is no other category with a major shortage. It has selling category Music, because Music has a major excess and there is no other category with a major excess. The number of movies to buy and sell is 15% of the total number of movies, which makes the percentage of movies in both categories optimal. This comes down to 300 movies.*

Within Action, we have to recalculate the percentages of movies in the different popularity classes, because the total number of movies in Action will go up to 500. Top 40 now contains $\frac{60}{500} \cdot 100 = 12\%$ *of the movies, New Release contains* 8% *and Other contains* 20%. *This means we have a minor shortage of* $40 - 12 = 28\%$ *in Top 40 which comes down to 140 movies, a major shortage of* 22% *in New Release which comes down to 110 movies, and a minor shortage of* 10% *in Other which comes down to 50 movies. So we certainly want to buy from New Release, the next priority is Top 40 and Other comes after that. It turns out that all shortages sum up to 300, so we can buy from all three popularity classes until they are optimal.*

Within Music, the total number of movies also becomes 500. This brings the number of movies in Top 40 to 32%, in New Release to 32% and in Other to 96%. This means we have a minor shortage of 8% in Top 40, a minor excess of 2% in New Release, which comes down to 10 movies, and a major excess of 66% in Other, which comes down to 330 movies. This means that we certainly want to sell from Other, and maybe also from New Release. Since we will only sell 300 movies, we will sell 300 movies from Other.

So the first transaction in the correct output is to buy within Action 110 movies from New Release, 140 movies from Top 40 and 50 movies from Other, and to sell within Music 300 movies from Other. This results in the distribution in Table 3. The percentage of movies in all categories is now optimal, so the entire shop is optimal. The correct output is the sequence of this single transaction.

Definition 13. *An implementation conforms to the specification of the Blockbuster case iff it takes a video store optimization problem* $\mathcal{O} = (C, P, \eta, min, opt, max)$ *as input, gives a finite sequence of transactions* $\langle(\beta_1, \sigma_1), ..., (\beta_n, \sigma_n)\rangle$ *as output and terminates, such that the following conditions hold. Here* $N = \sum_{c \in C} \eta(c)$ *is the total number of movies in the video store,* $\mathcal{O}_k = (C, P, \eta_k, min, opt, max)$ *is the result of updating* $\mathcal{O}$ *with* $(\beta_1, \sigma_1), ..., (\beta_{k-1}, \sigma_{k-1})$ *and* ϱ_k *is the relative distribution associated with* η_k.

1. *For any transaction* (β_k, σ_k), $1 \leq k \leq n$, *there are two distinct categories* c *and* d *such that the following hold:*
   - c *has a shortage and* d *an excess in* $\mathcal{O}_k$,
   - *For all popularity classes* p, *for all categories* $c' \neq c$, $\beta_k(c', p) = 0$,
   - *For all popularity classes* p, *for all categories* $d' \neq d$, $\sigma_k(d', p) = 0$,
   - $\sum_{p \in P} \beta_k(c, p) = \sum_{p \in P} \sigma_k(d, p) = \min(opt(c) - \varrho_k(c), \varrho_k(d) - opt(d)) \cdot \frac{N}{100}$.

   We call c *the buying category and* d *the selling category of* (β_k, σ_k).
2. *For any transaction* (β_k, σ_k) *with buying category* c, *for any other category* c' *with a shortage in* $\mathcal{O}_k$ *one of the following must hold:*
   - c *has a major shortage in* $\mathcal{O}_k$ *and* c' *has a minor shortage in* $\mathcal{O}_k$,
   - c *and* c' *both have a major shortage in* $\mathcal{O}_k$ *and* $|opt(c) - \varrho_k(c)| \geq |opt(c') - \varrho_k(c')|$,
   - c *and* c' *both have a minor shortage in* $\mathcal{O}_k$ *and* $|opt(c) - \varrho_k(c)| \geq |opt(c') - \varrho_k(c')|$.

Category	Popularity class	Number	Percentage
Action		**500**	**25%**
	Top 40	200	40%
	New Release	150	30%
	Other	150	30%
Drama		**500**	**25%**
	Top 40	250	50%
	New Release	200	40%
	Other	50	10%
Documentary		**500**	**25%**
	Top 40	175	35%
	New Release	175	35%
	Other	150	30%
Music		**500**	**25%**
	Top 40	160	32%
	New Release	160	32%
	Other	180	36%

Table 3: The distribution from Table 1 after updating conform the specification of the Blockbuster case

A similar condition holds for the selling category.

3. *For any transaction (β_k, σ_k) with buying category c, let $M = \sum_{p \in P} \eta_k(p, c) + \beta_k(c, p) - \sigma_k(c, p)$ be the total number of movies in the buying category after updating with (β_k, σ_k). We define ϱ' as $\varrho'(c, p) = \frac{\eta_k(c,p)}{M} \cdot 100$. This is the number of movies in p before updating with (β_k, σ_k), relative to the number of movies in c after updating with (β_k, σ_k). The following should hold:*
   - *For any popularity class p, if $\beta(c, p) > 0$ then $\varrho'(p) < opt(p)$*
   - *There should be one popularity class p such that $\beta(c, p) \leq opt(p) - \varrho'(p) \cdot \frac{M}{100}$ and for any other popularity class $p' \neq p$, $\beta(c, p') = 0$ or $\beta(c, p') = opt(p') - \varrho'(p') \cdot \frac{M}{100}$.*
   - *For any two distinct popularity classes p, q, if either $\beta(c, p) = opt(p) - \varrho'(p) \cdot \frac{M}{100}$ and $\beta(c, q) < opt(q) - \varrho'(q) \cdot \frac{M}{100}$ or $\beta(c, p) > 0$ and $\beta(c, q) = 0$ then one of the following should hold:*
     - $\varrho'(c, p) < min(p)$ *and* $\varrho'(c, q) \geq min(p)$,
     - $\varrho'(c, p) < min(p)$, $\varrho'(c, q) < min(p)$ *and* $opt(p) - \varrho'(c, p) \geq opt(q) - \varrho'(c, q)$,
     - $\varrho'(c, p) \geq min(p)$, $\varrho'(c, q) \geq min(p)$ *and* $opt(p) - \varrho'(c, p) \geq opt(q) - \varrho'(c, q)$.

   A similar condition holds for the selling category and σ.

4. *The result of updating $\mathcal{O}$ with $(\beta_1, \sigma_1), ..., (\beta_n, \sigma_n)$ should be optimal.*

Proposition 1. *Any sequence of transactions satisfying conditions 1 - 3 from Definition 13 is finite and also satisfies condition 4.*

Proof. Let $\mathcal{O} = (C, P, \eta, min, opt, max)$ be a video store optimization problem with relative distribution ϱ and let (β_1, σ_1), ... be a sequence of transactions for $\mathcal{O}$. Suppose these transactions satisfy condition 1 - 3. We will show that this sequence is finite and that the result of updating $\mathcal{O}$ with the transactions in this sequence is optimal.

Let $N = \sum_{c \in C} \eta(c)$ be the total number of movies in the video store. It is trivial to show that this number does not change after updating with any transaction that satisfies condition

1, so we assume N stays constant during the process of updating $\mathcal{O}$ with this sequence of transactions. Again, let $\mathcal{O}_k = (C, P, \eta_k, \varrho_k, min, opt, max)$ with relative distribution ϱ_k be the result of updating $\mathcal{O}$ with transactions $(\beta_1, \sigma_1), ..., (\beta_{k-1}, \sigma_{k-1})$, for finite k.

By condition 1, every transaction (β_k, σ_k) in the sequence has a buying category c and a selling category d. Also, the number of movies we buy and sell is the minimum of the amount we need to make the buying category optimal and the amount we need to make the selling category optimal: $\sum_{p \in P} \beta_k(c,p) = \sum_{p \in P} \sigma_k(d,p) = \min(opt(c) - \varrho_k(c), \varrho_k(d) - opt(d)) \cdot \frac{N}{100}$. This means that either the buying category or the selling category is optimal after updating with (β_k, σ_k), as we will now show formally.

Suppose $opt(c) - \varrho_k(c) \leq \varrho_k(d) - opt(d)$. Then $\sum_{p \in P} \beta_k(c,p) = (opt(c) - \varrho_k(c)) \cdot \frac{N}{100}$. Then

$$\begin{aligned}
\eta_{k+1}(c) = \sum_{p \in P} \eta_{k+1}(c,p) &= \sum_{p \in P} \eta_k(c,p) + \beta_k(c,p) - \sigma_k(c,p) \\
&= \sum_{p \in P} \eta_k(c,p) + \sum_{p \in P} \beta_k(c,p) \\
&= \eta_k(c) + (opt(c) - \varrho_k(c)) \cdot \frac{N}{100}
\end{aligned}$$

where we use the fact that since c is not the selling category, $\sigma_k(c,p) = 0$ for any $p \in P$. Then

$$\begin{aligned}
\varrho_{k+1}(c) &= \frac{\eta_{k+1}(c)}{N} \cdot 100 \\
&= \eta_k(c) \cdot \frac{100}{N} + opt(c) - \varrho_k(c) \\
&= \varrho_k(c) + opt(c) - \varrho_k(c) \\
&= opt(c)
\end{aligned}$$

so c is optimal in $\mathcal{O}_{k+1}$.

For the case that $opt(c) - \varrho_k(c) > \varrho_k(d) - opt(d)$ we can use a similar proof to show that d is optimal in $\mathcal{O}_{k+1}$. So either the buying category or the selling category of (β_k, σ_k) is optimal in $\mathcal{O}_{k+1}$.

For any category c, if c is optimal in $\mathcal{O}_k$ then by condition 1 it is not the buying or selling category of (β_k, σ_k), so $\beta_k(c,p) = \sigma_k(c,p) = 0$ for any $p \in P$. So $\eta_{k+1}(c) = \eta_k(c) + \sum_{p \in P} \beta_k(c,p) - \sum_{p \in P} \sigma_k(c,p) = \eta_k(c)$ and $\varrho_k(c) = \frac{\eta_k(c)}{N} \cdot 100 = \frac{\eta_{k+1}(c)}{N} \cdot 100 = \varrho_{k+1}(c)$. And since c is optimal in $\mathcal{O}_k$, $opt(c) = \varrho_k(c) = \varrho_{k+1}(c)$. So if c is optimal in $\mathcal{O}_k$, then c is also optimal in $\mathcal{O}_{k+1}$ and indeed in every $\mathcal{O}_j$ with $j \geq k$.

So every transaction (β_k, σ_k) results in the optimization of one category that was not optimal in $\mathcal{O}_k$, and this category stays optimal after the update with every successive transaction. Since there are only finitely many categories, this means that eventually all categories will be optimal, so the sequence $(\beta_1, \sigma_1), ...$ is finite and results in an optimal distribution. $\square$

4.3 An Optimization Algorithm

Even though we don't have access to the details of the implementation of the optimization algorithm, we designed an imperative algorithm that conforms to the specification in the previous section. This algorithm is very straightforward given the specification.

The algorithm works with several iterations. In each iteration we choose one category to buy from and one category to sell from. Within these categories we buy or sell from one or more popularity classes. At the end of the iteration we update the shop's distribution with these changes. We keep on iterating until the shop's distribution is optimal.

Pseudocode for the algorithm is in Algorithm 1. An iteration starts by determining from which category to buy. As we discussed in the previous section, this is a category with the greatest major shortage. Similarly, we choose the category to sell from as one with the greatest major excess. If there is no major shortage or there is no major excess, we choose a category with the greatest minor shortage or excess.

After that, we decide how many movies to buy and sell. As we discussed in the previous section we want to keep the total number of movies in the shop constant so we buy as many movies as we sell. This number is the minimal number of movies we need to buy and sell to make the percentage of movies in either the "buying" or "selling" category optimal. This means that the percentage of movies in at least one of these categories becomes optimal, but the other category may still have a shortage or an excess. This is fine because then it will be addressed in some later iteration.

At this point we have decided how many movies to buy and sell from which categories. Next, we need to decide from which popularity classes these movies should be. We keep track of the number of movies we're planning to buy and sell from each popularity class, and keep on picking new popularity classes to buy and sell from until the sum over all popularity classes equals the total number of movies we want to buy and sell. Just like with the categories, we pick the popularity classes with the greatest major shortages and excesses first, and the ones with the greatest minor excesses and shortages after that.

The control flow graph of Algorithm 1 is depicted in Figure 3. The nodes are numbered with the line numbers of Algorithm 1.

Theorem 1. *Algorithm 1 conforms to the specification of the Blockbuster case*

Proof. It is trivial to show that Algorithm 1 satisfies conditions 1 - 3 from Definition 13. Then by Proposition 1 it also satisfies condition 4 and conforms to the Blockbuster specification. □

We will now discuss two different approaches to generating test cases for the Blockbuster case.

4.4 A Constraint Logic Programming Approach

Constraint Logic Programming (CLP) is the incorporation of constraints in a logic programming language. In CLP we state a set of constraints over some set of variables. We can then use a constraint solver to automatically find the possible values these could have in order to make the constraints true.

Because the specifications from Section 4.2 are formulated as restrictions on the output of the implementation, they can easily be stated as constraints. J.C. van de Pol[1] provided an implementation of the Blockbuster specification as a declarative program with constraints in Eclipse[2] Prolog, an open-source CLP framework [3]. The specification implemented in this program is simplified by leaving out the popularity classes. It is straightforward to add the popularity classes to this implementation. However, for our purposes this simplified version was sufficient because it already shows the use of CLP for modeling specifications.

As we discussed in Section 1, for automated testing we need a test oracle, a way to generate input for our test case and a coverage criterion. We can use Prolog to solve the constraints of the Blockbuster specification to get the correct output on a certain input. This gives us a test oracle.

Additionally, we can use Prolog's backtracking abilities to compute the input we should give to get a certain test case. For example, we could say we want a test case that starts with a major excess and a major shortage, and needs two transaction to become optimal. We could use Prolog backtracking and constraint solving to generate an input that gives a test case with these properties. An example of what this will look like is given in Figure 4. We ask for a test case with these properties, and we get a profile, a shop distribution and the sequence of transactions the implementation should give as output on the input of this profile and distribution.

[1] `http://www.home.cs.utwente.nl/~vdpol/`
[2] `http://www.eclipse-clp.org/`

Algorithm 1 The optimization algorithm of the Blockbuster case

```
while the shop distribution is not optimal do
  if there is at least one category with a major shortage then
    C := the category with the largest major shortage
  else
    C := the category with the largest minor shortage
  end if

  if there is at least one category with a major excess then
    D := the category with the largest major excess
  else
    D := the category with the largest minor excess
  end if
  N := min(shortage(C), excess(D))

  SUM := 0
  while SUM < N do
    if there is at least one popularity class with a major shortage within C then
      P := the popularity class with the largest major shortage within C
    else
      P := the popularity class with the largest minor shortage within C
    end if
    K := min(shortage(P), N - SUM)
    set the number of movies to buy from P within C to K
    shortage(P) := 0
    SUM += K
  end while

  SUM := 0
  while SUM < N do
    if there is at least one popularity class with a major excess within D then
      P := the popularity class with the largest major excess within D
    else
      P := the popularity class with the largest minor excess within D
    end if
    K := min(excess(P), N - SUM)
    set the number of movies to sell from P within D to K
    excess(P) := 0
    SUM += K
  end while
  update the shop distribution
end while
```

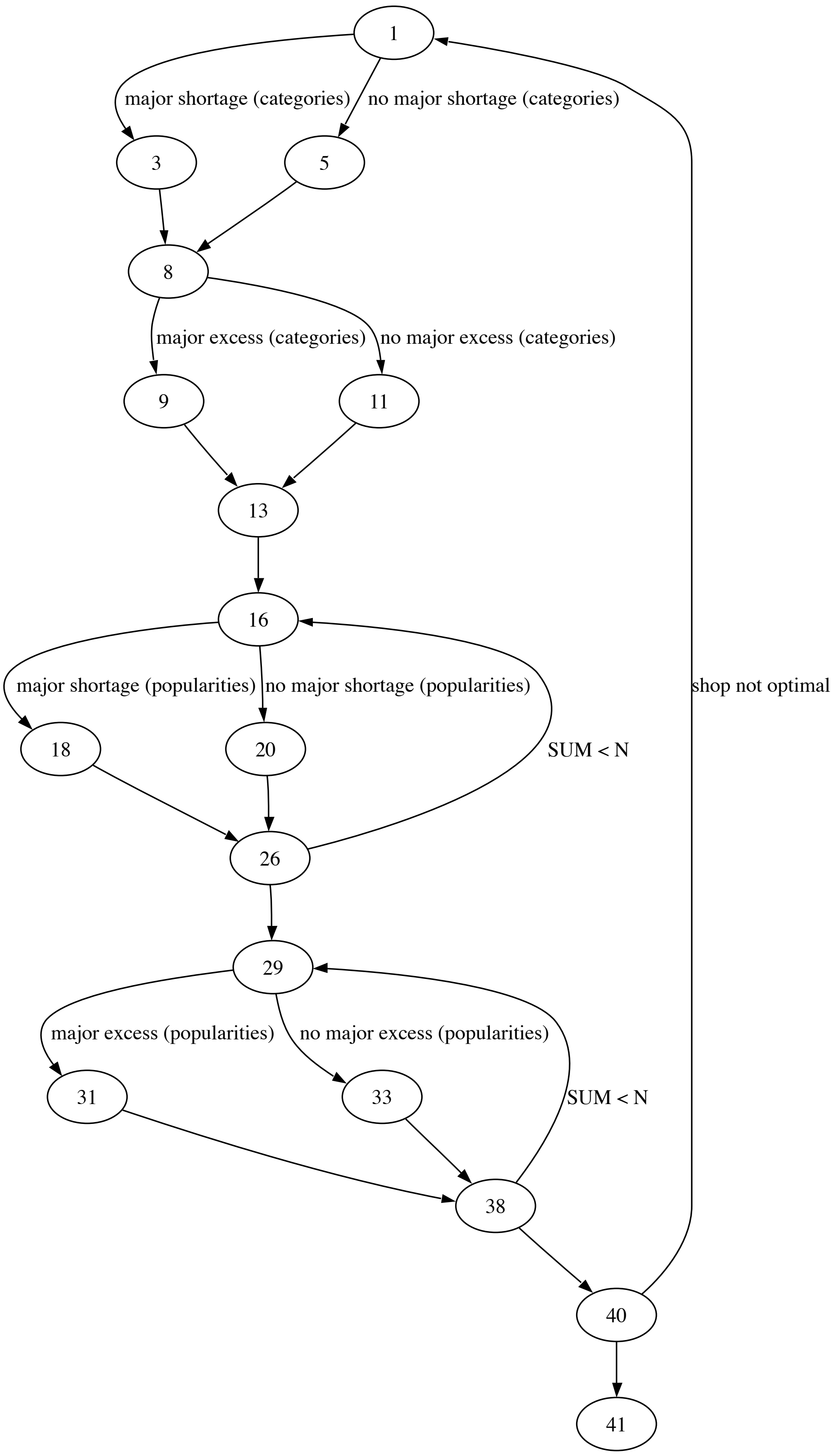

1
major shortage (categories)
no major shortage (categories)
3
5
8
major excess (categories)
no major excess (categories)
9
11
13
16
major shortage (popularities)
no major shortage (popularities)
shop not optimal
18
20
SUM < N
26
29
major excess (popularities)
no major excess (popularities)
31
33
SUM < N
38
40
41

Figure 3: The Control Flow Graph for Algorithm 1

```
:- generate_test(
   [swap(_,buy(major,_),sell(major,_)),
    swap(_,buy(_,_),sell(_,_))]).

Profile:
action    [15, 25, 50]
drama     [20, 25, 30]
docu      [10, 25, 60]
music     [5, 25, 40]

Shop:
[10, 40, 15, 35]

Run:
[10, 40, 15, 35]
swap(15, buy(major, action), sell(major, drama))
[25, 25, 15, 35]
swap(10, buy(minor, docu), sell(minor, music))
[25, 25, 25 25]
```

Figure 4: An example of the use of CLP for the Blockbuster case.

This is a very nice approach because we can request any kind of test case we like and immediately get the corresponding input and output. This input is chosen to match some user-defined restrictions (for example, there should be a major shortage and excess before the first transaction). The form of these restrictions depends greatly on the details of the Prolog implementation. In principle almost any kind of restriction is possible.

In this approach we did not define a coverage criterion, so we do not have an exact way to say which part of the implementation we "covered" with our test cases.

4.5 A Model-Based Testing Approach with Symbolic Transition Systems

A typical application of STSs is modeling the specifications of algorithms that have a lot of input and output. Usually all transitions have one or more input or output variables, because we only make a transition in the model if there is an input or output.

The Blockbuster specification has only one input and one sequence of outputs. If we apply the usual STS approach to this case we would get something like the STS in Figure 5. This STS has two locations. The first location is before the input is given. The transition from the first to the second location is the input: the shop's current distribution and the optimality profile. The transition from the second location back to itself is the output of one transaction. The implementation should give a sequence of transactions as output until the shop is optimal, in which case we move back to location 0 with an unobservable transition in the STS.

The restrictions on the output that are part of the specification of the algorithm are encoded in one big formula which is the guard of the output transition. We only give the beginning of this formula because it is very large, but it is clear that this STS is not very comprehensible.

In order to get a more comprehensible STS we decided to use more unobservable transitions (transitions without input or output) to separately model the different parts of the

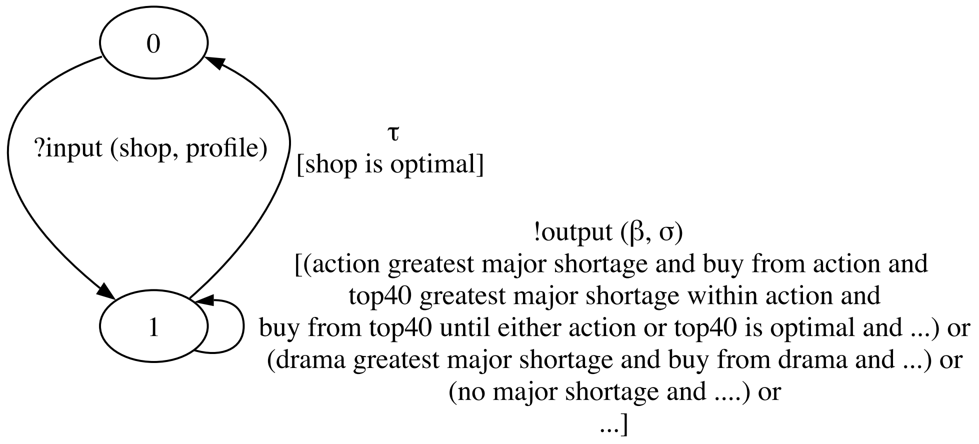

0
?input (shop, profile)
τ
[shop is optimal]
!output (β, σ)
[(action greatest major shortage and buy from action and
top40 greatest major shortage within action and
buy from top40 until either action or top40 is optimal and ...) or
(drama greatest major shortage and buy from drama and ...) or
(no major shortage and ....) or
...]
1

Figure 5: A small but complex STS for the Blockbuster algorithm

specification. The result is given in Figure 6. This STS has a lot more locations, but the transition guards are a lot smaller.

We begin in location 0, where we give the input: the current distribution of the shop and the optimality profile. In location 1 we have the possibility to go to location 3 if there is a category with a major shortage. There is a transition for every category (`action` and `drama` in this case), we can only take the transition if that category has the greatest major shortage, and then we also restrict the buying category to be that category. If no category has a major shortage, we can take the transition to location 2. From there we can go to location 3 if one of the categories has a minor shortage, and we restrict the buying category to be that category. This way we have specified the first part of the specification: the buying category should be the category with the greatest major shortage if it exists, and the category with the greatest minor shortage otherwise.

In location 3 we continue with the selling category. The selling category should be a category with the greatest major excess if it exists, and the greatest minor excess otherwise. We model this in a way very similar to the previous part: if there is a major excess, we immediately restrict the selling category to the category with the greatest major excess and switch to location 5, and if there is no major excess, we switch to location 4. In location 4 we restrict the selling category to be the category with the greatest minor excess and we also switch to location 5.

In location 5 we model the next part of the specification: we restrict the number of movies to buy from each popularity class within the buying category. For every popularity class there is a transition for the case that it has the greatest major shortage, where we restrict the number of movies to buy from that popularity class to the difference between the current number of movies it contains and the optimum. When we have restricted a certain popularity class we don't count it as a major shortage anymore and we can take the transition for a different popularity class, if that class has the greatest major shortage after the one we already restricted. If there are no more popularity classes that have a major shortage, we can take the transition to location 6. In location 6 we restrict the popularity classes with a minor shortage in a similar way.

If in location 5 or 6 the sum of the number of movies we plan to buy from each popularity class reaches the total number of movies we want to buy, we take the transition to location 7. In location 7 and 8 we restrict the number of movies to sell from each popularity class in a similar way: first we restrict the classes with major excesses in location 7, then we switch to location 8 and restrict the popularity classes with minor excesses. If in location 7 or 8 we reach the total number of movies to sell, we switch to location 9.

In location 9 we have applied all restrictions on one transaction from the sequence that forms the entire output. We give this transaction as output, update the shop with this transaction, and return to location 1. In location 1 we can either take the transition back to location 0 if the shop distribution is optimal, or otherwise take a transition to location 2 or 3 to restrict a new transaction.

This STS looks a lot like the Control Flow Graph in Figure 3. Again, we should keep in mind that the STS only models the specification of the implementation, and not the implementation itself. The reason they are so alike is that Algorithm 1 has been designed with the specification in mind, and the specification is quite strict and leaves little room for design choices.

Every run of a test case takes a certain path (or, in the case of non-determinism, a set of paths) through the STS, based on the input it gives to the implementation and the output the implementation gives back. We can use the STS for testing by giving input to the implementation, following the path that corresponds to the input through the STS, and checking whether the behavior of the implementation is a possible path through the STS. If it is a possible path, the implementation behaves correctly. If there is no path through the STS that matches the output of the implementation, then the output is wrong and we found an error.

The STSimulator Java library[3] by Lars Frantzen[4] can model an STS and simulate the paths a certain test case can take through the STS. We can instantiate input or output variables to our liking. The library then uses constraint solving to keep track of the possible values of the uninstantiated variables and to compute the values a variable should have in order to make certain transition guards true. We can use this to find out which output is possible given a certain input and the restrictions of the specification that are given in the STS. This gives us a test oracle. We can also instantiate no variables at all, and ask for the values of both input and output variables. This is a way to generate the input of a test case (and compute the corresponding output). However, the possibilities for this are quite limited: we can only generate a random input, and certain boundary cases such as the case where the values of all variables are minimized or maximized.

We implemented the STS from Figure 6 in the STSimulator library. A technical report of this implementation can be found in Appendix A. We used the library to generate test cases as we sketched above.

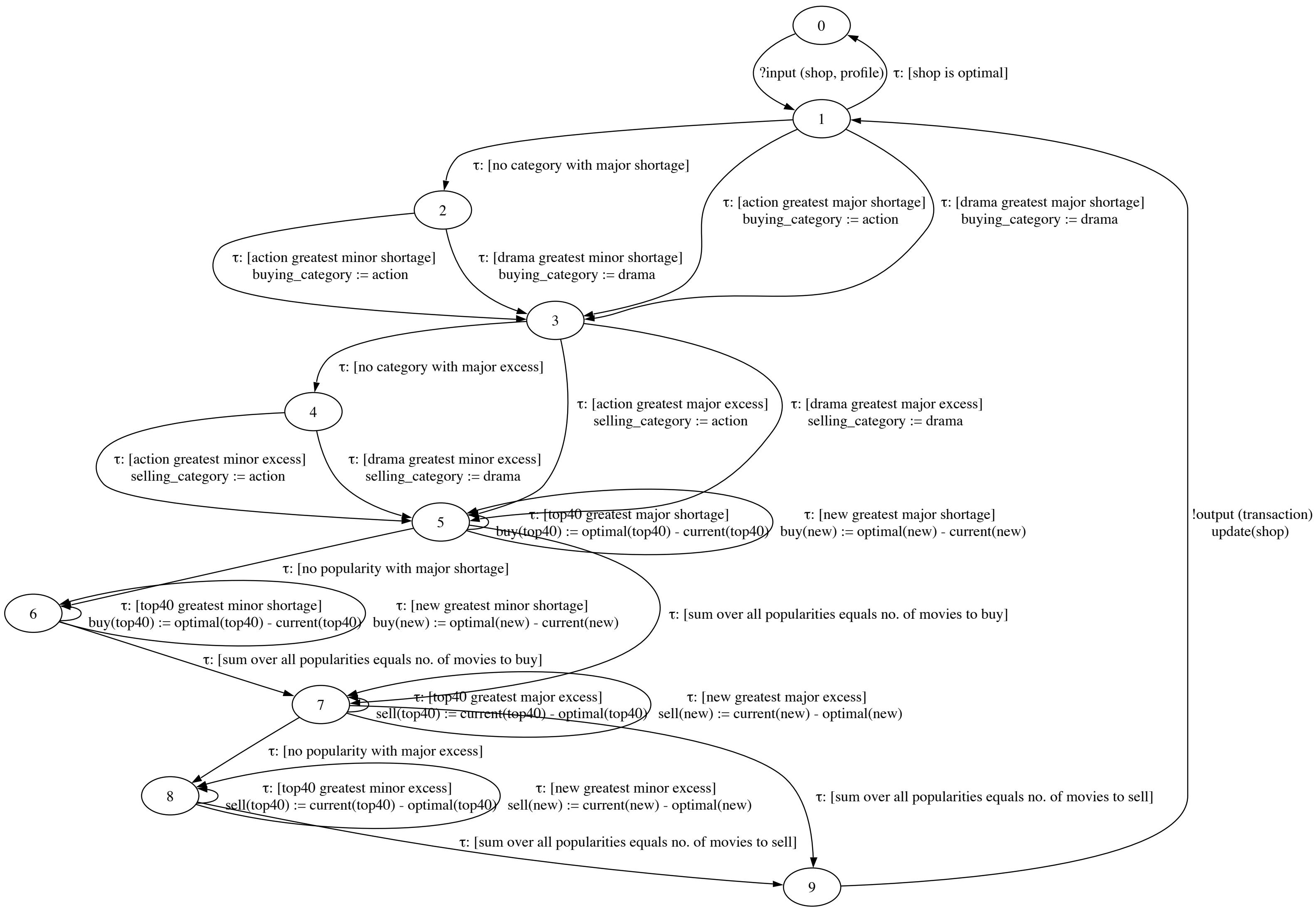
Modeling the specification with an STS gives us the possibility of defining a coverage criterion. Every test case takes a certain path through the STS and we could say for example that we want to cover all locations, or all transitions. With the STSimulator library we have the possibility of computing the locations or transitions that have been covered by a set of test cases.

Ideally, we would like to generate the input for our test cases based on a coverage criterion. For example, we could ask for a set of test cases that gives coverage of all locations in the STS. This is not currently implemented in the STSimulator library. The second part of this thesis will concentrate on the question of how to do this: how to generate a set of test cases for an implementation, based on an STS of the specification and some coverage criterion.

4.6 Comparison

We saw two different approaches to generating test cases for the Blockbuster case, based on its specification. The first approach, implementing the restrictions of the specification with Constraint Logic Programming, gives a test oracle and a way to generate the input of test cases based on user-defined criteria. These criteria can be defined in many different ways, dependent on the exact implementation of the restrictions. The second approach, using an STS to model the specification and implementing this using the STSimulator library, gives us a test oracle, a way to generate random test cases, and a way to define coverage

[3]`http://www.cs.ru.nl/~lf/tools/stsimulator/`
[4]`http://www.cs.ru.nl/~lf/`



0
?input (shop, profile)
τ: [shop is optimal]
1
τ: [no category with major shortage]
τ: [action greatest major shortage] buying_category := action
τ: [drama greatest major shortage] buying_category := drama
2
τ: [action greatest minor shortage] buying_category := action
τ: [drama greatest minor shortage] buying_category := drama
3
τ: [no category with major excess]
τ: [action greatest major excess] selling_category := action
τ: [drama greatest major excess] selling_category := drama
4
τ: [action greatest minor excess] selling_category := action
τ: [drama greatest minor excess] selling_category := drama
5
τ: [top40 greatest major shortage] buy(top40) := optimal(top40) - current(top40)
τ: [new greatest major shortage] buy(new) := optimal(new) - current(new)
!output (transaction) update(shop)
τ: [no popularity with major shortage]
6
τ: [top40 greatest minor shortage] buy(top40) := optimal(top40) - current(top40)
τ: [new greatest minor shortage] buy(new) := optimal(new) - current(new)
τ: [sum over all popularities equals no. of movies to buy]
τ: [sum over all popularities equals no. of movies to buy]
7
τ: [top40 greatest major excess] sell(top40) := current(top40) - optimal(top40)
τ: [new greatest major excess] sell(new) := current(new) - optimal(new)
τ: [no popularity with major excess]
8
τ: [top40 greatest minor excess] sell(top40) := current(top40) - optimal(top40)
τ: [new greatest minor excess] sell(new) := current(new) - optimal(new)
τ: [sum over all popularities equals no. of movies to sell]
τ: [sum over all popularities equals no. of movies to sell]
9

Figure 6: The Symbolic Transition System for the Blockbuster case

criteria. We can use these coverage criteria to see how well our random test cases cover the specification. In the future, it may be possible to extend the library so we can generate a set of test cases to cover the entire specification.

Both approaches are very interesting. As it is now, the CLP approach gives more control over which test cases we choose because we can set the input of the test case to match some user-defined restrictions. However, this does mean we have to actually define these restrictions. If we could extend the STS approach to include generating test cases based on a coverage criterion, the whole process could be automated. This does raise the question of which coverage criterion to choose. There are several well-known coverage criteria for similar problems, but it is very hard to say which one is best and there is no “perfect” criterion that rules out all errors in the implementation.

In the remainder of this thesis, we will investigate how we can use constraint solving to generate an optimal set of test cases based on an STS and some coverage criterion.

5 Coverage Criteria

While testing, we want to know when we have executed enough test cases to rule out (at least a big part of) all errors in the implementation. Since we do black-box testing we have no access to the implementation and all information we can get about the implementation is its reaction to a certain input. Theoretically, this means that the only way to rule out all errors is to try every possible input, which will often take an infinite amount of time. But even though we can never be sure to rule out all errors, we want to have some indication of which part of the algorithm we covered.

We already saw that STSs can be very similar to control flow graphs. Just as every run of a test case takes a path through the STS, every run of a test case also takes a path through the CFG of the implementation. In the context of white-box testing, many coverage criteria have been defined that are based on CFGs. We can easily apply these coverage criteria to STSs to get a notion of coverage of an STS. We discuss the most commonly used criteria.

5.1 Statement Coverage

In a CFG, every node represents a block of non-conditional statements. Statement coverage measures the fraction of these nodes that was covered by a certain test run. If a set of test cases gives 100% statement coverage on a CFG, it means every statement of the implementation was executed at least once in the run of one of the test cases. This does not necessarily mean we ruled out all errors, because we can execute a statement with a lot of different values for all variables used in the implementation, and some errors might only occur when the variables have certain values.

If we apply this to an STS we get location coverage. This is the fraction of locations that were covered by a certain set of test cases. It is less straightforward to say what part of the specification or implementation a location represents as this depends greatly on the design choices made while making the STS. Just as with CFGs, there are often infinitely many ways to reach a certain location because of the different values the variables may take. Still location coverage gives a good basic measure of how big a part of the STS we covered.

5.2 Decision Coverage

In a CFG, every conditional statement is represented by a branching node. Decision coverage measures the fraction of outcomes of the conditional statements that were covered by a set of test cases. So if we have a decision coverage of 100%, every conditional statement evaluated true and false at least once while executing the test cases. This means that every branch in the CFG has been visited at least once.

We could easily apply this to an STS by measuring the fraction of transitions that have been visited by the test cases. This is a stricter measure than state or location coverage: it might occur that a set of test cases gives 100% location coverage but less than 100% transition coverage, because there may be multiple transitions leading to a certain location.

5.3 Condition Coverage

Condition coverage of a CFG is about the conditions used in the conditional statements of the implementation. These conditions often consist of a number of atomic conditions, joined by disjunctions and conjunctions. Condition coverage is 100% when every atomic condition has evaluated both true and false once while executing the test cases. We can apply it to STSs by measuring the fraction of atomic formulas in the transition guards that evaluated true and false. This is different from decision coverage. For example, if a certain transition guard is of the form $\varphi = \varphi_1 \vee \varphi_2$, for full decision coverage it would be enough to have one test case that makes both φ_1 and φ_2 false and so φ false, and one test case that makes φ_1 true and φ_2 false and so φ true. For full condition coverage, we would also need a test case that makes φ_1 false. Condition coverage is not necessarily stricter. In the above example, if we have a test case that makes φ_1 true and φ_2 false and one that makes φ_1 false and φ_2 true, we have full condition coverage. However, we do not have full decision coverage because we have no test case that makes φ false.

There are many variations to condition coverage, for example coverage measures where every possible combination of truth values of atomic formulas have to occur, or combinations of decision and condition coverage.

5.4 Path Coverage

An even stricter coverage criterion is path coverage. Here we measure the fraction of all possible paths through the CFG that were taken by the test cases. We can apply this to STSs by taking the fraction of possible paths through the STS that was covered by the test cases. This is an even stricter measure than decision coverage but it is quite impractical: if there are loops in the STS there will be infinitely many possible paths through the STS.

5.5 Symbolic State Coverage

Except for these CFG-based coverage criteria another criteria has been suggested in [8], that is designed specifically for STSs. In symbolic state coverage, we consider each combination of a location with the transition guards that were met and the variable substitutions that were applied before reaching this location from the initial location. Symbolic state coverage is the fraction of these tuples that have been covered by the test cases. In a way, this is very similar to path coverage: we look at the possible ways to reach a certain location. Just as with path coverage, we get a problem with loops because we can make an infinite path through the STS, and the transition guards and variable substitutions also become infinitely many. This is solved in [8] by only looking at test cases of a certain fixed length.

5.6 Our Choice: Location Coverage

We decided to pick location coverage of the STS as our goal in generating test cases. Location coverage is used often in similar literature on white-box testing and it gives a pretty good measure of the fraction of the specification that was covered. As we mentioned above, there are stricter criteria, for example transition coverage. As we see in the Blockbuster case, it is very well possible that there are multiple transitions leading to one location and in this case, location coverage is in some sense incomplete. However, if the STS was designed in a sensible way, the locations will probably already say a lot about the different cases of the specification. We also see this in the Blockbuster case: there is usually a different transition for each category, but there is a different location for the case that there is no major excess. This way, the locations give a representation of higher-level case distinctions in the specification. There is a good chance that the implementation code for the different categories will be identical, so it is probably sufficient to test only one of them. However, it is highly interesting to test a case where there is no major shortage as well as a case where there is a major shortage.

Additionally, a stricter measure also gives more test cases, and because running test cases usually takes a lot of time we want to keep the set of test cases as small as possible. For this reason we want to choose a criterion that does not give a superfluous amount of test cases.

6 Related Work: White-box automated testing

In white-box automated testing, we assume we have access to the code of the implementation, and use the details of the code to generate test cases. For example, we look at the exact conditions of conditional statements and loops. Often the CFG of the implementation, or a similar graph or tree structure, plays a large role in these testing techniques. We looked at a number of existing white-box testing techniques because we can apply certain concepts from them to STSs.

6.1 EFFIGY

The first technique we considered was presented in [10] and implemented in the EFFIGY system. In this approach the implementation under test is executed on a symbolic input instead of a concrete one. This means that all variables get a value in terms of the symbolic input, for example $n = 5x + 2y$. At every conditional statement, the system tries to predict which branch the implementation will take. If this is not possible because some variable only has a symbolic value and not a concrete one, the user is asked to choose a branch, and the condition that the variables should meet in order to take that branch is remembered as the path condition. When the system encounters more conditional statements, their conditions get added to the path condition. This way, any concrete input that satisfies the path condition will follow the same path. All paths that were explored this way are stored in a tree together with their path conditions.

EFFIGY can be used during testing to find out which conditions the input should meet in order to take a certain path through the program. Also, it can be used to explore the relations between output and input variables, because we can see the value of the output in terms of the symbolic input. We can also manually add certain predicates to the path condition to see if they lead to a contradiction. This way we can check whether certain predicates are satisfied by every execution that follows a certain path. These predicates could for instance be some correctness condition.

6.2 AGATHA

The second approach is presented in [6] and implemented in the AGATHA toolset. It does symbolic execution very much like EFFIGY, only now applied to Extended Input Output Labeled Transition Systems which are extracted from UML models. Except for symbolic execution and the computation of a path condition the system also does the computation of concrete values that satisfy the path condition. A constraint solver is used for this. Since constraint solvers only have limited power and are not always able to find a solution, especially not when the constraints (path condition) are very large and complicated, rewriting rules are used to simplify the symbolic representation of variables and the path condition. Because a concrete value is computed for every path, after each symbolic execution we have a concrete test case which can be executed in order to test that path.

6.3 DART

The third technique we considered was presented in [9] and is called Directed Automated Random Testing (DART). It starts by executing the implementation on a random input. During execution a parallel symbolic execution is done: symbolic values for all variables and a path condition are maintained. After the first random execution, the path condition is considered and the condition of the last encountered conditional statement is negated in order to get the path condition for a different path. Then new input is generated using a constraint solver that satisfies this new path condition and by executing the implementation with this new input, it is driven down a new path. Then the next condition is negated and this process continues until all paths have been exercised.

This is not always possible, for example when the constraint solver is not strong enough to find concrete values that satisfy a certain path condition. In this case, DART tries to instantiate some variables with the concrete values from the previous execution. However, this could be a problem because satisfying the path condition might only be possible with certain concrete values which might be different from the ones of the previous execution.

6.4 Pex

Another approach we considered was presented in [11] and is called Pex. It is based on the same approach as DART only designed to cover a lot of statements of the implementation in a short amount of time, rather than doing a complete search for covering all statements. The possible paths through the implementation are divided into equivalence classes based on the structure of the implementation, and after every execution Pex picks a path from the least often chosen class. If a certain path condition cannot be solved by the constraint solver, Pex just skips the path in order to save time.

6.5 BLAST

The last technique we considered was presented in [5] and implemented in the BLAST software model checker. In this approach a tree of possible paths through the CFG is built, where repeating states are left out in order to handle loops and to keep the tree feasibly small. The main goal is to show that a certain erroneous state is reachable, because this will show that the implementation contains an error. For this purpose an abstract overapproximation of the path condition is computed, which is then specificied as far as is needed to show the reachability of that particular state. Then a constraint solver is used to solve this path condition and generate inputs that reach the error. BLAST is limited to linear arithmetic, which means that the computations done in the algorithms tested by BLAST can only consist of additions and multiplications with constants. In this limited arithmetic the constraint solver will always succeed in finding such a solution.

BLAST can also be used to show that certain predicates (possibly asserting correctness of the implementation) are true during execution, and to cover all locations that satisfy a certain predicate. This is done by invoking BLAST for every location in reversed depth-first order.

7 The Problem: Symbolic Reachability with Non-Determinism and a Limited Constraint Solver

As we stated before, our goal is to generate a set of test cases from a STS and a coverage criterion. We will now state this problem more precisely.

There is one aspect of STSs we haven't discussed yet: they can also model the specifications of non-deterministic algorithms. For example, an implementation that could give two different outputs on a certain input, both of which are correct. These "non-deterministic" choices are in practice often determined by some detail that was not included in the specification or in the STS, which makes them seem non-deterministic while they are not. In any case, we should handle them as if they are non-deterministic and because of this the STS becomes non-deterministic also. This means that in one location with the same input or output message, there are multiple possible successor locations.

In Section 1 we stated that a test case consists of the input values for the implementation and the corresponding output values. In the case of non-determinism we should refine this notion, because there might be multiple correct output values. This means that a test case is a collection of values for all input variables and a set of possible values for all output variables. In the end, if the output values are in this set then they are correct.

Every test case follows a path through the STS, based on the input and output values it consists of. This path is the part of the specification that is relevant to that particular input and output. Because of non-determinism, one test case might follow multiple paths through the STS. What the STS actually expresses, is that if the input and output corresponds to one or more paths through the STS, then the input and output is correct according to the specifications.

The exact paths through the STS that are followed by certain input and output depend on which transition guards are satisfied by the location variables and interaction variables. The interaction variables are directly determined by the input and output, the location variables depend on them and on each other through the update mappings. Each path is a sequence of transitions of the STS. It starts at the initial node, and after that we follow each transition of which the transition guard is satisfied, applying its update mapping to the location variables. In the end, a test case follows a certain path if it satisfies all transition guards on the path, each with the update mappings of the previous transitions applied to them.

We formalize this concept by defining a generalized transition relation. In this generalized relation we take a sequence of transitions. There may be multiple transitions with the same gate in this sequence and to distinguish between the interaction variables of the different transitions, we define sets of history variables $\mathcal{I}_1, \mathcal{I}_2, ...$ which are disjoint from each other and from the set Var of location and interaction variables. We define $\widehat{\mathcal{I}} = \bigcup_j \mathcal{I}_j$ and $\widehat{Var} = \mathcal{V} \cup \widehat{\mathcal{I}}$. The purpose of these variables is, that the interaction variables that are passed in the first transition of a path are taken from $\mathcal{I}_1$, those in the second transition from $\mathcal{I}_2$, etc. In addition, we assume to have bijective variable-renamings $r_n : \mathcal{I} \mapsto \mathcal{I}_n$ that map the interaction variables to those in set $\mathcal{I}_n$.

The formal definition of the generalized transition relation, which is straight from [8], is in Definition 14.

Definition 14. *The generalized transition relation* $\Longrightarrow \subseteq L \times \Lambda^* \times \mathfrak{F}(\widehat{Var}) \times \mathfrak{T}(\widehat{Var})^{\mathcal{V}} \times L$ *is defined as the smallest relation satisfying:*

(Sϵ) $l \xRightarrow{\epsilon,\ \top,\ \mathsf{id}} l,$

(Sτ) $l \xRightarrow{\sigma,\ \varphi\wedge\psi[\rho],\ \rho\circ\pi} l'$ *if* $l \xRightarrow{\sigma,\ \varphi,\ \rho} l''$ *and* $l'' \xrightarrow{\tau,\ \psi,\ \pi} l'$,

(Sλ) $l \xRightarrow{\sigma\cdot\lambda,\ \varphi\wedge(\psi[r_n])[\rho],\ \rho\circ r_n\circ\pi} l'$ *if* $l \xRightarrow{\sigma,\ \varphi,\ \rho} l''$ *and* $l'' \xrightarrow{\lambda,\psi,\pi} l'$ *and* $n = length(\sigma)+1$.

If $l \xRightarrow{\sigma,\ \varphi,\ \rho} l'$ then from location l we can reach location l' with a sequence of input and output messages of the types given in sequence σ that satisfy the conjunction of transition guards collected in φ, with all location variables updated according to substitution ρ. This means that if $l_0 \xRightarrow{\sigma,\ \varphi,\ \rho} l$ then any test case whose input and output is such that the interaction and location variables satisfy φ might follow the path σ from the initial location l_0 to location l. φ is called the *attainment constraint*, and is the equivalent for STSs of what is called a path condition in the literature on white-box testing.

Our goal is to find a set of test cases that give the highest possible location coverage of the STS. We can split this problem up in finding a set of paths that give the highest possible coverage, and for each path, computing the attainment constraint and then finding input and output values that satisfy the attainment constraint. For this task we could use a constraint solver.

This works out fine for input values, because we can choose whatever input value we want to give to the implementation. However, there is a problem here with output values. As we discussed above, one test case might have multiple output values for each output variable. We cannot know in advance which one of these output values the implementation will choose and show to the user. Suppose we want to follow a certain path which includes output value x. And suppose output value y would also be correct, only part of a different path. Then if the implementation gives output value y we will not have found an error because y is correct. But we will also not have covered our target path, because we need output x for that. We will come back to this problem later.

Constraint solvers only have limited power, and long attainment constraints will sometimes be very hard to solve. Also, a constraint solver will take more time as the size of the attainment constraint increases. For this reason, we should try to minimize the number of attainment constraints we need to solve, and we should try to simplify them as much as possible.

Now we can state our problem as follows: to find attainment constraints for a set of paths that give location coverage of the STS; and to find test cases satisfying those attainment constraints.

8 Automatic Test Generation with Symbolic Transition Systems

We combined some features from the techniques in Section 6 with basic search techniques and some novel ideas to come to an algorithm that generates a set of test cases that attempt to achieve the highest possible state coverage of an STS.

8.1 Random Paths and On The Fly Test Case Generation

Computing input and output values that satisfy a certain attainment constraint using a constraint solver is computationally very expensive. If we start out with random values, just like in the DART approach, we already get one solution to an attainment constraint for free: the random test case we generated is a solution to the attainment constraint of the path it takes.

However, if we take a random test case, there is a chance that we won't follow a long path through the STS, because we quickly get to a transition guard that is not satisfied by the random values we generated. This way, we won't cover many states. Because of this we decided to choose a random path through the STS and generate values satisfying the transition guards on the fly.

We start at the initial node and choose a random transition from there. Then we try to generate values for all interaction variables of that transition such that the transition guard is satisfied. If there are multiple possibilities for these values, we pick a random one. If we can find no values at all, we try a different transition. Then we do the same for the next node and we keep on going on like this until we get stuck, that is, until we can find no transition of which we can satisfy the transition guard.

This works out fine for generating the values of input variables. For output variables, things are a little different because we can always decide what input we give to the implementation, but we don't know in advance what output the implementation will return. Because of this, we don't generate a random value for the output variables, but we compute the set of possible solutions and ask the user to choose one of them. The purpose of this is, that the user feeds the randomly generated input to the implementation, and reports to the system the output that the implementation returns. If the implementation does not return any of the output in the set of possible solutions and the user does not give a correct value, then we try a different transition to continue our random path.

This way, generating random solutions for input values and asking the user to choose from a set of solutions for output values, we find a random path of satisfiable transition guards through the STS. The advantage of this is that we don't have to solve a long attainment constraint, only relatively short transition guards, and if we can't find a solution we can simply try a different transition.

Note that because of non-determinism, there might be other transitions than the one for which we generated solutions, that are also followed by the test case we generate. As with every test case, we get a set of paths that are followed by the test case, that at least contains the path for which we generated the solutions.

Also, we have to take care that the random path does not continue infinitely. This might happen if the STS contains loops. If there are some small loops, there will often be no problem because the algorithm might randomly take the loop a few times, but will probably at some point randomly choose to continue with a different transition. It might even be necessary to take the loop a few times, for example if a certain counter is increased in the loop and we can only continue with a different transition if this counter has a certain value. But there are STSs where this might be a serious problem, for example a circular STS that is one big loop. In such cases, the algorithm might go on infinitely searching for a random path. We can easily solve this problem by letting the user define some termination criterion. For example, every time we visit a location that we have visited before, we could ask the user whether we should continue or terminate.

8.2 Depth-First Search and Non-Determinism

After the first test case we generated this way, one or more paths of the STS and all states on those paths will be covered. Next, we want to cover all other states. An approach that is used in DART is as follows. Take the path the random test case took, go back to the last branch that has an uncovered child, and compute the attainment constraint for that other child. Then generate a test case that satisfies this new constraint and run it. This test case will take the other child and continue its path from there. At the end of this path go back again to the last branch that has an uncovered child, etc. This is a bit like depth-first search and this way, eventually all nodes will be covered. An additional advantage is that the first part of the attainment constraint for a new uncovered child will be the same as the attainment constraint for the previous test case, because the first part of the path from the initial node is the same. We might be able to use some information from the previous test case to find values for the new attainment constraint.

However, this approach does not work in our case because the STS might be non-deterministic. Consider the STS in Figure 7. Assume the first random test case non-deterministically took two paths: the path 0 - 1 - 3 and the path 0 - 2 - 5. Then we take the path 0 - 2 - 5 and check each of the nodes for uncovered children. 5 has no uncovered children so we continue to 2. 2 also has no uncovered children, so we return to 0. 0 also has no uncovered children because 1 was already covered by the first random test case, so the system assumes all nodes are covered and terminates. However, node 4 was never visited and is neglected in this approach.

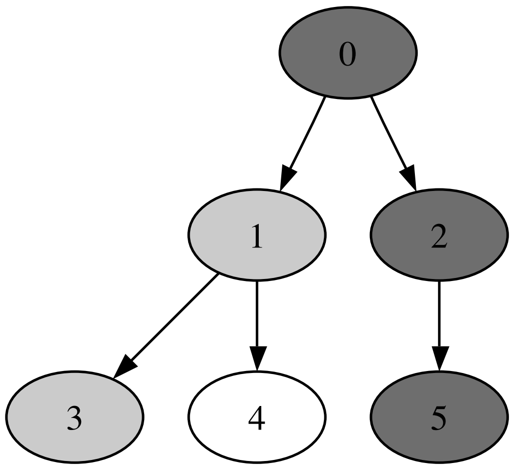


Figure 7: An example where the depth-first approach fails

One might argue that it would be possible to use this approach as long as we check the branches of all paths that were non-deterministically followed by the test case. This means we have to keep some kind of set of branches we still have to check, because if we find a branch with an uncovered child and we run a test case that covers that child, we will have to check all branches of the first run and those of the second run. If we then find another uncovered child we will also have to check all branches of that third run, etcetera. We thought this would give a lot of overhead, so we didn't choose this approach.

We decided on a different approach that can handle non-determinism. We start out with a collection of paths the previous test case took. For every node on one of the paths that has children that are not part of one of our paths already, we compute an approach of the distance of each child to the closest uncovered node. Then we choose the node on the path for which this distance is minimal, and continue to search for a test case which follows a path to the child with minimal distance.

We compute this distance to the closest uncovered node with a limited breadth-first search. We search all children of the node of which we want to compute the distance, and if none are uncovered then all grandchildren, all great-grandchildren, etcetera. At every node we check whether we have already visited it before in this search, and if so, we skip its descendants. We do this unto a limited depth. The first search is limited with the number of nodes in the STS. Because we search for a minimal distance, every search after that is limited by the smallest distance we already found. So if we already found a node with distance 5 to an uncovered node, we only search until depth 5 for every following node.

In the example of Figure 7, we search all children of nodes on the paths that are not themselves a part of one of the paths. There is only one of such children, namely node 4, and it has a distance to the closest uncovered node of 0 because it is uncovered itself.

Once we have found the child with minimal distance to an uncovered node, we want to force the next test case to follow a path to that uncovered node. For this purpose we remember already during the search the path to each potentially uncovered node from the node for which we compute the distance. This way, once we have found the node with minimal distance to an uncovered node, we know how to get to the uncovered node and we can try to force the path of the next test case there.

8.3 Constraint Propagation

Now we have a collection of paths the previous test case followed, a node on one of these paths, and a path from a child of that node to an uncovered node. Next, we want to find a test case that follows the path to this node, then to its child, and then to the uncovered node. We could compute the attainment constraint for this path and then use a constraint solver to find a solution to this constraint. However, to make the task of the constraint solver easier, it would be better if we could simplify the attainment constraint or if we would only have to solve a part of it.

Just as in the DART approach, we have the advantage that a part of the new path we want to follow is identical to (one of) the path(s) the previous test case followed. The satisfaction of the attainment constraint depends on the values of all interaction variables on the path. Because a part of the path is identical, we could choose the values of the interaction variables on the identical part the same as in the previous test case. Then we are sure to follow at least the identical part of the path.

For example, look at the STS in Figure 8. This is an STS of a coffee machine that first expects a coin to be inserted. If the value of the coin is below 50 cents it gives tea, otherwise the user should press one of two buttons. If button 1 is pressed the machine gives coffee, if button 2 is pressed the machine gives hot chocolate.

Suppose the first, random test case was one where a coin of 50 cents was inserted, button 1 was pressed and coffee was expected from the machine. This test case follows the grey path 0 - 1 - 3 - 6. Next, we want to take the path 0 - 1 - 4 to uncovered node 4. The attainment constraint for this path is $coin \geq 50 \wedge button = 2$. We could give this to a constraint solver and because this is a fairly simple constraint we would probably get a solution, namely something like $coin \mapsto 50, button \mapsto 2$.

But suppose our constraint solver was not powerful enough to solve this constraint, as is often the case in more complex systems. Then we could simplify the constraint by using information from the test case that took the grey path. The paths are identical up to node 1, so we give all interaction variables on the path up to node 1 the same value as they have in node 1 in the path followed by the previous test case. This means $coin \mapsto 50$ and this way we end up in node 1. Then the grey path continues to node 3 while we want to go to node 4 with our new test case. So we try to find a solution for the guard of the transition from node 1 to node 4, which is $button = 2$. This constraint is a lot easier to solve for the constraint solver, and it will surely find the solution $button = 2$, and this way we end up in node 4 as we intended.

However, this doesn't always work out as easy as in the previous example. Consider the

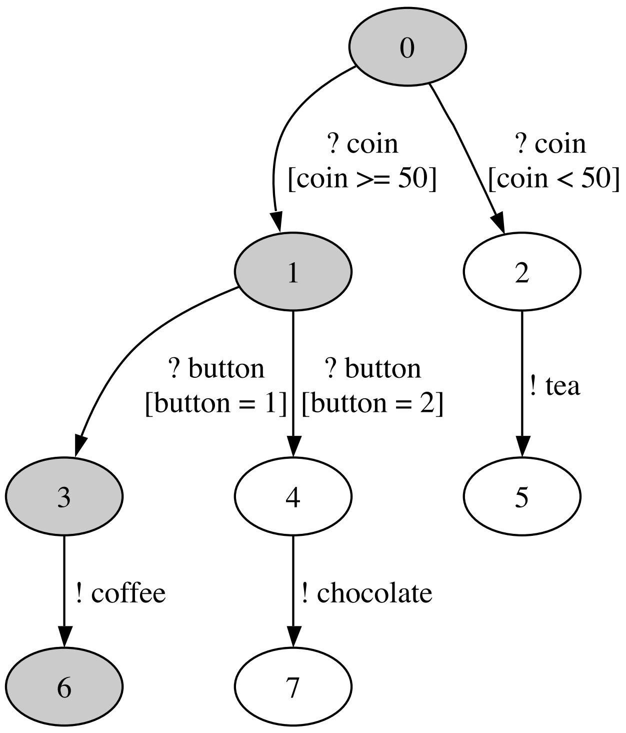


Figure 8: An example STS where we can simplify the attainment constraint very easily

STS in Figure 9, which also models the specification of a coffee machine. It expects two coins to be inserted, of which the first one should have a value of at least 20, and the second one at least 10. Then if the sum of the coins is greater than 80, the machine gives a cup of coffee, and otherwise it gives a cup of tea.

Suppose the first random test case was one where the first coin is one of 50 cents, the second coin is one of 20 cents, and then the machine is expected to give a cup of tea. This test case takes the grey path 0 - 1 - 2 - 3, and now we want to take a path to the uncovered node 4. The entire attainment constraint for the path 0 - 1 - 2 - 4 is $coin_1 > 20 \wedge coin_2 > 10 \wedge coin_1 + coin_2 > 50$. Note that we composed this attainment constraint using Definition 14. We used a renaming of the interaction variables *coin* of the transition from node 0 to 1 and from node 1 to 2 to $coin_1$ and $coin_2$ in order to distinguish between them, and we applied the substitutions $val := coin_1$ and $val := val + coin_2$ to get the formula $coin_1 + coin_2 > 50$ out of the transition guard $val > 50$ of the transition from 2 to 4.

The identical part of the old and the new part is 0 - 1 - 2. We try to simplify the attainment constraint by choosing the values of all interaction variables of this identical part the same as in the previous test case. This means $coin_1 \mapsto 50, coin_2 \mapsto 20$ and this also fixes the value of the location variable $val \mapsto 70$. Now we want to take the transition to node 4. This means we want to choose the interaction variables of the transition such that the transition guard is satisfied. The interaction variables are just the output variable *coffee*, and the transition guard is $val > 80$. Since *val* has the value 70 and it does not get updated in the transition to node 4, the constraint we need to solve is $70 > 80$. Of course we can never choose *coffee* such that $70 > 80$, and if we use a constraint solver to solve this transition guard we will find out it has no solution.

If the transition guard cannot be solved, this may very well be because we fixed the value of an interaction variable somewhere higher up the path that influences the location variables that are used in the transition guard. In this case, we fixed the values of $coin_1$ and $coin_2$ to 50 and 20 which results in that *val* has value 70. So in this case we should not fix

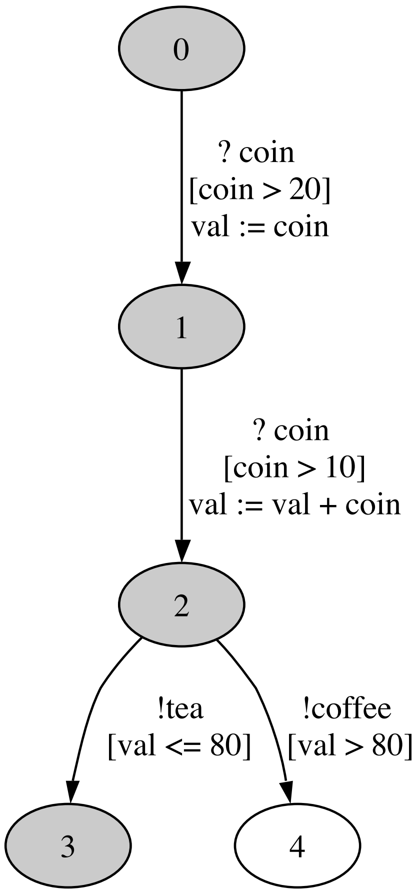


Figure 9: An example STS where we need to apply constraint propagation

all interaction variables along the identical part of the path to the values they had in the previous test case. However, we might still use information from the previous test case by fixing only a part of the interaction variables.

We do this as follows. We go just one node up on the path, and fix only the interaction variables up to there. In this case, we would go up to node 1 and fix all interaction variables up to there, that is, we would fix $coin_1 \mapsto 50, val \mapsto 50$. Now we want to follow the path through node 2 to node 4. This means we have to satisfy the guard of the transition to node 2 and of the transition to node 4. The constraint we need to solve is $coin_2 > 10 \wedge 50 + coin_2 > 80$. Again, we built this constraint using a variable renaming of *coin* to $coin_2$ and a substitution of *val* to $val + coin_2$ and to $50 + coin_2$ because *val* has value 50 in node 1.

Now if we give this constraint $coin_2 > 10 \wedge 50 + coin_2 > 80$ to a constraint solver, it will easily find a solution, for example $coin_2 = 50$. Then we can take the transitions to node 2 and 4 and we end up at the uncovered node we intended.

In this approach we “propagate” the constraint up the path the previous test case took: first we try to solve the constraint from the node at the end of the identical part of the path, and if that’s not possible we go up and keep on adding transition guards to the constraints until we either find a solution or we are back at the initial node. If we are back at the initial node and we can still not find a solution for the constraint, we will have calculated the entire attainment constraint and we know the state is unreachable by that particular path.

8.4 Non-Determinism and Simultaneous Constraint Propagation

There is one aspect of our approach we didn't discuss yet. Because of non-determinism, there might be multiple paths to the same node. Suppose the previous test case took two different paths which both lead to a certain node with an uncovered child. Next, we want to reach that uncovered child. Then we can try both paths to get to that node and then to the uncovered child.

Because of this, we keep a set of all sub-paths the previous test case followed that end in the ancestor of the node we want to reach. These paths may have different lengths in case of loops etc. Each of these paths has its own interaction variables up to the ancestor node, and its own valuations for these interaction variables. We try out each of these valuations and try to solve the constraints to get to the uncovered node. If this is not possible for each of these path, we propagate the constraint up to the previous node on each path. This might be a different node for each path, and because of this also a different constraint.

We give an example of how our approach works on the STS in Figure 10. This is an STS where the user first gives input x. The STS then non-deterministically either stores x or $x + 10$ in n. Then the user gives input y, which is stored in m. Then if $m < 10$ the implementation should output m and terminate, and otherwise the implementation should output m only if $n \geq 10$.

We start out with the random test case that gives input $x = 5$, $y = 2$ and expects output 2. This test case non-deterministically takes either the path 0 - 1 - 3 - 4 or 0 - 2 - 3 - 4. In the first case the values of the location variables will be $n \mapsto 5, m \mapsto 2$ and in the second case the values of the location variables will be $n \mapsto 15, m \mapsto 2$.

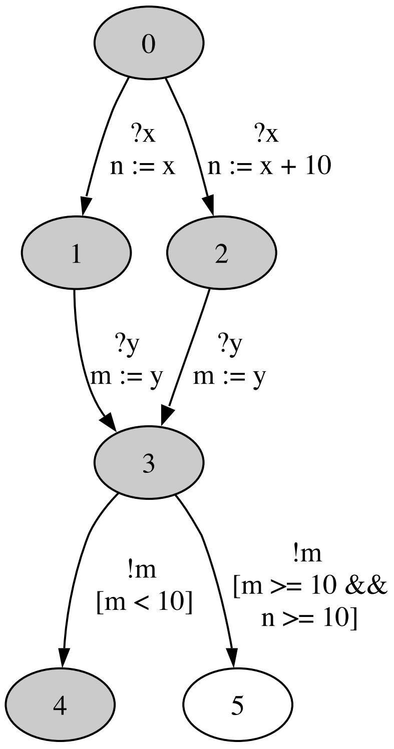


Figure 10: An example STS where we need to apply simultaneous constraint propagation

Now we want to follow a path to the uncovered node 5. This is a child of node 3. There are two paths the previous test case took to node 3: 0 - 1 - 3 and 0 - 2 - 3. In the first one the valuation of interaction and location variables is $x \mapsto 5, n \mapsto 5, y \mapsto 2, m \mapsto 2$ and in the second one it is $x \mapsto 5, n \mapsto 15, y \mapsto 2, m \mapsto 2$. We start out by simultaneously propagating up to the last node both paths have in common with the path to the uncovered child, namely node 3. We try to solve the constraint $m \geq 10 \wedge n \geq 10$ while applying either valuation. For the first valuation, with the substitution $m \mapsto 2, n \mapsto 15$, this becomes $2 \geq 10 \wedge 15 \geq 10$. For the second one it becomes $2 \geq 10 \wedge 5 \geq 10$. Clearly, both have no solution, so we need to propagate the constraint up further.

We do this simultaneously for both paths. For the first path we go up to node 1 and for the second path we go up to node 2. For the first path, we get valuation $x \mapsto 5, n \mapsto 5$ and we need to solve the constraint $y \geq 10 \wedge n \geq 10$, which becomes $y \geq 10 \wedge 5 \geq 10$ if we take into account the valuation $n \mapsto 5$. If we give this constraint to a constraint solver we immediately find out this constraint has no solution. This means that if we did not consider the other path we would have to propagate up further and compute solutions for an even larger constraint. Luckily, we can still try the other path. In the second path, we get valuation $x \mapsto 5, n \mapsto 15$. Again we need to solve the constraint $y \geq 10 \wedge n \geq 10$, which becomes $y \geq 10 \wedge 15 \geq 10$ if we take into account the valuation $n \mapsto 15$. Now if we use a constraint solver on this constraint we quickly get a solution $y \mapsto 15$ which solves the constraint and a test case with this value and the old value $x \mapsto 5$ brings us via node 3 to the uncovered node 5. This way we have covered the entire STS.

8.5 Unobservable messages

In Section 2.2 we introduced transitions with no input or output variables: unobservable messages. The purpose of these transitions is that a test case can at any time non-deterministically choose to take a transition with an unobservable message, as long as the transition guard is satisfied. So these unobservable messages are just another way in which non-determinism may occur, and they are handled by the algorithm just like any other non-deterministic choice of transitions.

8.6 Pseudocode

Detailed pseudocode of our algorithm is given in Appendix B. Here we give an abstract overview. The accompanying pseudocode is in Algorithm 2.

First we find a random path through the STS by searching for solutions of the guard of a random transition, as we described in Section 8.1. This way we generate a test case, and we compute the set of paths this test case may non-deterministically take. Then we enter a loop and we start searching for the node along one of the paths with minimal distance to an uncovered node, with a breadth-first search as we described in Section 8.2. Note that while computing the minimum we keep for every child of each node on the path an extra factor $tries$. This number denotes how often we already unsuccessfully tried to find a test case that follows a path to that child. If we are unsuccessful in finding a test case, this means we couldn't solve the attainment constraint which happens either if it is too hard for the constraint solver, or if there is no solution at all in which case taking the path is not possible. Keeping the number of tries prevents that we keep on trying the same path with no success.

When we have found the node n with minimal distance to an uncovered node, we remember the path from n to the uncovered node. Then we take all (sub)paths p that the previous test case followed that end in n. For each of them, we keep a path from the end of p to the uncovered node. This is the path for which we compute new interaction variable values. Initially this is for each of them the same, namely the path from n to the uncovered node. We also keep the attainment constraint $\Phi(p)$ for the path from the end of p to the uncovered node. This is the conjunction of all transition guards on the path.

Then we try for every path, to find a value for all interaction variables in $Path(p)$ that satisfy $\Phi(p)$. In the beginning we just try to find interaction variables that satisfy the transition guards on the path from n to the uncovered node. We do this for every path separately, because the values of location variables might differ between the paths. If we cannot find a solution, we propagate the constraint up one node by removing the last transition from p and

Algorithm 2 Algorithm for simultaneous constraint propagation

```
1: Find a random path through the STS and solutions for the transitions on the path
2: $P$ := the set of paths the generated test case took
3: loop
4:   $min$ := max_search_depth
5:   for all nodes $n$ on paths in $P$ do
6:     for all children $c$ of $n$ that do not occur in $P$ do
7:       do a breadth-first search for the closest uncovered node, limited until depth $min - 1$
8:       if $distance + tries(c) < min$ then
9:         $min$ := distance + tries(c)
10:        $minNode$ := $n$
11:        $minPath$ := path from $n$ to uncovered node
12:      end if
13:    end for
14:  end for
15:  if $min$ = max_search_depth then
16:    terminate
17:  end if
18:  $Q$ := the set of all (sub)paths in $P$ that lead from the initial state to $minNode$
19:  For every path $p$ in $Q$ keep a path $Path(p)$ from the end of $p$ to the uncovered node and
     an attainment constraint $\Phi(p)$ for that path
20:  For every $p$ in $Q$ initialise $Path(p)$ := $minPath$ and $\Phi(p)$ := the conjunction of transition
     guards in $minPath$
21:  for all paths $p$ in $Q$ do
22:    Compute values for the interaction variables of $Path(p)$ that satisfy $\Phi(p)$
23:    if we find a solution then
24:      Compose a new test case of $p$ and the found solution
25:      Run the new test case: report input to the user and let the user choose output
26:      P := the set of all paths the new test case took
27:      Break the for-loop
28:    else
29:      Remove the last transition from $p$ and add it to $Path(p)$
30:      Add the transition guard of that transition to $\Phi(p)$
31:      Continue the for-loop with the next path of $Q$
32:    end if
33:  end for
34:  if we are back at the initial node for every path and we can't find any solution then
35:    tries(c) += 1
36:  end if
37: end loop
```

adding it to the beginning of $Path(p)$. Also, we add the transition guard of this transition to $\Phi(p)$. Then we first try to find a solution on one of the other paths. If we can't find a solution on any of the paths, we try to find a solution to each of the longer constraints we propagated up. We keep on going on like this until we are back at the initial node for every path, or we have found a solution. In the first case we have failed to find a solution for a path to the uncovered node, we increase tries for the child of n that was part of the path to the uncovered node, and we start the loop over again. If we do find one or more solutions, we compose a test case of the previous test case up to the end of p and the solution, and run it. If there is more than one possible solution we choose a random solution in case of input transitions, or let the user choose a solution in case of output transitions. We keep track of all possible paths the new test case might take and save this in a new set of paths. Then we start the loop over again.

8.7 Completeness

We thought about a completeness proof of our algorithm. Of course it would be nice if we could prove that we will always cover the entire STS. However, we cannot guarantee that every state will be reachable, or that our constraint solver is strong enough to solve every constraint we would like to solve. Also, the way in which we search for paths to an uncovered node is not flawless. In our approach we compute the length of the shortest path to an uncovered node. However, this path may be impossible in the sense that there are no solutions for the attainment constraint of this path. There might be a different path from one of the nodes the previous test case covered to that same uncovered node, but we will only try to take the shortest path. We could imagine a case where all nodes except one are covered and the shortest path from the nodes covered by the last test case is impossible. Then we would keep on trying to take that shortest path until the number of tries equals the maximum search depth.

If it were not for this problem we might be able to prove that, given an unlimited constraint solver and an STS in which all states are reachable, we would cover all states. But this does not matter much because in practice the constraint solver will give much larger limitations than this one case where our search strategy prevents total completeness. Also, in practice this problem will almost never occur because we keep on trying different test cases and we will probably eventually find a way to get to the uncovered node from the path of one test case or another. Also, we maximize the chance that we do find a way to reach the node by trying to reach it from every path the test case non-deterministically followed.

8.8 Computational Complexity

We also tried to do a complexity analysis of our algorithm. However, we realized this would not make much sense, since the main part of the computational cost will be in computing solutions for the constraints. This is done by the constraint solver, which we see as an external factor. Also, in our algorithm we make a lot of effort in keeping the constraints we have to solve as small as possible. If we assume the time it costs to solve a constraint to be constant, there might be a much more efficient way to solve the problem. We could for instance for every path just compute the entire attainment constraint at once. But the whole point of the problem is that simple constraints will be solved faster than more complex ones, so this will probably be less efficient in practice. One might think about an analysis in terms of the complexity of the constraints that need to be solved, but there is really no way to decide which constraints are complex and which are simple, because some transition guards may be very complex or even have no solution, while others may be as simple as $\varphi = \top$. Because of this we concluded that a complexity analysis is not really possible. We do not think this is a problem, since the main part of the computational cost will be in using the constraint solver and not in the algorithm itself.

We haven't discussed any data structures our algorithm should use. For now we just talked about sets of paths, and in the pseudocode in Appendix B we use lists. However, to save computational cost, in the real implementation we might use trees to store the different paths.

8.9 Constraint Solvers

As we mentioned above, we treat the constraint solver we use as an external factor. Since solving the constraints will probably take the largest part of computational cost, the power of our constraint solver matters a lot. A lot of research has been done on constraint solving and there are several constraint solvers used in practice. We decided to leave the constraint solving algorithms out of our scope and our algorithm has been designed to work with any constraint solver.

In our algorithm we use a constraint solver to compute all possible solutions to a constraint, and then we pick one of them. There might be infinitely many solutions, so this sounds a bit ambitious. However, often constraint solvers give back one or more intervals of possible solutions, and then we can easily pick a value out of this interval. Also, if this is not possible with a certain constraint solver we can easily adapt the algorithm to ask the constraint solver for one solution and use that one. This gives the user less freedom in the case of an output variable because in our original approach the user could choose the output value it observed from the implementation, but this adapted algorithm will still be very useful in testing.

9 Demonstration

In this section we will demonstrate the use of our algorithm. We have a toy example of an STS of the specification of a very special calculator. This calculator successively takes two inputs, x and y. It non-deterministically chooses to either add or multiply these two values. However, for addition both values should be at least 1 and for multiplication both values should be at least 2. If these conditions are not met, the calculator does nothing.

If the calculator has accepted and either multiplied or added the input variables x and y, and the result is less than or equal to 2, then it gives this result as output. If y is exactly 5, the calculator gives the result times 2 as output.

The STS of the specification of this calculator is in Figure 11. State 0 is the initial state. Then we get an input message with the value of x and, if the conditions are met, we either go to state 1 for multiplication or to state 2 for addition. Then we get an input message with the value of y and if the conditions are met we save the result of the calculation in m and the value of y in n.

Then we are at state 3. Here we decide what output we will give. If $m \leq 2$, we output m. If n (which holds the value of y) equals 5, we take a silent step to state 5 and multiply m with 2. Then, we output m.

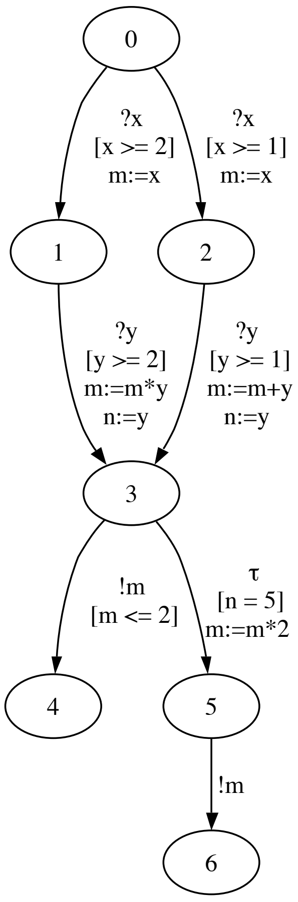


Figure 11: The specification STS of a very special calculator

We will show how our algorithm covers all states of this example STS, and how we find errors in the implementation of such a calculator. The control flow graph of a correct implementation is in Figure 12. This is a CFG of an implementation, and a whole different kind of model than the STS of the specification given in Figure 11, even though they look alike. For this purpose we thought of a number of erroneous implementations, usually called mutants in formal testing literature. The Control Flow Graphs of these implementations are in Figure 13.

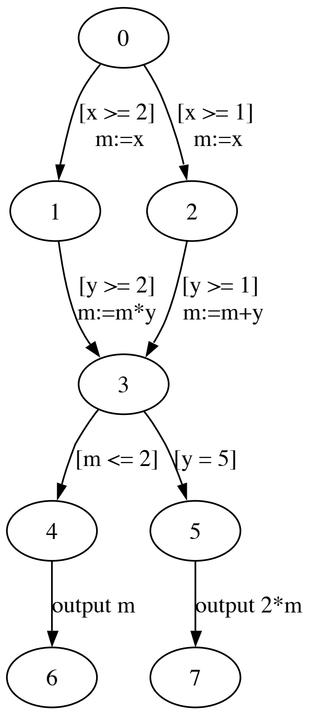


Figure 12: The CFG of a correct implementation of the specification

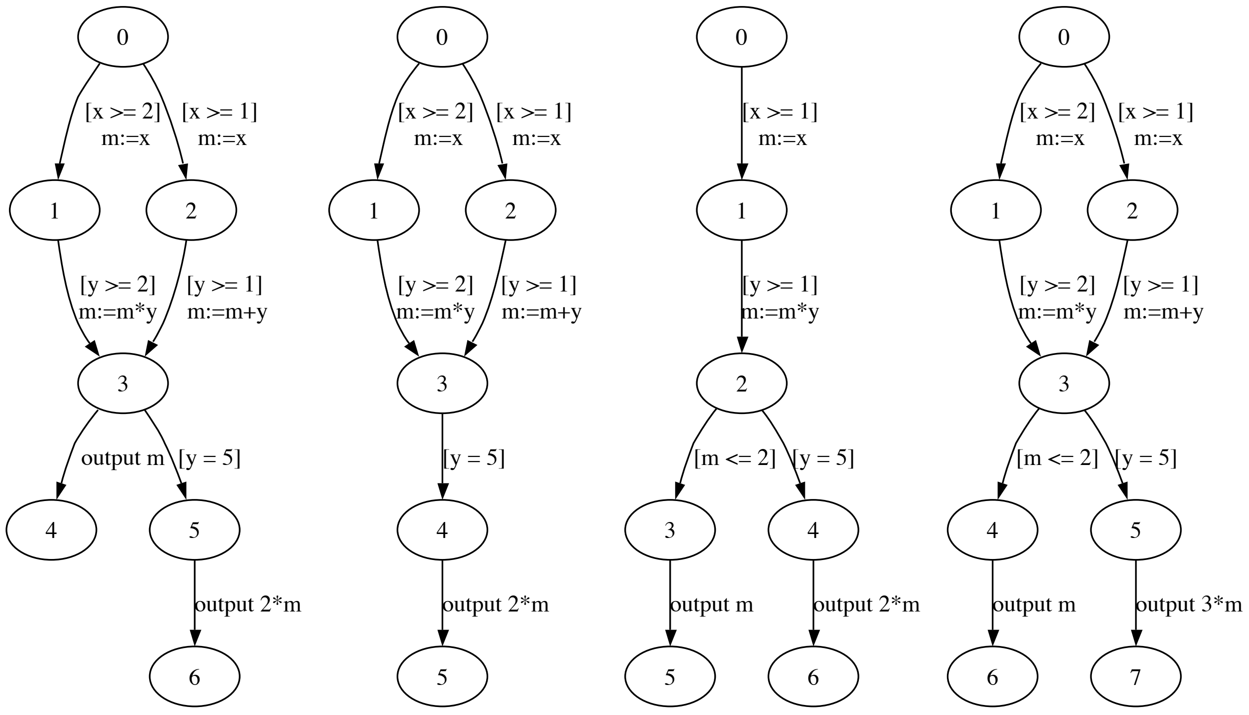


Figure 13: The CFGs of four faulty implementations of the specification

The first mutant does not have the $m \leq 2$ restriction and as a result always gives the result of the calculation as output, also if it is more than 2. The second implementation never gives the result as output, also not if it is less than or equal to 2. The third implementation always does multiplication and never addition. The fourth implementation gives an error in the case that $y = 5$: in this case it returns three times the result of the calculation, instead of two times.

We will show how our algorithm covers all states of the specification, and finds the errors in the faulty implementations this way. We start with searching for a random path, starting at the initial location. We choose either the switch to state 1 or the one to state 2, and try to find an input value for x that satisfies the switch restriction. In both cases there are many, so we choose a random one. For example, say we choose $x = 24$. We compute the result of an input message $x = 24$ from the initial state. The result is that we non-deterministically either go to state 1 or state 2. We remember both paths and continue finding a path. We randomly choose one of the remembered paths, and try to solve the restriction of a switch departing from the last state on the path. Say we choose the path going to state 1. Then we try to solve the restriction $y \geq 2$. Again, there are many possible solutions and we randomly choose one, say $y = 16$. Then we compute the result of an input message $y = 16$ on all paths. On the path of location 1 we get $m = 24 * 16 = 384$ and on the path of location 2 we get $m = 24 + 16 = 40$. On both paths we get $n = 16$ and on both paths we end up in location 3. Then we try to find for either path a switch restriction leaving from location 3 that we can satisfy. However, both $m \leq 2$ and $n = 5$ can't be satisfied on either path, so our random paths end in location 3.

During this random walk through the STS we report all steps to the user, and we also report that on both paths, we cannot satisfy any more switch restrictions. The purpose is that the user should run the implementation simultaneously, entering the inputs given by the algorithm, and check whether the behavior of the implementation matches the behavior of the STS. If the user does this, the first mutant will already be "caught": while the STS can take no more transitions from location 3, the first mutant will, given the same input, output the value of m: non-deterministically either 40 or 384.

The algorithm continues with a breadth-first search for uncovered nodes. Immediately we find uncovered node 4, which is a child of node 3. We keep a set of all paths to node 3, which is just the set of the two paths we found in our random walk. For both paths we already found, the path to the uncovered node is 3 - 4. The attainment constraint for this path is $m \leq 2$. On the first path, this becomes $384 \leq 2$ and on the second path this becomes $40 \leq 2$. Both constraints cannot be solved, so we start propagating this constraint up one node on both paths. On the first path we go up to node 1 and the constraint becomes $y_2 \geq 2 \wedge 24 * y_2 \leq 2$, since m had value 24 in node 1 on the first path. On the second path we go up to node 2 and the constraint becomes $y_2 \geq 1 \wedge 24 + y_2 \leq 2$ because m also had value 24 in node 1 on the second path. Both constraints cannot be solved, so we propagate up again. In both paths we end up in node 0. For the first path, the constraint becomes $x_1 \geq 2 \wedge y_2 \geq 2 \wedge x_1 * y_2 \leq 2$, which has no solution. For the second path, the constraint becomes $x_1 \geq 1 \wedge y_2 \geq 1 \wedge x_1 + y_2 \leq 2$. Luckily, this constraint can be solved and we find the only solution $x_1 = 1, y_2 = 1$.

We compute the result of this input. We start out at location 0. With input $x = 1$ we can only go to location 2. There we get input $y = 1$ and we continue to location 3 with $m = 1 + 1 = 2$ and $n = 1$. From there, since $m \leq 2$, we continue to location 4 with output $m = 2$ and we terminate.

Again, during this computation we continuously report the behavior of the STS to the user. If the user checks the implementations against this behavior, mutant 2 and 3 will show their errors. On input $x = 1, y = 1$, mutant 2 will not give any output and mutant 3 will give output $x * y = 1$. Both are different from the output given in the specification and the behavior of the specification STS.

At this point, state 0 to 4 have been covered. The algorithm will search the children of nodes on the path we took last. Node 4 has no children, but node 3 has an uncovered child: node 5. The constraint for going to node 5 is $n = 5$. On the single path we took last n has value 1, so the constraint becomes $1 = 5$ which is clearly false. So we propagate the constraint up to node 2, the previous node on the path. Then the constraint becomes $y_2 \geq 1 \wedge y_2 = 5$. This constraint has one solution: $y_2 = 5$. Because we found a solution we do not need to propagate the constraint up any further and we do not need to compute and solve the entire attainment constraint. Our next test case gets input $x = 1$ (from the previous test case) and $y = 5$.

We compute the result of this test case. From location 0 with input $x = 1$ we only go to location 2 and not to 1, because $x \not\geq 2$. From there we continue to location 3 with input $y = 5$. Then $m = 6$ and $n = 5$. Then from location 3 we continue to location 5, and m becomes 12. Then we continue to location 6 and we output the value $m = 12$.

With this test case we also spot the error in mutant 4. On input $x = 1, y = 5$ the mutant will output $3 * (x + y) = 18$ instead of 12.

At this point, all states have been covered. The algorithm will search all children of nodes in the last path and after concluding that each of them has infinite distance to the closest uncovered node, terminate.

With our algorithm, we found the error in all four mutants. Without our algorithm, the STSimulator library could only generate random test cases. A random test case would probably find the error in the first mutant, just as the first random walk of our algorithm did. But because state 4 and 5 of the specification are only reached in very special cases (state 4 only if $x = 1$ and $y = 1$ and state 5 only if $y = 5$), a random test case would probably never find the error in the second, third and fourth mutant. This shows that our algorithm would be a very useful addition to the library.

10 Conclusion

We performed a case study of two black-box testing techniques: Constraint Logic Programming and formal testing with Symbolic Transition Systems. Both have their advantages and disadvantages. In Constraint Logic Programming, we have a test oracle and a way to generate test cases based on user-defined criteria. Since these criteria are user-defined, the user has a lot of control over which test cases are generated. However, this also means that there is more effort to be done by the user.

With Symbolic Transition Systems, we also have a test oracle. Additionally, we can generate random test cases. We can define a coverage criterion on the Symbolic Transition Systems and we have designed an algorithm to generate test cases based on this coverage criterion.

With our algorithm, we can in most cases cover all locations of the Symbolic Transition System. This gives us some sense of how great a part of the specification of the algorithm under test we covered. Our algorithm can handle non-deterministic systems and complex constraints. These constraints are solved using an external constraint solver. In order to save computational cost they are simplified as much as possible.

We thought about all details of our algorithm and we presented detailed pseudocode, which is ready for implementation. We demonstrated how our algorithm can be applied on a toy example, and using our algorithm we found the errors in four erroneous implementations of this example. Three of these errors would not have been found with random test cases. We think our algorithm is a useful addition to the theory of Symbolic Transition Systems.

11 Acknowledgements

The research that resulted in this thesis was carried out during an internship at Axini B.V. We would like to thank them for their supervision.
We would like to thank Inge Bethke for her reliable and steady academic and moral support.
Last but not least, we would like to thank Gineke Sietsma for her help, many interventions and wise advice.

References

[1] Pauli Aho. Extending a generic constraint solver over polymorphic data. Master's thesis, Helsinki University of Technology, Department of Computer Science and Engineering, Laboratory for Theoretical Computer Science, 2005.

[2] Krzysztof R. Apt. *Principles of Constraint Programming.* Cambridge University Press, New York, NY, USA, 2003.

[3] Krzysztof R. Apt and Mark G. Wallace. *Constraint Logic Programming using Eclipse.* Cambridge University Press, New York, NY, USA, 2007.

[4] Christian Bessire and Jean-Charles Régin. Arc consistency for general constraint networks: Preliminary results. 1997.

[5] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. The software model checker BLAST. *International Journal on Software Tools for Technology Transfer (STTT)*, 9(5):505–525, 10 2007.

[6] Céline Bigot, Alain Faivre, Jean-Pierre Gallois, Arnault Lapitre, David Lugato, Jean-Yves Pierron, and Nicolas Rapin. Automatic test generation with AGATHA. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 591–596, 2003.

[7] Lars Frantzen, Jan Tretmans, and Tim A. C. Willemse. Test generation based on symbolic specifications. In *FATES 2004, number 3395 in LNCS*, pages 1–15. Springer-Verlag, 2005.

[8] Lars Frantzen, Jan Tretmans, and Tim A. C. Willemse. A symbolic framework for model-based testing. *Formal Approaches to Software Testing and Runtime Verification*, pages 40–54, 2006.

[9] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 213–223, New York, NY, USA, 2005. ACM.

[10] James C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[11] Nikolai Tillmann and Jonathan de Halleux. Pex - white box test generation for .net. *Tests and Proofs*, pages 134–153, 2008.

[12] Jan Tretmans. Test generation with inputs, outputs and repetitive quiescence. *Software - Concepts and Tools*, 3:103–120, 1996.

[13] Jan Tretmans. Model based testing with labelled transition systems. *Formal Methods and Testing*, pages 1–38, 2008.

A Technical Report: The Implementation of the Blockbuster STS

We implemented the Blockbuster specification in an STS. We used the STSimulator Java library[5]. The structure of this STS is in Figure 14.

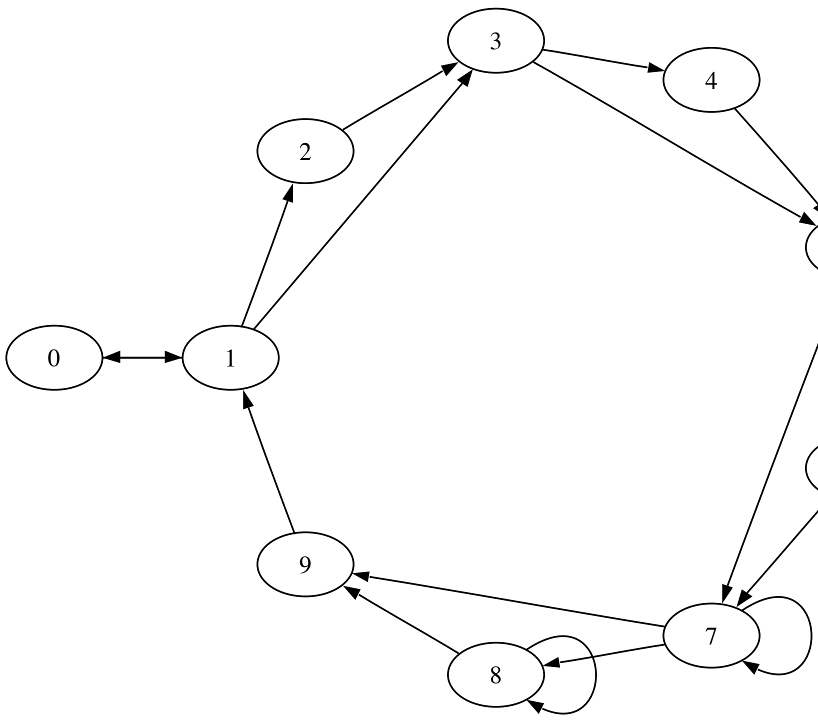


Figure 14: The structure of the Blockbuster STS

Because a full picture of the STS with all its conditions and update mappings is too complex to be comprehensible, we will discuss each of the locations separately.

State 0

The initial location is location 0. In this location the STS receives input and goes to location 1. It is also the final location: if a sufficient number of iterations have been made and the distribution of the shop is optimal, the STS returns to location 0 and waits for new input. The input expected while going from location 0 to location 1 is the following (for two categories `action` and `drama` and two popularity types `top40` and `new`):

```
drama_low           % lower bound on percentage drama
drama_opt           % optimal value for percentage drama
drama_high          % upper bound on percentage drama
action_low          % lower bound on percentage action
action_opt          % optimal value for percentage action
action_high         % upper bound on percentage action
top40_low           % lower bound on percentage top40
top40_opt           % optimal value for percentage top40
top40_high          % upper bound on percentage top40
```

[5]http://www.cs.ru.nl/~lf/tools/stsimulator/

```
new_low             % lower bound on percentage new
new_opt             % optimal value for percentage new
new_high            % upper bound on percentage new
shop.action.perc    % percentage of movies in category action
shop.action.no      % number of movies in category action
shop.action.top40   % number of movies in category action and popularity type top 40
shop.action.new     % number of movies in category action and popularity type new
shop.drama.perc     % percentage of movies in category drama
shop.drama.no       % number of movies in category drama
shop.drama.top40    % number of movies in category drama and popularity type top 40
shop.drama.new      % number of movies in category drama and popularity type new
total_no            % the total number of movies in the shop
```

The switch restriction for going from location 0 to location 1 consists of some basic constraints on the input: the percentages in the shop distribution should add up to 100%, each lower bound should be less than the optimal value and each upper bound should be more than the optimal value, etc.

State 1

State 1 is the starting point of an iteration. There is a switch back to location 0 for the case that the shop distribution is optimal and we don't need to do any more iterations. The switch restriction for this is:

$$\varphi_{1\to 0} = (\texttt{shop.action.perc} = \texttt{action\_opt}) \wedge (\texttt{shop.drama.perc} = \texttt{drama\_opt})$$

If the shop distribution is not optimal yet, we start an iteration by choosing a category to buy from. We search for the category with the largest major shortage. For every category, there is a switch to location 3 for the case that is has the largest major shortage. The restrictions for these switches are as follows:

$$\begin{aligned}
\varphi^{action}_{1\to 3} \quad = \quad & (\texttt{shop.action.perc} < \texttt{action\_low}) \wedge \\
& ((\texttt{shop.drama.perc} \geq \texttt{drama\_low}) \vee \\
& \;(\texttt{drama\_opt} - \texttt{shop.drama.perc} \leq \texttt{action\_opt} - \texttt{shop.action.perc}))
\end{aligned}$$

$$\begin{aligned}
\varphi^{drama}_{1\to 3} \quad = \quad & (\texttt{shop.drama.perc} < \texttt{drama\_low}) \wedge \\
& ((\texttt{shop.action.perc} \geq \texttt{action\_low}) \vee \\
& \;(\texttt{action\_opt} - \texttt{shop.action.perc} \leq \texttt{drama\_opt} - \texttt{shop.drama.perc}))
\end{aligned}$$

The update mappings for these switches are below. We set the buying category and calculate the number of movies we need to buy. For now, this is the shortage in the buying category. However, if there is less excess in the selling category than there is shortage in the buying category, this might still change.

$$\begin{aligned}
\rho^{action}_{1\to 3} \quad = \quad & \{\texttt{buying\_category} \mapsto \texttt{action}, \\
& \;\texttt{buy\_no} \mapsto \texttt{action\_opt} - \texttt{shop.action.perc}\} \\
\rho^{drama}_{1\to 3} \quad = \quad & \{\texttt{buying\_category} \mapsto \texttt{drama}, \\
& \;\texttt{buy\_no} \mapsto \texttt{drama\_opt} - \texttt{shop.drama.perc}\}
\end{aligned}$$

Of course if there is no major shortage, none of the restrictions for these switches to location 3 will be fulfilled. In this case we want to search for a minor shortage and we do this in location 2. So we have a switch to location 2 for the case that there is no major shortage. The restriction for this switch is below. There should be no major shortage in any category, and there should be at least a minor shortage in some category (otherwise the distribution is already optimal and we should switch to location 0 instead).

$$\begin{aligned}\varphi_{1\to 2} &= (\texttt{shop.action.perc} \geq \texttt{action\_low}) \wedge \\ &\quad (\texttt{shop.drama.perc} \geq \texttt{drama\_low}) \wedge \\ &\quad ((\texttt{shop.action.perc} < \texttt{action\_opt}) \vee \\ &\quad\ (\texttt{shop.drama.perc} < \texttt{drama\_opt}))\end{aligned}$$

The update mapping for this switch is empty: we just move to location 2 without changing any variables.

State 2

In location 2, we search for the category with the largest minor shortage. If we have found it, we switch to location 3. The restrictions for these switches are below. We check whether there is a minor shortage and no other category has a larger shortage.

$$\begin{aligned}\varphi^{action}_{2\to 3} &= (\texttt{shop.action.perc} < \texttt{action\_opt}) \wedge \\ &\quad (\texttt{drama\_opt} - \texttt{shop.drama.perc} \leq \texttt{action\_opt} - \texttt{shop.action.perc}) \\ \varphi^{drama}_{2\to 3} &= (\texttt{shop.drama.perc} < \texttt{drama\_opt}) \wedge \\ &\quad (\texttt{action\_opt} - \texttt{shop.action.perc} \leq \texttt{drama\_opt} - \texttt{shop.drama.perc})\end{aligned}$$

The update mappings for these switches are equal to $\rho^{action}_{1\to 3}$ and $\rho^{drama}_{1\to 3}$.

State 3

In location 3 we have decided from which category to buy. We need to decide from which category we're going to sell. We do this practically the same way as with buying. For every category, there is a switch to location 5 for the case that we decide to sell from that category. Restrictions for these switches are below. There should be a major excess, and there should not be any category with a larger major excess.

$$\begin{aligned}\varphi^{action}_{3\to 5} &= (\texttt{shop.action.perc} > \texttt{action\_high}) \wedge \\ &\quad ((\texttt{shop.drama.perc} \leq \texttt{drama\_high}) \vee \\ &\quad\ (\texttt{shop.drama.perc} - \texttt{drama\_opt} \leq \texttt{shop.action.perc} - \texttt{action\_opt})) \\ \varphi^{drama}_{3\to 5} &= (\texttt{shop.drama.perc} > \texttt{drama\_high}) \wedge \\ &\quad ((\texttt{shop.action.perc} \leq \texttt{action\_high}) \vee \\ &\quad\ (\texttt{shop.action.perc} - \texttt{action\_opt} \leq \texttt{shop.drama.perc} - \texttt{drama\_opt}))\end{aligned}$$

If these restrictions are fulfilled we switch to location 5 and we set the selling category. We also calculate the final number of movies we want to buy and sell. This is the minimum of the number we previously calculated (the shortage in the buying category) and the excess in the selling category. This way, we make sure we don't buy or sell too much and pass the optimum.

$$\begin{aligned}\rho^{action}_{3\to 5} &= \{\texttt{selling\_category} \mapsto \texttt{action}, \\ &\quad \texttt{buy\_no} \mapsto \min(\texttt{buy\_no}, \texttt{shop.action.perc} - \texttt{action\_opt})\} \\ \rho^{drama}_{3\to 5} &= \{\texttt{selling\_category} \mapsto \texttt{drama}, \\ &\quad \texttt{buy\_no} \mapsto \min(\texttt{buy\_no}, \texttt{shop.drama.perc} - \texttt{drama\_opt})\}\end{aligned}$$

Of course there is also the possibility that there is no major excess at all. In this case we need to search for the largest minor excess and we do this in location 4. So we have a switch to location 4 with the restriction that every category has no major excess:

$$\begin{aligned}\varphi_{3\to 4} &= (\texttt{shop.action.perc} \leq \texttt{action\_high}) \wedge \\ &\quad (\texttt{shop.drama.perc} \leq \texttt{drama\_high})\end{aligned}$$

The update mapping for this switch is empty.

State 4

In location 4 we search for the largest minor excess. For each category there is a switch to location 5 for the case that it has the largest excess. We check whether the category has a minor excess and no other category has a larger excess:

$$\begin{aligned}\varphi_{4\to 5}^{action} &= (\texttt{shop.action.perc} > \texttt{action\_opt}) \wedge \\ &\quad (\texttt{shop.drama.perc} - \texttt{drama\_opt} \leq \texttt{shop.action.perc} - \texttt{action\_opt}) \\ \varphi_{4\to 5}^{drama} &= (\texttt{shop.drama.perc} > \texttt{drama\_opt}) \wedge \\ &\quad (\texttt{shop.action.perc} - \texttt{action\_opt} \leq \texttt{shop.drama.perc} - \texttt{drama\_opt})\end{aligned}$$

The update mappings for these switches are identical to $\rho_{3\to 5}^{action}$ and $\rho_{3\to 5}^{drama}$.

State 5

In location 5, we know from which categories we are going to buy and sell. Next we decide from which popularity types within the buying category we will buy. We search for the largest major shortage within the popularity types by comparing the number of movies in the popularity type, with the lower bound on the percentage of all movies in the buying category after buying new movies. So this is a percentage of the current number of movies in the buying category plus the number of movies we are going to buy.

For every popularity type, there is a switch for the case that that type has the largest major shortage. This switch goes back to location 5, because if the shortage in that popularity type is less than the total number of movies we will buy, we also want to buy from other categories. The switch restrictions are below. We introduce for each popularity type a variable buy_type with the number of movies we are already planning to buy of that type. These variables are initialized to 0. If there is a major shortage taking into account this number, and if it is the largest shortage, we update it to indicate buying more movies of this popularity type.

$$\begin{aligned}\varphi_{5\to 5}^{top40} &= (\texttt{buying\_category.top40} + \texttt{buy\_top40} < \\ &\qquad (\texttt{top40\_low} * (\texttt{buying\_category.no} + \texttt{buy\_no})/100)) \wedge \\ &\quad ((\texttt{buying\_category.new} + \texttt{buy\_new} \geq (\texttt{new\_low} * (\texttt{buying\_category.no} + \texttt{buy\_no})/100)) \vee \\ &\quad ((\texttt{new\_opt} * (\texttt{buying\_category.no} + \texttt{buy\_no})/100) - \texttt{buying\_category.new} + \texttt{buy\_new} \leq \\ &\quad (\texttt{top40\_opt} * (\texttt{buying\_category.no} + \texttt{buy\_no})/100) - \texttt{buying\_category.top40} + \texttt{buy\_top40})) \\ \varphi_{5\to 5}^{new} &= (\texttt{buying\_category.new} + \texttt{buy\_new} < (\texttt{new\_low} * (\texttt{buying\_category.no} + \texttt{buy\_no})/100)) \wedge \\ &\quad ((\texttt{buying\_category.top40} + \texttt{buy\_top40} \geq \\ &\qquad (\texttt{top40\_low} * (\texttt{buying\_category.no} + \texttt{buy\_no})/100)) \vee \\ &\quad ((\texttt{top40\_opt} * (\texttt{buying\_category.no} + \texttt{buy\_no})/100) - \\ &\qquad \texttt{buying\_category.top40} + \texttt{buy\_top40} \leq \\ &\quad (\texttt{new\_opt} * (\texttt{buying\_category.no} + \texttt{buy\_no})/100) - \texttt{buying\_category.new} + \texttt{buy\_new}))\end{aligned}$$

In the update mapping we set the buy_type variables to the number of movies we want to buy from this particular popularity type. This is the excess in the popularity type, as long as the sum over all popularity types stays below the total number of movies we want to buy.

$$\begin{aligned}
\rho^{top40}_{5\to 5} &= \{\texttt{buy\_top40} \mapsto \min((\texttt{top40\_opt} * (\texttt{buying\_category.no} + \texttt{buy\_no})/100) - \\
&\qquad \texttt{buying\_category.top40}, \\
&\qquad \texttt{buy\_no} - (\texttt{buy\_top40} + \texttt{buy\_new}))\} \\
\rho^{new}_{5\to 5} &= \{\texttt{buy\_new} \mapsto \min((\texttt{new\_opt} * (\texttt{buying\_category.no} + \texttt{buy\_no})/100) - \\
&\qquad \texttt{buying\_category.new}, \\
&\qquad \texttt{buy\_no} - (\texttt{buy\_top40} + \texttt{buy\_new}))\}
\end{aligned}$$

If there is no major shortage in the popularity types (taking into account the number of movies we are planning to buy), but we still need to buy more movies (the sum over all popularity types of movies we are planning to buy is lower than the total number of movies we want to buy), we switch to location 6. The restriction for this switch is below.

$$\begin{aligned}
\varphi_{5\to 6} &= (\texttt{buying\_category.top40} + \texttt{buy\_top40} \geq (\texttt{top40\_low} * (\texttt{buying\_category.no} + \texttt{buy\_no})/100)) \land \\
&\quad (\texttt{buying\_category.new} + \texttt{buy\_new} \geq (\texttt{new\_low} * (\texttt{buying\_category.no} + \texttt{buy\_no})/100)) \land \\
&\quad (\texttt{buy\_top40} + \texttt{buy\_new} < \texttt{buy\_no})
\end{aligned}$$

The update mapping for this switch is empty: we just go to location 6 to search for minor shortages within the popularity types.

Another possibility is that we have reached the total number of movies we want to buy. In this case, we switch to location 7. The restriction for this switch is:

$$\varphi_{5\to 7} = (\texttt{buy\_top40} + \texttt{buy\_new} = \texttt{buy\_no})$$

The update mapping for this switch is empty.

State 6

State 6 is very similar to location 5, only now we search for a minor shortage in the popularity types instead of a major shortage. The switch restrictions are below:

$$\begin{aligned}
\varphi^{top40}_{6\to 6} &= (\texttt{buying\_category.top40} + \texttt{buy\_top40} < \\
&\qquad (\texttt{top40\_opt} * (\texttt{buying\_category.no} + \texttt{buy\_no})/100)) \land \\
&\quad ((\texttt{new\_opt} * (\texttt{buying\_category.no} + \texttt{buy\_no})/100) - \texttt{buying\_category.new} + \texttt{buy\_new} \leq \\
&\quad (\texttt{top40\_opt} * (\texttt{buying\_category.no} + \texttt{buy\_no})/100) - \texttt{buying\_category.top40} + \texttt{buy\_top40}) \\
\varphi^{new}_{6\to 6} &= (\texttt{buying\_category.new} + \texttt{buy\_new} < \\
&\qquad (\texttt{new\_opt} * (\texttt{buying\_category.no} + \texttt{buy\_no})/100)) \land \\
&\quad ((\texttt{top40\_opt} * (\texttt{buying\_category.no} + \texttt{buy\_no})/100) - \texttt{buying\_category.top40} + \texttt{buy\_top40} \leq \\
&\quad (\texttt{new\_opt} * (\texttt{buying\_category.no} + \texttt{buy\_no})/100) - \texttt{buying\_category.new} + \texttt{buy\_new})
\end{aligned}$$

The update mappings for these switches are identical to $\rho^{top40}_{5\to 5}$ and $\rho^{new}_{5\to 5}$.

When we reach the total number of movies we want to buy, we switch to location 7. The switch restriction for this switch is identical to $\varphi_{5\to 7}$, and the update mapping is empty.

State 7

In location 7 we know how many movies from each popularity type we are going to buy. Now we need to decide how much from each popularity type we're going to sell. We do this practically the same way. First we search for the largest major excess, taking into account the number of movies we're planning to sell:

$$\begin{aligned}
\varphi^{top40}_{7\to7} &= (\texttt{selling\_category.top40} - \texttt{sell\_top40} > \\
&\qquad (\texttt{top40\_high} * (\texttt{selling\_category.no} - \texttt{buy\_no})/100))) \wedge \\
&\quad ((\texttt{selling\_category.new} - \texttt{sell\_new} \leq (\texttt{new\_high} * (\texttt{selling\_category.no} - \texttt{buy\_no})/100))) \vee \\
&\quad (\texttt{selling\_category.new} - \texttt{sell\_new} - (\texttt{new\_opt} * (\texttt{selling\_category.no} - \texttt{buy\_no})/100)) \leq \\
&\quad \texttt{selling\_category.top40} - \texttt{sell\_top40} - \\
&\qquad (\texttt{top40\_opt} * (\texttt{selling\_category.no} - \texttt{buy\_no})/100)))) \\
\varphi^{new}_{7\to7} &= (\texttt{selling\_category.new} - \texttt{sell\_new} > \\
&\qquad (\texttt{new\_high} * (\texttt{selling\_category.no} - \texttt{buy\_no})/100))) \wedge \\
&\quad ((\texttt{selling\_category.top40} - \texttt{sell\_top40} \leq \\
&\qquad (\texttt{top40\_high} * (\texttt{selling\_category.no} - \texttt{buy\_no})/100))) \vee \\
&\quad (\texttt{selling\_category.top40} - \texttt{sell\_top40} - \\
&\qquad (\texttt{top40\_opt} * (\texttt{selling\_category.no} - \texttt{buy\_no})/100)) \leq \\
&\quad \texttt{selling\_category.new} - \texttt{sell\_new} - (\texttt{new\_opt} * (\texttt{selling\_category.no} - \texttt{buy\_no})/100))))
\end{aligned}$$

The update mapping is also similar to the one in location 5: we update the number of movies to sell from this popularity type to the minimum of the excess and the remaining number of movies we need to sell.

$$\begin{aligned}
\rho^{top40}_{7\to7} &= \{\texttt{sell\_top40} \mapsto \min(\texttt{selling\_category.top40} - \\
&\qquad (\texttt{top40\_opt} * (\texttt{selling\_category.no} - \texttt{buy\_no})/100), \\
&\qquad \texttt{buy\_no} - (\texttt{sell\_top40} + \texttt{sell\_new}))\} \\
\rho^{new}_{7\to7} &= \{\texttt{sell\_new} \mapsto \min(\texttt{selling\_category.new} - \\
&\qquad (\texttt{new\_opt} * (\texttt{selling\_category.no} - \texttt{buy\_no})/100), \\
&\qquad \texttt{buy\_no} - (\texttt{sell\_top40} + \texttt{sell\_new}))\}
\end{aligned}$$

If there remains no major excess but we need to sell more movies we switch to location 8:

$$\begin{aligned}
\varphi_{7\to8} &= (\texttt{selling\_category.top40} - \texttt{sell\_top40} \leq \\
&\qquad (\texttt{top40\_high} * (\texttt{selling\_category.no} - \texttt{buy\_no})/100)) \wedge \\
&\quad (\texttt{selling\_category.new} - \texttt{sell\_new} \leq (\texttt{new\_high} * (\texttt{selling\_category.no} - \texttt{buy\_no})/100)) \wedge \\
&\quad (\texttt{sell\_top40} + \texttt{sell\_new} < \texttt{buy\_no})
\end{aligned}$$

These switches have an empty update mapping.

The last possibility is that the movies we are planning to sell from the different popularity types already add up to the total number of movies we want to buy or sell. In this case we switch to location 9, again with an empty update mapping:

$$\varphi_{7\to9} = (\texttt{sell\_top40} + \texttt{sell\_new} = \texttt{buy\_no})$$

State 8

In location 8 we search for the largest minor excess:

$$\begin{aligned}
\varphi^{top40}_{8\to 8} &= (\texttt{selling\_category.top40} - \texttt{sell\_top40} > \\
&\qquad (\texttt{top40\_opt} * (\texttt{selling\_category.no} - \texttt{buy\_no})/100)) \wedge \\
&\quad (\texttt{selling\_category.new} - \texttt{sell\_new} - (\texttt{new\_opt} * (\texttt{selling\_category.no} - \texttt{buy\_no})/100) \leq \\
&\quad \texttt{selling\_category.top40} - \texttt{sell\_top40} - (\texttt{top40\_opt} * (\texttt{selling\_category.no} - \texttt{buy\_no})/100)) \\
\varphi^{new}_{8\to 8} &= (\texttt{selling\_category.new} - \texttt{sell\_new} > (\texttt{new\_opt} * (\texttt{selling\_category.no} - \texttt{buy\_no})/100)) \wedge \\
&\quad (\texttt{selling\_category.top40} - \texttt{sell\_top40} - \\
&\qquad (\texttt{top40\_opt} * (\texttt{selling\_category.no} - \texttt{buy\_no})/100) \leq \\
&\quad \texttt{selling\_category.new} - \texttt{sell\_new} - (\texttt{new\_opt} * (\texttt{selling\_category.no} - \texttt{buy\_no})/100))
\end{aligned}$$

The update mappings are identical to $\rho^{top40}_{7\to 7}$ and $\rho^{new}_{7\to 7}$.

Again, if we have reached the total number of movies we want to buy or sell, we switch to location 9. The restriction for this switch is identical to $\varphi_{7\to 9}$, and the update mapping is empty.

State 9

In location 9, we know exactly which movies we want to buy and sell, so we give output, update the shop distribution, and switch back to location 1. The output consists of the new shop distribution, the buying and selling category, and the buy_type and sell_type variables that indicate how much to buy and sell of each popularity type. The shop update is listed below:

$$\begin{aligned}
\texttt{buying\_category.top40} &\mapsto \texttt{buying\_category.top40} + \texttt{buy\_top40} \\
\texttt{buying\_category.new} &\mapsto \texttt{buying\_category.new} + \texttt{buy\_new} \\
\texttt{buying\_category.no} &\mapsto \texttt{buying\_category.no} + \texttt{buy\_no} \\
\texttt{buying\_category.perc} &\mapsto (\texttt{buying\_category.no} * 100)/\texttt{total\_no} \\
\texttt{selling\_category.top40} &\mapsto \texttt{selling\_category.top40} - \texttt{sell\_top40} \\
\texttt{selling\_category.new} &\mapsto \texttt{selling\_category.new} - \texttt{sell\_new} \\
\texttt{selling\_category.no} &\mapsto \texttt{selling\_category.no} - \texttt{sell\_no} \\
\texttt{selling\_category.perc} &\mapsto (\texttt{selling\_category.no} * 100)/\texttt{total\_no}
\end{aligned}$$

After switching back to location 1, a new iteration might start.

B Pseudocode of the Automatic Test Case Generation

Below is detailed pseudocode of our algorithm. This pseudocode is ready to be implemented. In the pseudocode we use the following definitions.
An instantiated location is a tuple (l, v) with $l \in L$ a location of the STS, and $v : Var \mapsto \mathfrak{U}$ a valuation for all location variables of the STS.
An instantiated transition is a tuple (t, m) with $t = (l, \lambda, \varphi, \rho, l')$ a transition and $m : \lambda \mapsto \mathfrak{U}$ a message, which is a valuation of all interaction variables in λ, or τ.
An instantiated path is a succession $\langle (l_0, v_0), (t_0, m_0), ..., (l_{n-1}, v_{n-1}), (t_{n-1}, m_{n-1}), (l_n, v_n) \rangle$ of instantiated locations and transitions. We use it to denote a path through the STS and a way to follow that path with a certain succession of messages.
An uninstantiated path is a succession $\langle l_0, t_0, ..., l_{n-1}, t_{n-1}, l_n \rangle$ of uninstantiated locations and transitions. It denotes just a path through the STS, without giving a way to follow that path. It might not even be possible to follow the path.

Algorithm 3 Algorithm for automatic test case generation

Input

$\mathcal{S}$: an STS

ι: an initial valuation of all location variables

```
 1: $l_0$ := the initial location of $\mathcal{S}$
 2: $I := randomPath(\{\langle(l_0, \iota)\rangle\})$
    {*Find a set of random instantiated paths through the STS, see Algorithm 4*}
 3: $tries(t) := 0$ for all transitions $t$
    {*Variable denoting how often we unsuccessfully tried to reach an uncovered node through t*}
 4: **loop**
 5:   $d_{min}$ := max_search_depth {*Usually, max_search_depth = no. of nodes in $\mathcal{S}$*}
 6:   **for all** instantiated paths $i$ in $I$ **do**
 7:     **for all** instantiated locations $(l, v)$ in $i$, from last to first **do**
 8:       **for all** transitions $t = (l, \lambda, \varphi, \rho, l')$ departing from $l$ in $\mathcal{S}$ such that $l'$ does not occur
          in any instantiated location in any instantiated path in $I$ **do** {*Search all children of
          nodes on the path for uncovered descendants*}
 9:         $(d, u) := computeUncoveredDistance(l', d_{min} - tries(t) - 1)$ {*See Algorithm 6. d
            is the distance, u is an uninstantiated path to the closest uncovered node.*}
10:         **if** $d + tries(t) < d_{min}$ **then** {*We found a new minimum that we didn't try very
            often yet*}
11:           $d_{min} := d + tries(t)$
12:           $u_{min} := \langle l, t\rangle \cdot u$
13:           $t_{min} := t$
14:         **end if**
15:         **if** $d_{min} = 0$ **then**
16:           break outer for {*Stop the search because the distance will never get below 0*}
17:         **end if**
18:       **end for**
19:     **end for**
20:   **end for**
21:   **if** $d_{min}$ = max_search_depth **then**
22:     break **loop** {*There are no more uncovered nodes or all transitions have been tried too
        often*}
23:   **end if**
24:   $(l, \lambda, \varphi, \rho, l') := t_{min}$ {*This transition leads to the uncovered node we will try to reach*}
25:   $I' := \emptyset$ {*We construct a set of all paths we already know that lead to $t_{min}$*}
26:   **for** every occurrence of $l$ in some $i = \langle(l_0, v_0), ..., (l_k, v_k), (t_k, m_k), (l, v), ...\rangle$ in $I$ **do**
27:     add $\langle(l_0, v_0), ..., (l_k, v_k), (t_k, m_k), (l, v)\rangle$ to $I'$
28:   **end for**
29:   $(I', success) := followPath(I', u_{min})$ {*Try to follow one of the paths through $t_{min}$ to the
      uncovered node, see Algorithm 7*}
30:   **if** success = true **then**
31:     $I := I'$ {*This is the new set of paths we followed, continue the loop*}
32:   **else**
33:     $tries(t_{min}) = tries(t_{min}) + 1$ {*Keep the old set of paths we followed and try a different
        transition*}
34:   **end if**
35: **end loop**
```

Algorithm 4 randomPath: Take a random path through the STS, until we can't take any more transitions or we reached a final location

Input

I: a set of instantiated paths we followed most recently. Every instantiated path in this set has the same succession of messages.

Output

I: a new set of instantiated paths which is the result of following a random path

```
1: for all $i$ in $I$ in random order do
2:   Let $(l, v)$ be the last instantiated location in $i$
3:   for all transitions $t = (l, \lambda, \varphi, \rho, l')$ such that $\lambda \neq \tau$ departing from $l$ in random order do
4:     Use a constraint solver to compute the set $M$ of all messages $m : \lambda \rightarrow \mathfrak{U}$ that satisfy $\varphi$,
       given $v$
5:     if $M \neq \emptyset$ then {We found possible messages to take transition $t$}
6:       if $\lambda$ is an input gate then
7:         m := a random member of $M$
8:       else if $\lambda$ is an output gate then {The user should say which output was observed
         from the implementation}
9:         Ask the user to choose a member of $M$ and store the result in $m$
10:        if the user does not give correct values then
11:          $M := \emptyset$ {Try the next transition}
12:        end if
13:      end if
14:    end if
15:    if $M \neq \emptyset$ then
16:      break outer for {If we found a solution, choose this transition and continue. Other-
         wise, try a different transition.}
17:    end if
18:  end for
19: end for
20: if $m = null$ then {We can't satisfy any transition guard so the path ends}
21:   return I
22: else
23:   $I := processMessage(I, m)$ {Process this message, see Algorithm 5}
24:   return $randomPath(I)$ {Continue the path}
25: end if
```

Algorithm 5 processMessage: process one message, succeeding an instantiated path
Input
I: the instantiated path we followed so far
m: the message
Output
I': the instantiated path after processing m

```
1: Report m to the user
2: $I' := \emptyset$
3: for all instantiated paths $i$ in $I$ do
4:   Let $(l, v)$ be the last instantiated location in $i$
5:   for all transitions $t = (l, \lambda, \varphi, \rho, l')$ such that $\lambda$ is of the type of $m$ and $\varphi$ is satisfied by $v$
     and $m$ do {Follow transition $t$}
6:     $Covered(l')$ := true
7:     $v' := \rho(v)$ {Apply the update mapping of $t$ to the valuation of $l$}
8:     Add $i \cdot\ (s, m) \cdot (l', v')$ to $I'$
9:     for all transitions $t' = (l', \lambda', \varphi', \rho', l'')$ such that $\lambda = \tau$ and $\varphi$ is satisfied by $v'$ do {Also
       follow all possible $\tau$ transitions after this transition}
10:      $Covered(l'')$ := true
11:      $v'' := \rho'(v')$
12:      Add $i \cdot (s, m) \cdot (l', v') \cdot (s', \tau) \cdot (l'', v'')$ to $I'$
13:    end for
14:  end for
15: end for
16: return $I'$
```

Algorithm 6 computeUncoveredDistance: compute the distance from a certain location to the closest uncovered node, until a certain maximum depth.

Input

l: a location

n: the maximum depth of the search

Output

d: the distance of l to the closest uncovered node

u: an uninstantiated path from l to the closest uncovered node

```
1: $L := \{l\}$ {*The set of locations we searched*}
2: $U := \{\langle l \rangle\}$ {*The paths we search*}
3: **for** i is 0 to n **do**
4:   $U' = \emptyset$
5:   **for all** $u = \langle l_0, t_0, ..., l_i \rangle$ in $U$ **do**
6:     **if** $l_i$ is uncovered **then**
7:       **return** $(i, u)$
8:     **else** {*Continue the path*}
9:       **for all** transitions $t_i = (l_i, \lambda, \varphi, \rho, l_{i+1})$ departing from $l_i$ **do**
10:        **if** $l_{i+1}$ is not in $L$ **then** {*If we haven't already searched* $l_{i+1}$}
11:          Add $l_{i+1}$ to $L$
12:          Add $\langle l_0, s_0, ..., l_i, t_i, l_{i+1} \rangle$ to $U'$
13:        **end if**
14:      **end for**
15:    **end if**
16:  **end for**
17:  $U := U'$
18:  **if** $U = \emptyset$ **then** {*The search ended, l has no uncovered descendants*}
19:    break
20:  **end if**
21: **end for**
22: **return** $(n + 1, \langle\rangle)$ {*Search failed, either because l has no uncovered descendants or because
    the maximum search depth has been reached*}
```

Algorithm 7 followPath: try to follow an uninstantiated path, succeeding a member of a set of instantiated paths. If that's not possible, try constraint propagation.

Input

I: an ordered list of instantiated paths, which all end in the same location

u: an uninstantiated path, starting in the final location of the instantiated paths in I

Output

I: a set of instantiated paths, all with the same messages, of which at least one ends in an instantiation of u

success: a value that indicates whether a correct output I was found

```
 1: $\langle l_0, t_0, ..., t_{n-1}, l_n \rangle := u$ {*The uninstantiated path we want to follow*}
 2: $\psi := \top$ {*We will construct the constraints for taking path u*}
 3: $length$ := the no. of switches in the longest element of $I$ + the no. of switches in $u$
 4: **for** $i$ from $n-1$ to 0 **do**
 5:    $(l, \lambda, \varphi, \rho, l') := t_i$
 6:    Apply variable renaming $r_{length-(n-i)}$ to $\varphi$ and $\rho$
 7:    Apply substitution $\rho$ to $\psi$
 8:    $\psi := \varphi \wedge \psi$
 9: **end for**
10: $U$ := an ordered list of the same size as $I$, of which all elements equal $u$
11: $\Psi$ := an ordered list of the same size as $I$, of which all elements equal $\psi$
    {*When we do constraint propagation, every member of I will get its own uninstantiated path we want to follow and
    a matching constraint*}
12: **for** i = 0 to size(I) - 1 **do**
13:    $\langle (l_0, v_0), ..., (l_{k-1}, v_{k-1}), (t_{k-1}, m_{k-1}), (l_k, v_k) \rangle := I[i]$
14:    $\langle l_k, t_k, l_{k+1}, ..., t_{n-1}, l_n \rangle := U[i]$
       {*Try to find a solution to the constraints of the uninstantiated path, given the instantiated path*}
15:    Compute the set $W$ of all assignments $w$ of interaction variables such that $v_k \cup w \models \Psi[i]$
       {*These interaction variables are from the gates of U[i]. We use a constraint solver.*}
16:    **if** $W \neq \emptyset$ **then** {*We found possible solutions to the constraint, which is a sequence of messages*}
17:       $I' := \{\langle (l_0, v_0) \rangle\}$ {*Compute the paths that are the result of these messages*}
18:       **for** j is 0 to k-1 **do** {*First the instantiated path*}
19:          $I' := processMessage(I', m_j)$
20:       **end for**
21:       **for** j is k to n - 1 **do** {*Then all messages of the solution*}
22:          $(l_j, \lambda, \varphi, \rho, l_{j+1}) := s_j$
23:          **if** $\lambda$ is an input gate **then**
24:             Randomly choose some $m : \lambda \rightarrow \mathfrak{U}$ that is part of an element of $W$
25:          **else if** $\lambda$ is an output gate **then**
26:             Let the user choose some $m : \lambda \rightarrow \mathfrak{U}$ that is part of an element of $W$
27:          **end if**
28:          Remove all members of $W$ that are inconsistent with $m$
29:          $I' := processMessage(I', m)$
30:       **end for**
31:       **return** (I', true) {*We found a solution so return it*}
32:    **else** {*Do constraint propagation*}
33:       **if** $(l_k, v_k)$ is the only element of $I[i]$ **then** {*We are already at the initial node so we can't propagate any further*}
34:          Remove $I[i]$ from $I$
35:          Remove $U[i]$ from $U$
36:          Remove $\Psi[i]$ from $\Psi$
37:          $i := i - 1$
38:       **else**
39:          $I[i] := \langle (l_0, v_0), ..., (l_{k-1}, v_{k-1}) \rangle$ {*Remove the last state from the instantiated path*}
40:          $U[i] := \langle l_{k-1}, s_{k-1}, l_k, s_k, ..., s_{n-1}, l_n \rangle$ {*Add it to the uninstantiated path*}
41:          $(l_{k-1}, \lambda, \varphi, \rho, l_k) := s_{k-1}$
42:          Apply variable renaming $r_{length-(n-(k-1))}$ to $\varphi$ and $\rho$ {*Compute the new constraint*}
43:          Apply substitution $\rho$ to $\Psi[i]$
44:          $\Psi[i] := \varphi \wedge \Psi[i]$
45:       **end if**
46:    **end if**
47: **end for**
48: **return** $(\emptyset, false)$
```