

Delft University of Technology
Master's Thesis in Embedded Systems

Extending behavioral test models with symbolic data

Christiaan Hartman



Extending behavioral test models with symbolic data

Master's Thesis in Embedded Systems

Embedded Software Section
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Mekelweg 4, 2628 CD Delft, The Netherlands

Christiaan Hartman
c.hartman@student.tudelft.nl

13th May, 2013

Author

Christiaan Hartman (c.hartman@student.tudelft.nl)

Title

Extending behavioral test models with symbolic data

MSc presentation

24th May 2013

Graduation Committee

Prof.dr. Koen Langendoen (chair)	Delft University of Technology
Prof.dr. Mohammad Reza Mousavi	Delft University of Technology, Eindhoven University of Technology and Halmstad University
dr.ir. Tim A.C. Willemse	Eindhoven University of Technology

Abstract

Model-based testing is a technique to automatically generate test cases and oracles in order to test an implementation. A typical behavioral model used for model-based testing is a Labeled Transition System (LTS). However creating the LTS models needed for testing is a complex task, and especially when data is involved the number of states and thus the complexity will explode fast. Incorporating data in an LTS leads to a Symbolic Transitions System (STS), where data is stored in variables and actions have guards and updates. The result is that the number of states is reduced and the complexity is moved to the update rules and guards.

In order to simplify the construction of STS an algorithm is presented that is capable of extracting data from an implementation and enriching the LTS model in order to form an STS. A domain expert can then inspect the model for possible faults and in the future the model can be used for regression testing of the implementation.

Preface

This thesis will mark the end of my journey as a student and educational life, although its unlikely that one will ever be done learning. It was a long but interesting journey that started at an elementary school in Delft and now ends at the Technical University of Delft.

This project builds on my interests for the development of high quality and reliable embedded software. That started when working on software for the Delfi-n3Xt nanosatellite during my Battlers thesis, and later in classes like Software Validation and Embedded Real-Time Systems.

First of all, I would like to thank my supervisor Mohammad Reza Mousavi for his support and guidance during this project. I also want to wish him good luck with his new adventures and projects in Sweden at the Halmstad University. I would like to thank all the fellow students at the TU Delft that I worked with during projects and especially the fellow students working on their thesis project at the Embedded Software Group during my time there. And of course I want to thank friends and family who supported me during all those years of study.

Christiaan Hartman

Delft, The Netherlands
13th May, 2013

Contents

Preface	v
1 Introduction	1
1.1 Coffee machine example	2
1.2 Problem statement	4
1.3 Summary of the results	4
1.4 Structure of the report	5
2 State of the art	7
2.1 Traditional test techniques	7
2.1.1 Functional testing	7
2.1.2 Symbolic Execution	9
2.2 Model-based testing	10
2.2.1 FSM based testing	10
2.2.2 LTS based testing	11
3 Models	13
3.1 First Order Logic	13
3.2 Labeled transition systems	14
3.2.1 Input-Output Transition systems	15
3.2.2 Symbolic Transition Systems	16
3.3 Conformance Testing	18
3.3.1 symbolic IOCO	19
3.4 Control Flow Graph	20
4 Enriching Behavioral Models with Data	23
4.1 Basic Algorithm	23
4.1.1 Control Flow Graph generation	24
4.1.2 Symbolic Transition system generation	27
4.1.3 Matching relation	32
4.1.4 Matching Algorithm	33
4.2 Theorem	34

5	Implementation	37
5.1	Loading LTS models	38
5.2	Generation of the CFG	39
5.3	Matching algorithm implementation	41
5.3.1	Matching algorithm	41
5.4	Using the application	43
5.5	Example	44
6	Conclusions and Future Work	49
6.1	Conclusions	49
6.2	Future Work	49
6.2.1	Improvements to the parser	50
6.2.2	Error detection and report	50
6.2.3	Model extension	50
6.2.4	Merging internal actions	50
6.2.5	Formalize a proof for Theorem 4.2.3	51

Chapter 1

Introduction

The reason to test software is because it is known that the people who designed and implemented it are not perfect and thus can and will make mistakes that result in bugs, faults and eventually failures. Furthermore we want to make a judgment about the quality of the implementation, not just based on a feeling but based on quantitative data. This is even more important for embedded systems as once they are deployed it is hard to update the software; in some cases it is even impossible for an end user at the deployment site to update the software and the embedded system has to be returned to a service point or a technician has to come and update the software manually. Although more and more embedded systems are connected to some sort of network and can be updated over this network, testing is still important to guarantee a certain quality of software and to minimize the number of needed bug fixes and thus the cost associated with fixing the problem and deploying the update.

With testing one can never guarantee that there are no bugs left in the system; however, one can establish a certain level of quality by testing. The easiest approach to testing would be to try all possible input combinations of the implementation, as this would basically guarantee finding all the possible bugs. Consider a 32-bit adder as found in modern day CPUs. Adding two 32-bit numbers gives 2^{32+32} possible input permutations. Assuming that the processor runs at $3.5GHz$ and is capable of executing one addition per clock cycle, running all these tests would take 61001 years. Things get even worse if you consider that the CPU can also do subtractions, multiplications, divisions and much more instructions. Furthermore modern CPU's support techniques to speed up the execution of instructions, among others, pipelining, caching and instruction level parallelism. The state of the pipeline, cache and functional units might result in concurrency issues when certain instructions are executed in sequence or parallel and data may or may not be persistently in the cache. Thus exhaustive testing of all input combinations is infeasible for all but some trivial implementations.

One testing method to automatically generate test cases is model-based testing. In model-based testing a model is used that describes the desired behavior at a given level of abstraction and the implementation is tested if it conforms to this model. Using the model it is possible to automatically construct test cases to test if the behavior of the implementation conforms to the desired behavior as specified by the model.

1.1 Coffee machine example

In this report a vending machine that servers coffee and tea is used as the running example. The vending machine is capable of serving coffee after the coffee button is pressed and serves tea when the tea button is pressed. After serving coffee or tea it will wait for a new user input to again serve tea or coffee. A model of the coffee machine is shown in Figure 1.1.

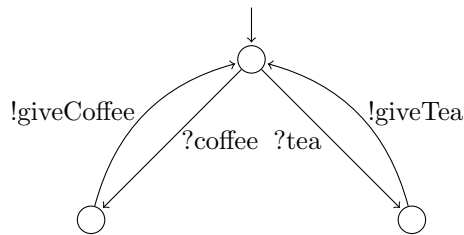


Figure 1.1: Simple coffee machine

Using the Label Transition System (LTS) depicted in Figure 1.1, with input actions `?coffee`, `?tea` and output actions `!giveCoffee`, `!giveTea` it is possible to generate test cases to test an implementation, and to verify if it is a correct implementation of the model. As with all testing techniques, a relevant question is when to stop testing. One might think that executing both input actions once is sufficient to test the implementation, but what if the vending machine only has a supply to produce five cups of coffee? Executing the coffee action once will show no fault, twice no fault, three times no fault, and so on until five cups of coffee have been served and the vending machine needs a refill. Executing for the sixth time the coffee action will not produce coffee and thus will show a fault in the implementation. Unfortunately, there is no way in model based testing (as with other black-box testing techniques) to know when the system is completely tested and no faults are left in the system. As a result, white-box metrics such as statement coverage and branch coverage have to be used to determine if enough tests have been executed.

Assume that in this case the implementation is correct and the vending machine is supposed to have a supply of five cups of coffee. That means that the model needs to be updated to incorporate the supply variable. One of

the strengths of model-based testing is that just the model needs updating, after which again automatically all the test cases can be generated (the old test cases are simply discarded). This means that it is not necessary to evaluate all the previously made test cases and decide if they are still valid or might need updating, as one would have needed to do with manually crafted test cases. The model required to implement the supply variable is shown in Figure 1.2. Since LTS models have no explicit notation of data the number of states has grown considerably revealing a downside of LTS.

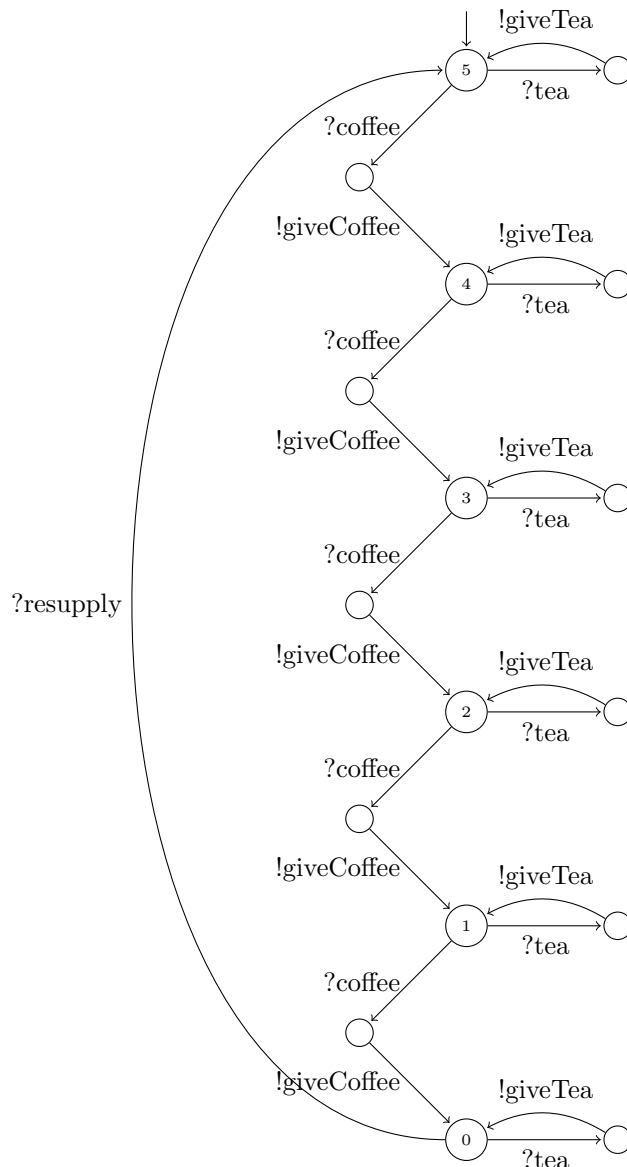


Figure 1.2: Coffee machine with encoding for the supply

As LTSs have no notion of data, the variable values have to be encoded into the state of the model. A maximum supply of five for coffee already results into seventeen states; when adding a supply of five for tea the state space of the model will quickly explode. This will only get worse when the supply values are increased. This can be solved by adding symbolic data to the model, resulting in a Symbolic Transition System (STS). The STS model for the coffee machine is shown in Figure 1.3 where the supply is present as a variable, and the `?coffee` action has a guard to indicate if the action can be executed or not. Furthermore the action `!giveCoffee` and `?resupply` have been annotated with update rules that update the supply value to indicate the new situation.

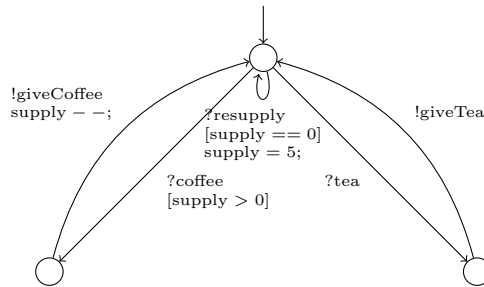


Figure 1.3: Simple coffee machine with data

1.2 Problem statement

Drawing a model with a repetition of states like in the coffee example is a tedious and error prone job for larger and more complex systems with multiple variables. The system can be modeled directly into a STS to lower the number of states but entering all the guards and update rules into the system is a lot of work, and when the code changes these need to be updated while the model might remain in essence the same. To simplify matters one may exploit the implementation to annotate the model with relevant information regarding data and its information of the control flow. The resulting model can then be manually inspected by a domain expert to make sure that it matches the intended behavior of the system. When the generated guards and update rules are correct the resulting STS can be used for regression testing purposes.

1.3 Summary of the results

In this report an algorithm is proposed that is capable of enriching the data in a behavioral model based on the source code that implements the beha-

viator. Based on the proposed algorithm an application was developed that is capable of parsing a Java class and LTS models built using the application called Yed. Based on the information in the Java file and the model, the application is able to execute the algorithm and enrich the supplied model with data. Using the application a small demonstration is given to show that the process works. Furthermore formal relations between the models and source code are proven.

1.4 Structure of the report

Following the introduction, first an overview of the current state of the art is given in Chapter 2. Then the formal definitions of the models and related concepts are given in Chapter 3. Building on these formal definitions the method for enriching LTS models with symbolic data is given in Chapter 4. This is followed by the implementation in a tool and some examples in Chapter 5. Finally a conclusion of the work done and a discussion of the future work are given in Chapter 6.

Chapter 2

State of the art

In this chapter an overview of the current testing techniques is given that are relevant to this research. Unfortunately the field of software testing is so wide that it is impossible to review all the techniques here, thus only an overview of relevant techniques is given.

2.1 Traditional test techniques

2.1.1 Functional testing

Functional testing is a form of black box testing where the specifications of the system are used to systematically test the implementation under test (IUT). This means that all tests are constructed without any knowledge of the internal operations of the IUT and thus all test cases will come from the specifications. As with most approaches, testing all possible inputs is infeasible for all but very basic programs and as a result different approaches have been developed to select test cases that are the most likely to reveal faults.

An overview of different functional testing techniques is given in chapter 2.4 of [11] and a brief explanation of these techniques is given next.

Partition Testing / Equivalence Partitioning In partition testing the input domain is divided into a set of input classes that when combined cover the complete input domain of the specification. Test cases are then constructed to cover all combinations of input classes.

Boundary Value Testing For boundary value testing the input is divided into input classes, and for each boundary three values are selected. One just inside the boundary, one just outside the boundary and one nominal value. For example when integer X is an input and has a boundary $X < a$ the test values should be: just inside the boundary $a - 1$, just outside

of the boundary a and a nominal value b such that $b < a - 1$. Boundary value analysis is a simple and easy way of constructing test cases that test for common one off errors in loops and if statements.

Some extended forms of boundary testing are, Robustness Testing, Worst Case Testing and Robust Worst Case Testing. In these forms the number of test cases is extended so that the boundaries are covered more extensively.

Special Value Testing Besides testing the boundaries in a system there are also special values that a programmer can spot. For example the quadratic formula $x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$ with as inputs $ax^2 + bx + c = 0$. We know that the solution x can either have one or two distinct real roots, or two distinct complex roots depending on the value of the discriminant. A logical choice would be to choose inputs in such a way that all three cases are tested. Furthermore when variable a of the quadratic formula equals zero the equation is no longer a quadratic equation, but if this case is not handled correct it will result in an division by zero in the quadratic-formula and thus should be treated as a special case.

Design-Based Functional Testing The requirements specify what certain requirements functions should do. When implementing these requirements functions it is not unlikely that other functions are constructed to implement the requirements. The requirements function can be seen as the root node of a tree, with the functions that are used to implement the requirement at the second level. This can be refined even further where even simpler functions are used on level $n + 1$ to implement the simple functions at level n . The smaller functions that are used to design the requirements function are called design functions, and will have requirements that are needed to implement the original requirement.

Selection of the design functions to implement the requirement should be done in such a way that the functions are accessible for testing and it must be possible to find a test data set to test the function. The input and output of each of the design functions should be completely specified to allow the use of functional testing techniques to construct tests cases.

As this technique involves splitting up the requirements and the IUT into smaller parts, it is not really black-box testing. But it uses functional testing techniques that are black-box to test the smaller subsections of the system.

Cause-Effect Graphing In causeeffect graphing, the output of the program can be divided into classes which are called effects. Impute values or causes that generate the same effect are grouped together, in a limited-entry decision table to which further optimizations can be applied. The test cases are now chosen to exercise each column of the table.

2.1.2 Symbolic Execution

Symbolic execution is a technique that is used to automatically generate test cases that result in a high coverage. To accomplish this the program is symbolically executed, and all the branches in the program are symbolically evaluated to find all the paths and the corresponding path condition (pc). The concrete values to force the execution along the path are found by finding an assignment to the variables that satisfies the pc, using a constraint solver. Solving the pc for all the collected constraints results in a set of test cases. Consider the following code snippet:

```
func(int a, int b)
{
    //Some code
    if(a>100) {
        //Some more code
        if(b > 300 && b < 400) {
            //And even more code
        }
    }
    return ;
}
```

In order to test the code snippet, all the paths need to be symbolically executed and concrete values for the variables a and b need to be found for the paths. At the start of the program the pc is initialized to *TRUE* and symbolic execution is started. For every branch a copy is made of the current pc and the negation of the condition is added to it, and the current pc is updated with the condition. For example if the path chosen assumes *TRUE* for the first *if* statement then $pc = (a > 100)$, now when for the next if statement *FALSE* is chosen $\neg(b > 300 \wedge b < 400)$ is added to the pc resulting in the following constraint: $pc = (a > 100) \wedge \neg(b > 300 \wedge b < 400)$. A constraint solver can now solve the pc to find an assignment for the variables a, b that satisfies the pc and thus exercises the chosen path. If there is no assignment that satisfies the pc the path is called infeasible, and cannot be executed. Analysis of the infeasible paths can reveal pieces of dead code (i.e code that is unreachable for any combination of input values), this as dead code implies an infeasible path but unfortunately an infeasible path does not imply dead code.

Loops with the stop condition depending on the data will result in a huge amount and potentially infinite different paths. To solve this techniques such as exploiting assertions and compositional reasoning can be used; an explanation of these techniques is found in Chapter 7 of [12]. An overview of the recent research trends in symbolic execution is given in [2].

2.2 Model-based testing

A contemporary trend in testing is generating test cases from models or model-based testing. In model-based testing the IUT is tested to check if it conforms to the model that describes the desired behavior. In some cases the model is part of the existing requirements, otherwise the model needs to be created specifically for testing purposes. Once a suitable model is constructed tools can be used to extract test cases according to for example the IOCO testing theory [10] or finite state machine (FSM) based testing theory [3]. Although the model can be completely verified using model checking techniques, due to the limitations of testing only a limited number of test cases can be executed and thus the testing can never be complete. Thus one can never be completely certain that the implementation conforms to the model.

2.2.1 FSM based testing

Finite state machine (FSM) testing [3] is based on mealy state machines where the output depends on the inputs and the current state of the system. In testing FSM's there are five fundamental problems:

1. *Homing/Synchronizing Sequence* When no reset is available or it is unreliable, a homing sequence is constructed and used to move the machine into a known state. The state after executing the homing sequence is based on the outputs that the machine produces during the execution of the homing sequence and is unknown at the start. Where the homing sequence moves the state of the machine to a none predetermined known state, an synchronizing sequence moves it to an specific predetermined state.
2. *State Identification* here the complete state digram of the machine is known, except for the initial state. The goal is to construct an input sequence that reveals the state at which the machine was when the input sequence was started.
3. *State Verification* As with problem 2 the state diagram is known, but not the initial state. Now the question is given that it is in state s verify that this is true.
4. *Machine Verification/Fault Detection/Conformance Testing* Given a specification machine or model M , check if the model M is equivalent to the implementation.
5. *Machine Identification* is the problem of extracting the transition diagram of an given machine using a test sequence. This is used in for example reverse engineering of communication protocols and other state machines.

For FSM based testing most of the fundamental problems have been solved, only it is still not known how to construct checking sequences deterministically in polynomial time. Practically there are still problems that need to be solved, like the state space explosion for larger applications that involve data and variables.

2.2.2 LTS based testing

Ioco [10] is a conformance relation with its origin in the theoretical area of testing equivalences and refusal testing. To model systems ioco uses labeled transition systems (LTS) with inputs and outputs, where input actions are indicated by ? and output actions with !. With the implementation i and the specifications s the ioco notation of conformance is defined as: $i \mathbf{ioco} s \leftrightarrow_{def} \forall \sigma \in \text{Straces}(s) : \text{out}(i \mathbf{after} \sigma) \subseteq \text{out}(s \mathbf{after} \sigma)$. The function $\text{out}(s)$ denotes all the output action that are possible in state s and $s \mathbf{after} \sigma$ gives the state of s after the trace σ is preformed. $\text{Straces}(s)$ is the set of all the possible traces over the model s , where a trace is a sequence of actions. This means that the set of output actions of the IUT after all possible traces in the model needs to be a subset of the model after the same trace. The paper describing the ioco relation [10] also gives an algorithm to generate all possible test cases, but as always with testing the number of test cases is far too large for practical purposes. One of the ongoing research questions is to make a good selection amongst these test cases.

sIOCO Using an LTS to model complex systems will quickly result into a state explosion, as LTSs lack the required abstraction to handle data in the system efficiently. As a result this will lead to a large number of states that are needed to represent all the combinations of the variable values. To solve this the notation of LTS is extended to Symbolic Transition System (STS) by adding restrictions (guards) and update mappings to transition labels. The resulting notation has a higher level of abstraction and can handle complexer applications without resulting in a state explosion. The IOCO relation for STSs is called Symbolic IOCO (SIOCO) [6] and is an extension of IOCO to deal with STS models. As a given STS can be converted into an LTS by the rules in Definition 11 of [6] denoted by $\llbracket S \rrbracket$, following this definition a STS S and physical system P are SIOCO compatible $P \mathbf{sioco} S$ iff $P \mathbf{ioco} \llbracket S \rrbracket$.

As STSs can be converted into LTSs, the algorithm proposed in the ioco paper [10] can be reused to generate all the test cases, but as the conversion of the STS into a LTS results into a state space explosion this is impractical. To overcome this an algorithm is proposed in [6] that avoids the state space explosion by combining the test generation from the STS with an on-the-fly execution of the test cases. That way only a part of the state space has to

be generated avoiding the state space explosion that occurs when generating the complete state space.

Chapter 3

Models

During the development of complex systems, models are used to predict behavior, test and evaluate the performance of systems. For software systems the de facto standard is the Unified Modeling Language (UML) which offers a wide range of different models. But in the field of formal testing, more formal models like labeled transition systems and symbolic transition systems are used. In this chapter a description will be given of the models that will be used in the remainder of this report. But before the definitions of the models can be given some basic of logic is recalled.

3.1 First Order Logic

The basic concepts form first order logic are used as defined in [6, 7]. The first order logic structure is assumed as:

- A logical signature $\mathcal{S} = (F, P)$ 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 $\mathcal{M} = (\mathcal{U}, (f_{\mathcal{M}})_{f \in F}, (p_{\mathcal{M}})_{p \in P})$ where:
 - \mathcal{U} being a nonempty set called the universe.
 - For all $f \in F$, $f_{\mathcal{M}}$ is a function $\mathcal{U}^n \mapsto \mathcal{U}$, with arity n .
 - For all $p \in P$, $p_{\mathcal{M}} \subseteq \mathcal{U}^n$ is a predicate over elements of the universe with arity n .

Given a set of variables \mathcal{X} , the set of terms over these variables is written as $\mathcal{T}(\mathcal{X})$ and are built using function symbols in F and variables in $X \subseteq \mathcal{X}$.

Terms $t \in \mathcal{T}(\emptyset)$ are called ground-terms and $\text{var}(t)$ is used to indicate the set of variables used in the term t .

A function $\sigma : \mathcal{X} \mapsto \mathcal{T}$ is called a term-mapping. And the identity mapping, is defined for all $x \in \mathcal{X}$ as $\mathbf{id}(x) = x$. For term mappings the following notation is used. The function $\mathcal{T}(Y)^X$ assigns to each variable $x \in X$ a term $t \in \mathcal{T}(Y)$ and for all the variables $x \notin X$ the term x where $X \cup Y \subseteq \mathcal{X}$.

The set of first order formulas σ is denoted by $\mathcal{F}(X)$, for each $X \subseteq \mathcal{X}$. The set of bound variables in the first order formula σ is denoted as $\mathbf{bound}(\sigma)$ and the set of free variables as $\mathbf{free}(\sigma)$. A tautology is represented by \top .

A valuation ϑ is a function $\vartheta : \mathcal{X} \mapsto \mathcal{U}$. The set of all the valuations is defined as $\mathcal{U}^{\mathcal{X}} =_{def} \{\vartheta : \mathcal{X} \mapsto \mathcal{U} \mid \vartheta \text{ is a valuation of } \mathcal{X}\}$. $\vartheta \in \mathcal{U}^X$ is written where $X \subseteq \mathcal{X}$ when only the valuations of the variables X are of interest. For all the other variables not in X the valuation is set to an arbitrary element of the set \mathcal{U} . For two valuation $\vartheta \in \mathcal{U}^X$ and $\varsigma \in \mathcal{U}^Y$ where $X \cap Y = \emptyset$, the union

is defined as: $(\varsigma \cup \vartheta) =_{def} \begin{cases} \vartheta(x) & \text{if } x \in X \\ \varsigma(x) & \text{if } x \in Y \\ *, \text{ is an arbitrary element of } \mathcal{U} & \text{otherwise} \end{cases}$

If a formula σ is satisfied with respect to a given valuation ϑ this is denoted by $\vartheta \models \sigma$. Extending the evaluation to whole terms based on a valuation ϑ is called a term-evaluation denoted $\vartheta_{\text{eval}} : \mathcal{T}(\mathcal{X}) \mapsto \mathcal{U}$. The composition of functions $f : B \mapsto C$ and $g : A \mapsto B$ is denoted as $f \circ g$.

3.2 Labeled transition systems

A labeled transition system (LTS) is a model that allows to describe the operation of a system by a sequence of actions performed by the system. A common way of displaying these systems is by using a directed graph where the nodes represent the states and the edges labeled with the action name the actions.

The definition of LTS as described in the ioco paper [10] will be followed and is repeated here for convenience.

Definition 3.1. A labeled transition system is a 4-tuple $\langle Q, L, T, q_0 \rangle$ where

- Q is a countable, non-empty set of states;
- L is a countable set of labels;
- $T \subseteq Q \times (L \cup \{\tau\}) \times Q$, with $\tau \notin L$, is the transition relation;
- $q_0 \in Q$ is the initial state.

Where $q \xrightarrow{\mu} q'$ is written when $(q, \mu, q') \in T$ and $q \xrightarrow{\mu_1 \dots \mu_n} q'$ when there is a sequence of states such that $\exists q_0, \dots, q_n : q = q_0 \xrightarrow{\mu_1} q_1 \xrightarrow{\mu_2} \dots \xrightarrow{\mu_n} q_n = q'$. Furthermore $q \rightarrow$ and $q \not\rightarrow$ indicate that there is or is no state q' .

Besides the observable actions as defined in L that define the interactions of the system with the environment there is a internal action represented by the label τ ($\tau \notin L$). The internal action is a special action in the sense that the environment is unable to observe the internal action. It is assumed that the states of the system are unobservable, and that the environment is unable to determine if and when an internal action is executed. A weaktrace starting from state s with a, b and c actions interleaved with a number of τ 's before and after each action and leading to state s' is denoted $s \xRightarrow{a \cdot b \cdot c} s'$.

Definition 3.2. For an LTS $p = \langle Q, L, T, q_0 \rangle$, with $q, q' \in Q$ and $a, a_i \in L$ and where ϵ defines the empty sequence of actions, the notation of weak trace is defined below:

$$\begin{aligned} q \xRightarrow{\epsilon} q' &\Leftrightarrow_{def} q = q' \vee q \xrightarrow{\tau \cdots \tau} q' \\ q \xRightarrow{a} q' &\Leftrightarrow_{def} \exists q_1, q_2 : q \xRightarrow{\epsilon} q_1 \xrightarrow{a} q_2 \xRightarrow{\epsilon} q' \\ q \xRightarrow{a_1 \cdots a_n} q' &\Leftrightarrow_{def} \exists q_0 \cdots q_n : q = q_0 \xRightarrow{a_1} q_1 \xRightarrow{a_2} \cdots \xRightarrow{a_n} q_n = q' \end{aligned}$$

3.2.1 Input-Output Transition systems

An Input-Output transition system (IOTS) is an LTS where all the actions in L are either inputs or outputs of the considered system for which it is assumed that all input actions are enabled in all states and outputs are never refused by the environment. All output actions belong to the set L_U and are decorated with ! and all input actions belonging to the set L_I and are decorated with ?.

Definition 3.3. An Input-Output transition system is a 5-tuple $\langle Q, L_I, L_U, T, Q_0 \rangle$ where:

- $\langle Q, L_I \cup L_U, T, Q_0 \rangle$ is a LTS.
- L_I and L_U are sets of input and output actions that are disjoint $L_I \cap L_U = \emptyset$
- All input actions are enabled in any reachable state.

Starting in a state it is possible to execute a sequence of actions leading to a new state. This is denoted by p **after** σ , where p is the starting state and σ the chosen path. Due to nondeterministic behavior, it is however possible that there is not just one possible state but a set of states that the system can be in after executing a sequence of actions.

Definition 3.4. For a sequence of actions σ and a starting state p , or a set of states P , p **after** σ is defined as follows:

- p **after** $\sigma =_{def} \{p' | p \xrightarrow{\sigma} p'\}$
- P **after** $\sigma =_{def} \bigcup \{p \text{ after } \sigma | p \in P\}$

For an IOTS the set of output actions that can possibly be executed by the model in state a is denoted by $out(a)$. Combined with **after** this makes it possible to define the set of all the output actions the system can generate after a certain sequence of actions σ starting in the state a , denoted by $out(a \text{ after } \sigma)$. When the system is in a state where it is impossible to perform an output action, the state is called quiescent. We annotate quiescent states with a particular event, denoted by δ , to explicitly model quiescence in the traces and the output of the system as an event. We add a self transition labeled with δ to the model, to explicitly define such situations in the traces and output sets of the system.

Definition 3.5. Given that q is a state and Q a set of states of a transition system, $out()$ is defined as follows:

- $\delta(q) =_{def} \forall \mu \in L_U \cup \{\tau\} : q \not\xrightarrow{\mu}$
- $out(q) =_{def} \{x \in L_U | p \xrightarrow{x}\} \cup \{\delta | \delta(q)\}$
- $out(Q) =_{def} \bigcup \{out(q) | q \in Q\}$

Definition 3.6. Given a IOTS $p = \langle Q, L_I, L_U, T, q_0 \rangle$ where L^* indicates a set containing all the finite sequence over L and ϵ indicates the empty sequence, the following concepts are defined:

- $L_\delta =_{def} L \cup \{\delta\}$
- $p_\delta =_{def} \langle Q, L_I, L_U \cup \{\delta\}, T \cup T_\delta, q_0 \rangle$ where $T_\delta =_{def} \{q \xrightarrow{\delta} q | q \in Q, \delta(q)\}$
- $Straces(p) =_{def} \{\sigma \in L_\delta^* | p_\delta \xrightarrow{\sigma}\}$

The $Straces(p)$ are called the suspension traces of p .

3.2.2 Symbolic Transition Systems

Symbolic Transition Systems (STS) [6, 7] extend the notation of LTS by adding guards and update rules. This results in a model with a higher level of abstraction that can handle more complex applications using a more succinct representation.

Definition 3.7. A Symbolic Transition System is a tuple $\langle Q, q_0, V, i, I, A, \rightarrow \rangle$, where:

- Q is a countable set of locations and $q_0 \in Q$ is the initial location.
- V is a countable set of location variables.
- $i \in \mathcal{T}(\emptyset)^V$ is the initializations of the location variables.
- I is a set of interaction variables, disjoint from V .
- A is a finite set of gates. The unobservable gate is denoted τ ($\tau \notin A$); we write A_τ for $A \cup \{\tau\}$. The arity of the gate $\lambda \in A$, is a natural number. And the type of a gate $\lambda \in A_\tau$, denoted $\text{type}(\lambda)$, is a tuple of length $\text{arity}(\lambda)$ of distinct interaction variables. Where the unobservable action has no interaction variables $\text{arity}(\tau) = 0$.
- $\rightarrow \subset Q \times A_\tau \times \mathcal{F}(V \cup I) \times \mathcal{T}(V \cup I)^V \times Q$ is the switch relation. We write $q \xrightarrow{\lambda, \varphi, \rho} q'$ for $(q, \lambda, \varphi, \rho, q') \in \rightarrow$ where λ is called the switch restriction and ρ the update mapping.

An example of an STS model is given in Figure 3.1. This STS model represents a coffee machine that is capable of serving multiple cups of coffee at the same time, and has a supply for coffee. In this model there is a single interaction variable `Cups`, that holds the number of cups the user wants. There are two location variables: `nrCups` and `supply`, the `nrCups` variables is used to store the number of cups the user requested, as the interaction variables are only local to the switch relation. The `supply` variable holds the number of cups of coffee that are available in the machine. The guard and the update rules are placed below the gate where guards are enclosed in square brackets and the update rules are separated by the ';' character. Observe that sending a value using an output action is done by defining an interaction variable with the gate, and assigning it a value is done via the guard that restricts the action to the assigned value.

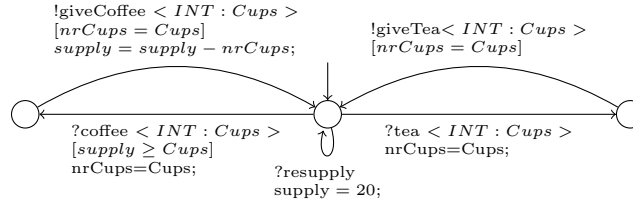


Figure 3.1: STS model of a coffee machine

Generalized switch relation STS

Weak traces over actions in STS, are more complex than with LTS as it is not possible to simply combine the guards and update rules. The main issue is that the update rules in the current action have an effect on the guards and the update rules in the next actions. In order to define weak actions [7] defines history variables, that represent the data on a point that could have been communicated over a gate in the system.

Definition 3.8. Given an STS $S = \langle Q, q_0, V, i, I, A, \rightarrow \rangle$. Assuming we have a history variable set I_1, I_2, \dots that are disjoint from each other and from $V \cup I$ of S . We define $\widehat{Var} =_{def} V \cup \widehat{I}$ with $\widehat{I} =_{def} \bigcup_j I_j$. The variable-renaming $r_n \in I_n^I$ is assumed to be bijective.

The generalized switch relation $\Rightarrow \subseteq Q \times A^* \times \mathcal{F}(\widehat{Var}) \times \mathcal{I}(\widehat{Var})^V \times Q$ is defined as the smallest relation following these rules:

$$(S\epsilon) \quad q \xrightarrow{\epsilon, \top, \mathbf{id}} q$$

$$(S\tau) \quad q \xrightarrow{\sigma, \varphi \wedge \psi[\rho], [\rho] \circ \pi} q' \text{ if } q \xrightarrow{\sigma, \varphi, \rho} q'' \text{ and } q'' \xrightarrow{\tau, \psi, \pi} q'$$

$$(S\lambda) \quad q \xrightarrow{\sigma, \lambda, \varphi \wedge (\psi[r_n])[\rho], ([\rho] \circ \pi)} q' \text{ if } q \xrightarrow{\sigma, \varphi, \rho} q'' \text{ and } q'' \xrightarrow{\lambda, \psi, \pi} q' \text{ and } n = \text{length}(\sigma) + 1$$

As with the generalized switch relation for LTS, unobservable actions are hidden without affecting the observable actions in the system. An example of a weak symbolic action is given in Figure 3.2.

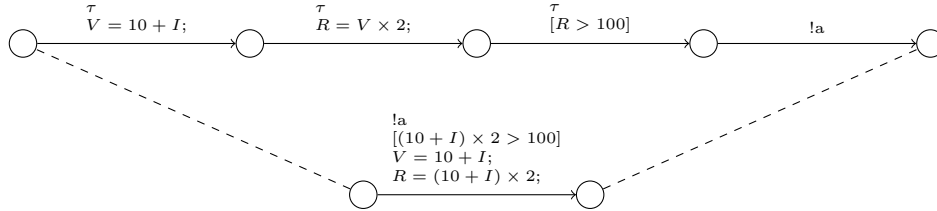


Figure 3.2: Example of weak STS action

3.3 Conformance Testing

Input-output conformance (ioco) [10] is a conformance relation used in model based testing that uses LTS models as specifications.

Definition 3.9. Given an implementation i as an IOTS with the input and output labels L_I, L_U and a specification s as LTS with the same labels the ioco relation is defined as follows:

$$i \text{ ioco } s \Leftrightarrow_{def} \forall \sigma \in \text{Straces}(s) : \text{out}(i \text{ after } \sigma) \subseteq \text{out}(s \text{ after } \sigma)$$

3.3.1 symbolic IOCO

The ioco relation is extended to the setting of STS models, resulting in a notion called symbolic ioco (sioco) [6]. A given STS can be converted into an LTS by creating a state for every combination of variable valuation and location and the actions between the states encode the guards and update rules, by the presence or lack of the action to the next state that represents the new variable value. This conversion is indicated by $\llbracket A \rrbracket$ and is defined below.

Definition 3.10. Given a STS $S = \langle Q, q_0, V, i, I, A, \rightarrow \rangle$ the interpretation of S as a LTS is given by $\llbracket S \rrbracket = \langle Q', L, T, q'_0 \rangle$ where:

- $Q' = Q \times \mathcal{U}^V$ are the state and all the possible combinations of variable values.
- $q'_0 = (q_0, \text{eval} \circ i) \in Q'$ initial state built up from the initial variable values and initial state.
- $L = \bigcup_{\lambda \in A_\tau} (\{\tau\} \times \mathcal{U}^{\text{arity}(\lambda)})$
- $T \subseteq Q' \times L \times Q'$ as defined:
$$\frac{q \xrightarrow{\lambda, \varphi, \rho} q' \quad \varsigma \in \mathcal{U}^{\text{type}(\lambda)} \quad v \cup \varsigma \models \varphi \quad v' = (v \cup \varsigma) \text{eval} \circ \rho}{(q, v) \xrightarrow{(\lambda, \varsigma(\text{type}(\lambda)))} (q', v')}$$

Using the conversion from STS to LTS makes it possible to define the sioco relation based on the ioco relation.

Definition 3.11. Given a model S in the the form of an STS and an implementation I in the form of an IOTS, the sioco relation is defined as follows:

$$I \text{ sioco } S \text{ iff } I \text{ ioco } \llbracket S \rrbracket$$

And when implementation I is in the form of an Input-Output STS the sioco relation is defined as follows:

$$I \text{ sioco } S \text{ iff } \llbracket I \rrbracket \text{ ioco } \llbracket S \rrbracket$$

3.4 Control Flow Graph

A Control Flow Graph (CFG) is a directed Graph where the nodes represent program statements and the edges indicate statements that can be executed in a sequence. Using a CFG it is possible to represent the control flow structure of a module or a program. Labels with conditions are present on the edges to indicate the edge taken after special statements such as *if*, *while*, *for*, *until* and *switch*.

Definition 3.12. A Control Flow Graph is an LTS = $\langle Q, L, \rightarrow, q_0 \rangle$ where:

- Q is the set of program statements, including the: *if*, *while*, *for*, *until* and *switch* statements.
- L is a set of labels containing conditions belonging to the special statements. Where *else* signifies the path taken iff none of the other conditions are met.
- \rightarrow indicates the next program state, i.e. $q \xrightarrow{l} q'$ when there is an execution path in which q' appears immediately after q .
- q_0 indicates the first program statement to be executed.

Listing 3.1 shows a code snippet of a simple java hello world example with a for loop. The matching CFG of this example is shown in Figure 3.3.

```
public static void main(String args [])
{
    for(int i = 0; i < 10; i++) {
        System.out.println("Hello World!");
    }
}
```

Listing 3.1: Hello world example with for loop

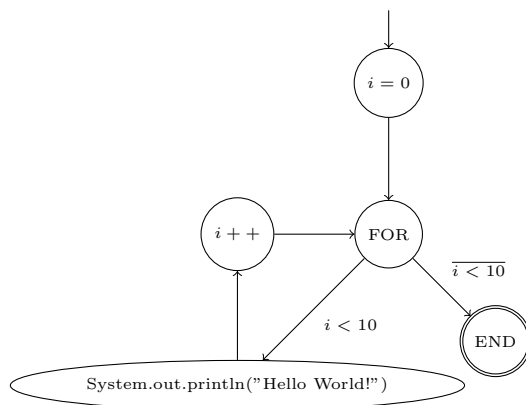


Figure 3.3: Control flow graph of the hello world example in Listing 3.1

Chapter 4

Enriching Behavioral Models with Data

The goal of this chapter is to describe an algorithm, which given a behavioral model as an LTS and an implementation, extracts data-related information from the implementation and attaches it to the corresponding behavior in the LTS. In other words the, goal of the algorithm is to enrich a specification represented by an LTS model using data dependencies found in an actual implementation. The result is an STS if the LTS is correctly implemented.

Many complications arise when the complete problem is considered; to manage these difficulties, extra assumptions are made to simplify the problem.

4.1 Basic Algorithm

For the basic version of the algorithm a couple of extra assumptions are used to simplify the development of the algorithm. Thus the following items are assumed:

- No internal τ actions are present in the specifications.
- The application is single threaded.
- All the variables in the code have some influence on the control flow.

The input of the algorithm is an LTS describing the desired behavior of the system, and the source code of the implementation of the system. The output of the system is an STS model in the same shape as the supplied LTS. The following two basic steps are performed to generate the STS:

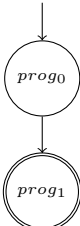
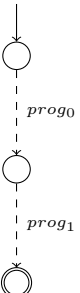
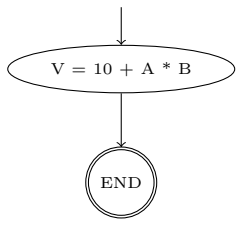
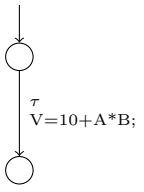
1. Extract a CFG from the code.
2. Generate an STS based on the comparison of the CFG and the LTS.

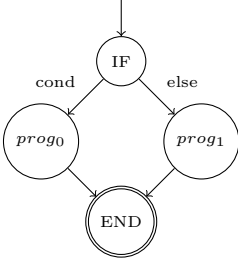
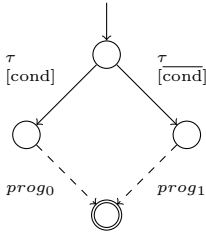
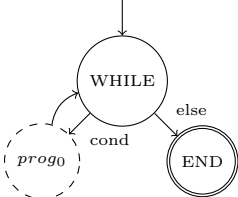
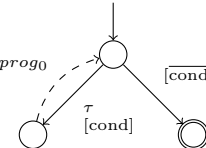
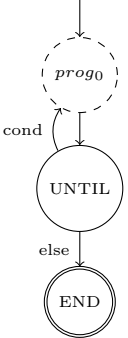
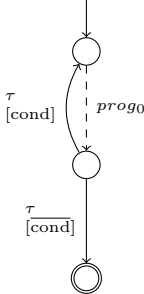
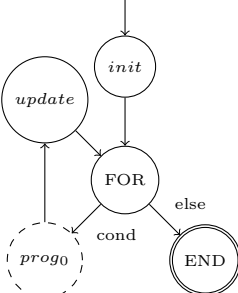
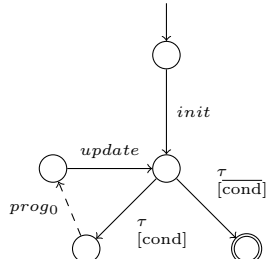
In the first step the code is processed to extract the CFG, in a future enhancement post processing can be done to ensure that only the necessary information is present in the CFG and that variables and statements not affecting the input-output behavior are removed. The second step then builds the relations between the states and actions of the LTS and CFG. Based on these relations the information in the LTS and the CFG is combined to form the final STS.

4.1.1 Control Flow Graph generation

Translation of code into a CFG is done using the translation scheme given in Table 4.1. In the actual CFG, nodes such as *endif* and *endwhile* may be merged with the next *Prog_n* node.

Table 4.1: From code to CFG and to STS

Code	CFG equivalent	STS equivalent
Sequential composition: <i>prog0</i> ; <i>prog1</i> ;		
Statement: $V = 10 + A * B$;		
Continued on next page		

Code	CFG equivalent	STS equivalent
Conditional: IF(cond) THEN <i>prog0</i> ELSE <i>prog1</i> END		
While loop: WHILE(cond) DO <i>prog0</i> END		
Repeat-until loop: REPEAT <i>prog0</i> UNTIL(cond) END		
For loop: FOR(<i>init</i> ; cond; <i>update</i>) <i>Prog0</i> END		

Continued on next page

Code	CFG equivalent	STS equivalent
Switch Condition: SWITCH case1: prog0 break case2: prog1 break default : prog2 break END		
Function call: Foo(Arg0,Arg1,...)		

Testing software systems can be divided into the following testing levels:

Unit testing, isolates the individual components (classes) using scaffolding in order to test the components with test cases.

Integration testing, combines several components in order to find faults in their interactions.

System level testing, the completely integrated system is tested in order to verify that it meets its requirements.

Extracting a CFG from code can be done for all three the testing levels as long as the source code is available, the challenge for system level testing lies in identifying the input and output actions of the system. Aside from the complexity of detecting the input and output actions, one can question the meaning of the CFG when it includes for example code from a GUI library that is used by the applications but the code is not part of the project. For Unit testing and for integration testing the inputs and outputs of the system can respectively be defined as the functions of the components and

the function calls made by the components. For that reason the focus in this research is on Unit testing and Integration testing where inputs and outputs of the systems will be function calls to and from the system under test respectively.

For every function that represents an input action to the system a CFG can be constructed, describing the control flow of that function. Once done for all the input functions this will give a set of separate CFG's for every input action of the system. These separate CFG's can be combined into one large model of the implementation by creating an initial state that has an input action for every function, and where the end of the function returns back to the initial state.

Extracting a CFG from the coffee machine code snippet given in Listing 4.1 results in the STS as shown in Figure 4.1. As a CFG or STS generated from code can be directly translated into each other both are now referred to as CFG when generated from code.

```
public class CoffeeMachineController {  
  
    void resupplyCoffee()  
    {  
        if(coffeeSupply == 0) {  
            coffeeSupply = 5;  
        }  
    }  
  
    void Coffee()  
    {  
        if(coffeeSupply >= 1) {  
            CM.outputCoffee();  
            coffeeSupply--;  
        }  
    }  
  
    void Tea()  
    {  
        CM.outputTea();  
    }  
  
    private int coffeeSupply = 0;  
    private CoffeeMachine CM;  
}
```

Listing 4.1: Simple coffee machine that can serve coffee and tea

4.1.2 Symbolic Transition system generation

After generating the CFG from the code, the CFG will have a large number of states and τ actions that are not present in the LTS. These τ actions represent updates and guards on variables and need to be placed with the

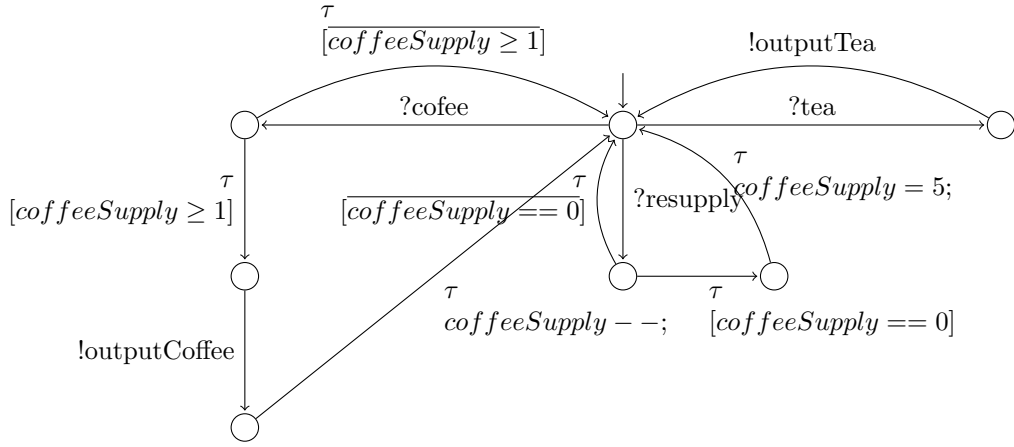
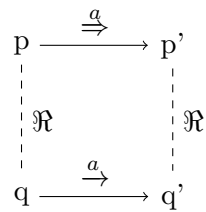


Figure 4.1: STS of the coffee machine implementation

correct actions in the LTS.

Equivalence relation

In order to map these τ actions to the correct action into the LTS, a relation between a weak action in the CFG and an action in the LTS has to be constructed. This is done, by first creating a relation between the states and actions of the LTS and CFG, in such a way that there are states p, p' in the CFG and q, q' in the LTS, where p has a relation with q and p' has a relation with q' , and where both can execute the same action, i.e. $p \xrightarrow{a} p'$ and $q \xrightarrow{a} q'$ Graphically this can be expressed as follows where \mathfrak{R} indicates the relation:



For the relation between the action in the LTS and the weak action in the CFG, the relation between the states p and q is referred to as the start of the relation. And the relation between the states p' and q' is referred to as the end of the relation.

Consider a simple example of which the code listing and the corresponding CFG are given in Figure 4.2. The two τ transitions in the CFG represent the two conditions from the if statement, followed by their respected output actions.

```

void a ()
{
    if (con) {
        b ();
    } else {
        c ();
    }
}

```

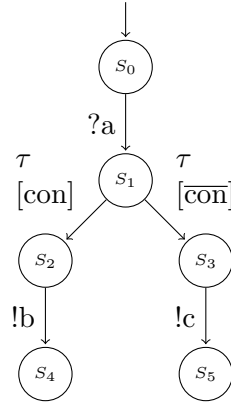


Figure 4.2: Example listing containing *if* statement and its matching CFG

For this application two different valid LTS's exist and for both of these the chosen relation should link the states such that the actions in the LTS are connected to one weak trace in the CFG. To accomplish this the matching relation needs to satisfy the following conditions;

1. The implementation (CFG) contains far more details (states and transitions) than the model (LTS) and hence, the relation should be able to match several transitions of the implementation to a single transition in the model.
2. Starting from the initial state, the start of a relation between actions in the CFG and LTS should be the end of an other relation between actions in the CFG and LTS or the relation between the initial states.

In order to build the relations between the actions there needs to be a relation between the states of the CFG and LTS as outlined before. One of the possibilities would be to use an ioco like relation; $\text{out}(Q_S \text{ after } \epsilon) \subseteq \text{out}(Q_L \text{ after } \epsilon)$ between the states where Q_S is a state in the CFG and Q_L a state in the LTS. As can be seen in Figure 4.3 the result for the first LTS model looks promising, as the relations are created as expected.

For the second LTS model the result is a bit different as can be seen in Figure 4.4 there are three options for state L_1 that would be correct according to the relation, where in truth only the relation between L_1 and S_1 is correct. The problem with the subset in the relation is that the actions are not required to be there but if they are there the relation should be build correctly. To overcome this the output equivalence relation is introduced that requires both states to have the same output actions.

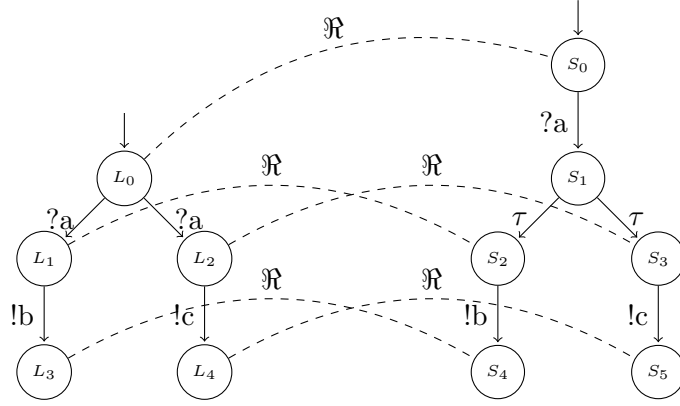


Figure 4.3: ioco like relation, left LTS model 1, right CFG

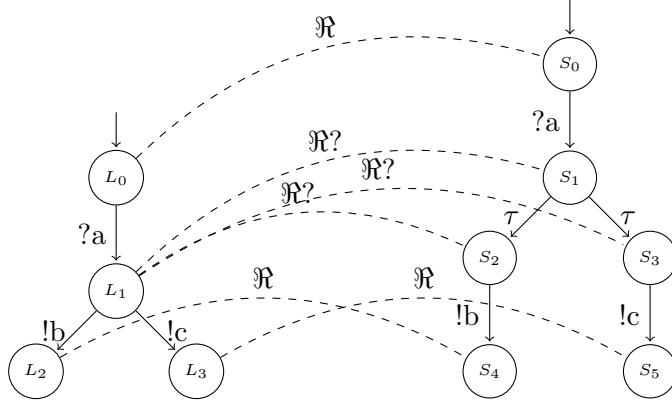


Figure 4.4: ioco like relation, left LTS model 2, right CFG

Definition 4.1. Output equivalence. Given a specification $\text{IOTS} = \langle Q_S, L_U \cup L_I, \rightarrow_S, q_{0s} \rangle$ and an implementation $\text{IOTS} = \langle Q_I, L_U \cup L_I, \rightarrow_I, q_{0i} \rangle$, a state of $q_s \in Q_S$ is output equ. to a state $q_i \in Q_I$, denoted by $q_s =_{\text{out}} q_i$ iff:

$$\text{out}(q_s \text{ after } \epsilon) = \text{out}(q_i \text{ after } \epsilon)$$

Applying the output equivalence to the first LTS model will result in the same relations as for the ioco like relation, as depicted in Figure 4.3. However for the second LTS model the resulting relation have changes as depicted in Figure 4.5. As can be seen there is now just one possible relation for the states L_1 , and this relation ensures that the relations between the actions are chosen such that the correct relations between the actions will be build.

Unfortunately only the output equivalence relation is not enough to guarantee that all the paths in the CFG are chosen correctly. The example given

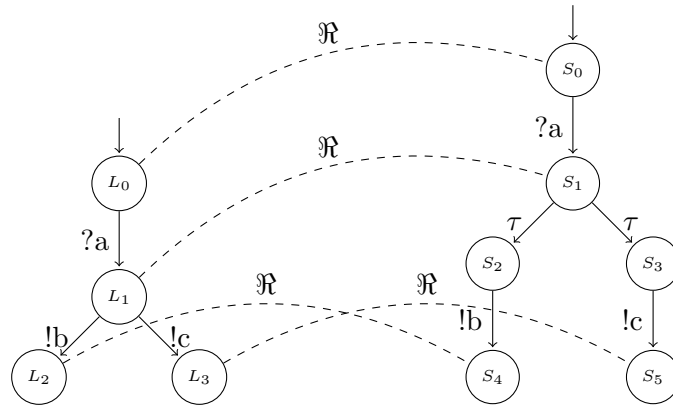


Figure 4.5: output equivalence relation, left LTS model 2, right CFG

in Figure 4.2 is rather simple as there are no variables present in the code and thus the number of τ actions is limited to the two from the *if* statement. When extra statements are added this will result in more τ actions as can be seen in Figure 4.6 where the variable *VarA* is introduced.

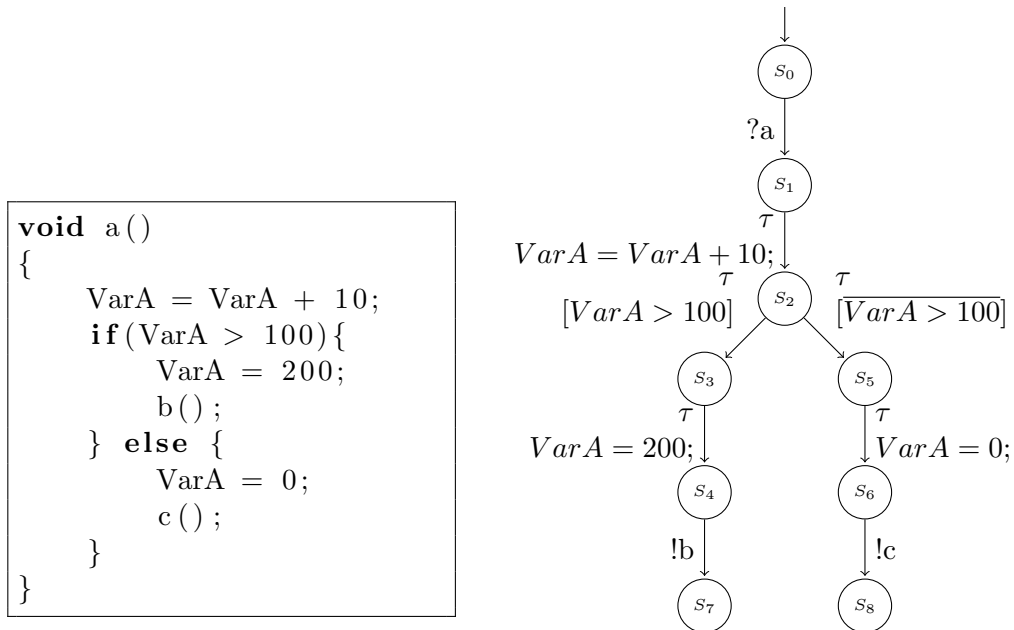


Figure 4.6: More complicated example containing multiple τ actions.

Again the same two LTS models are possible where one of them is shown in Figure 4.7. Unfortunately both will show that some nodes in the LTS can be connected to multiple nodes in the CFG, this is not a problem per definition but it does complicate things. What the relation shows is that

some τ actions can be placed on either the previous action or the next action in the LTS, and that there is no unique correct way to place them based on the LTS. The only thing that needs to be ensured is that these τ actions are placed just with one actions in the LTS. This is ensured by requiring that the first relations between actions start at the initial state and that subsequent relations between actions start where previous relations have ended.

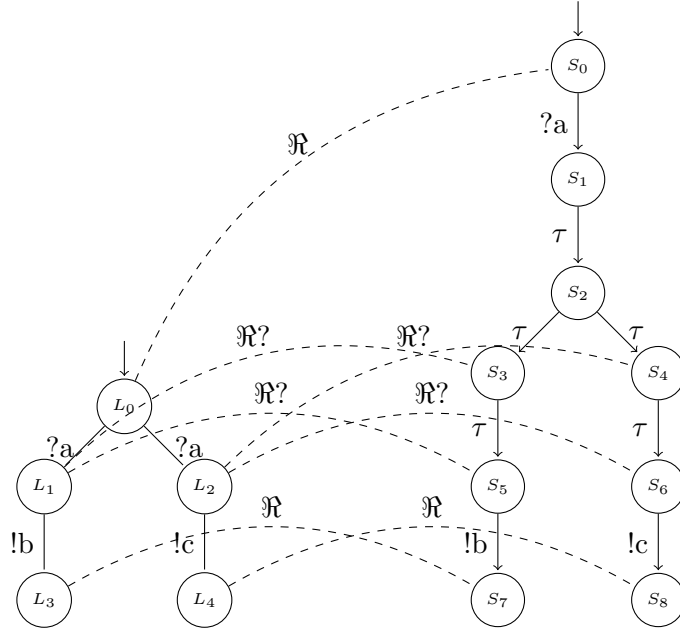


Figure 4.7: Output equivalence on implementation with multiple τ actions.

4.1.3 Matching relation

In order to ensure that the relations are correctly built as outlined in the previous sections. A matching relation is proposed.

Definition 4.2. Matching Relation. Given a specification $\text{IOTS} = \langle Q_S, L_U \cup L_I, \rightarrow_S, q_{S0} \rangle$ and an implementation $\text{IOTS} = \langle Q_I, L_U \cup L_I, \rightarrow_I, q_{I0} \rangle$. A relation $\mathfrak{R} \subseteq Q_S \times Q_I$ and function $f_{\mathfrak{R}}(Q_S \times L \times Q_S) \mapsto (Q_I \times L \times Q_I)^*$ is a matching relation iff:

- $(q_{S0}, q_{I0}) \in \mathfrak{R}$, i.e., the initial states are in the relation.
- $\forall (q_0, q_1) \in \mathfrak{R}, q_0 =_{out} q_1$, i.e., states in the relation are output equivalent.
- $\forall (q_0, q_1) \in \mathfrak{R}, q'_0 \in Q_S, a \in L_U \cup L_I, q_0 \xrightarrow{a} q'_0 \implies \exists q'_1, q_1 \xrightarrow{a} q'_1 \wedge f_{\mathfrak{R}}(q_0, a, q'_0) = q_1 \xrightarrow{a} q'_1 \wedge (q'_0, q'_1) \in \mathfrak{R}$ i.e., for every state in the relation

there exists an action in the specification that can be mimicked by an action in the implementation such that the resulting states have a relation. And the action in the specification may only have a relation with one weak action in the implementation.

In order to store the built relations between states and actions between the CFG and the LTS a relation graph is used.

Definition 4.3. Given an LTS $\langle Q, q_0, L, \rightarrow \rangle$ and an STS $\langle Q', q'_0, V, i, I, L, \rightarrow \rangle$, a relation graph is a 4-tuple $\langle Q'', r_0, \tau^*.L.\tau^*, \rightarrow'' \rangle$ with:

- $Q'' = Q \times Q'$, i.e., the Cartesian product of the sets of states in LTS and STS, respectively.
- $r_0 = (q_0, q'_0)$ The initial relation between the initial states of the LTS and STS.
- $\tau^*.L.\tau^*$ is the set of weak actions over the actions of the STS.
- $\rightarrow'' \subseteq (Q'' \times (\tau^*.L.\tau^*) \times Q'')$

4.1.4 Matching Algorithm

The goal of the matching algorithm is to build all the variations of the STS that meet the requirements as set in the matching relation. The algorithm does this in the following three steps:

1. initialization,
2. a refinement step that is repeated until the complete relation is built, and
3. a merging step that merges the relevant τ actions and adds them to the LTS to form an STS.

Initialization Given an CFG I and a LTS S , the initial states of these systems are checked to be output equivalent. If so, a partial solution relating only the initial states of I and S is added to the solution set P . (A partial solution is a matching relation of which the domain of states is a strict subset of LTS's states.)

Refinement For the refinement step a state $p = (q, s)$ is taken from the partial solution in the solution set P . The state p should be chosen such that at least for a transition $q \xrightarrow{l} q'$, q' is not in the domain of the states in the considered partial solution. The set of partial solutions is then refined based on all possible matching $s \xrightarrow{\tau^*.l.\tau^*} s'$, such that $q' =_{out} s'$ and $\forall (q'' \times s'') \in P : q'' = q' \implies s'' = s'$

In order to ensure that the requirement: $f_{\mathfrak{R}}(q, l, q') = s \stackrel{a}{\Rightarrow} s'$ is not violated it must be ensured that all of the states in the LTS are part of at most one relation. This as in order for the function to give a different solution, one or both the states q, q' will need to have two or more relations with states in the CFG.

The refinement process is continued until the possible solution set is empty indicating that there is no solution, or until all the possible solutions are complete meaning that the set of states for all the partial solution that need to be evaluated is empty.

Construction of the final STS The final STS is based on the structure of the LTS and the information in the relation graph. The elements of the STS $S = \langle Q', q'_0, V', i', I', A', \rightarrow' \rangle$ are defined as followed:

- Q' Is defined based on the states of the LTS.
- q'_0 Is defined based on the initial state of the LTS.
- V' Is the set of location variables of the CFG.
- i' The initialization of the location variables of the CFG.
- I' Is the set of Interaction variables from the CFG.
- A' Is the set of gates from the CFG.
- \rightarrow' Is defined based on the weak actions in the relation graph and the relations build between the CFG and the LTS. If there is and $(Q_i, C_i) \xrightarrow{l} (Q_j, C_j)$ and $C_i \xrightarrow{\lambda, \varphi, \rho} C_j$ then $Q_i \xrightarrow{\lambda, \varphi, \rho} Q'_j$

4.2 Theorem

Theorem 4.2.1. Given an LTS Q and the STS Q' derived from the code, the matching algorithm will only generate a STS iff the systems are ioco equivalent, i.e. $Q \mathbf{ioco} Q'$ when abstracted from data.

Proof. In order to prove that $Q \mathbf{ioco} Q'$. we use induction on the length of the traces of Q' . We prove that for every trace σ in Q' the following relation $\text{out}(q_0 \mathbf{after} \sigma) \subseteq \text{out}(q'_0 \mathbf{after} \sigma)$ holds.

Base case The base trace is ϵ . By step 1 of the algorithm, the initial states of the two transition systems must be output equivalent and thus the ioco relation for the trace ϵ must hold.

Induction step Assuming that for each σ of length n , $\text{out}(q_0 \mathbf{after} \sigma) \subseteq \text{out}(q'_0 \mathbf{after} \sigma)$ holds, prove that $\text{out}(q_0 \mathbf{after} \sigma \cdot a) \subseteq \text{out}(q'_0 \mathbf{after} \sigma \cdot a)$ holds.

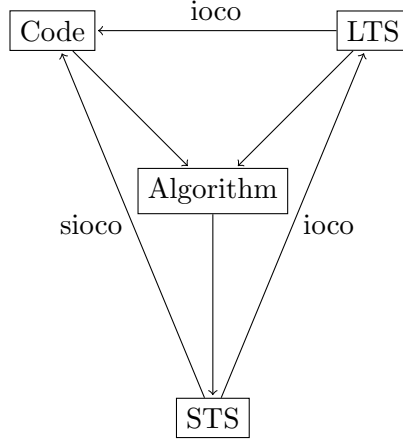


Figure 4.8: Relations between code and the models.

According to the IH there exists a state $q_i \in Q$ and a state $q'_i \in Q'$ that have a relation and thus are output equivalent after the trace σ : $\exists q_i \in Q, \exists q'_i \in Q' : \text{out}(q_i) = \text{out}(q'_i) \wedge q_i \in (q_0 \text{ after } \sigma) \wedge q'_i \in (q'_0 \text{ after } \sigma)$

According to the matching relation two states only have a relation if: $a \in \text{out}(q_i) \exists q_{i+1} \in Q, q_i \xrightarrow{a} q_{i+1} \implies \exists q'_{i+1} \in Q', q'_i \xrightarrow{a} q'_{i+1}$ and $q_{i+1} =_{\text{equ}} q'_{i+1}$. Thus $\text{out}(q_0 \text{ after } \sigma \cdot a) \subseteq \text{out}(q'_0 \text{ after } \sigma \cdot a)$ holds

As the base case and the induction step have been proven it has been shown according to the induction hypotheses that the theorem is valid over all the traces of Q . □

Theorem 4.2.2. Given an STS Q_s generated by the algorithm based on the LTS Q_l . The following relation holds $Q_s \text{ ioco } Q_l$ when abstracted from data.

Proof. As Q_s is generated based on Q_l by placing guards and update rules on the actions, removing these update rules and actions again produces the same LTS as Q_l thus $Q_s =_{id} Q_l$ when abstracted from data. And as two exactly the same LTS must be ioco equivalent as after every trace the set of states will be the same and the output actions that are possible will be the same as well. □

Theorem 4.2.3. Given an implementation Q_c in the form of an STS derived from the code, and the STS Q_S generated by the algorithm based on Q_c . The following relation holds $Q_c \mathbf{sioco} Q_s$

When abstracted from data the STS model will be the same as the LTS model that was used in the generation and thus according to Theorem 4.2.1 the STS is ioco with respect to the implementation when abstracted from data. As the algorithm makes no changes to the LTS but only adds guards and updates rules based on the implementation and the matching relation to generate the STS. This results in an STS that given an state and a variable valuation after executing the same sequence of actions is capable of executing the same actions or less than the implementation.

Chapter 5

Implementation

To demonstrate the working of the proposed algorithm a demo application is developed. A schematic overview of the internal operations in the application can be seen in Figure 5.1.

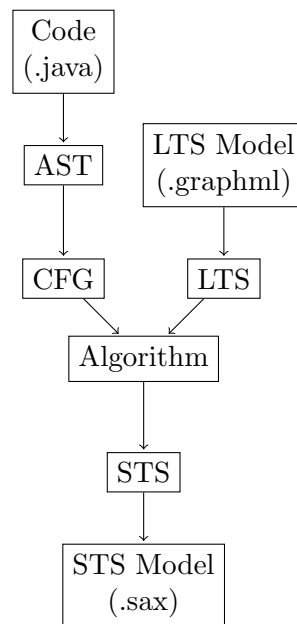


Figure 5.1: Schematic overview of application.

A detailed explanation of Figure 5.1 is provided in the remainder of this chapter, which is organized as follows. In Section 5.1, we explain how we parse and load the LTS in our tool. In Section 5.2, we describe how the CFG is generated from code. In Section 5.3, we show how the matching algorithm is implemented. In Section 5.5, we conclude by providing some examples.

5.1 Loading LTS models

To design and draw the LTS an application called Yed¹ is used. This application was chosen as it offers a simple way to draw LTS models and as the models are stored in XML format, loading such models is fairly simple. The only downside of using Yed is that indicating the initial state is not possible. To work around this problem an extra node is added with an edge to the initial node. Making this node invisible results in an arrow to the initial state, while loading the model the invisible node and the edge leading out of it are interpreted as an arrow pointing to the initial state of the LTS. An example LTS in Yed is shown in Figure 5.2.

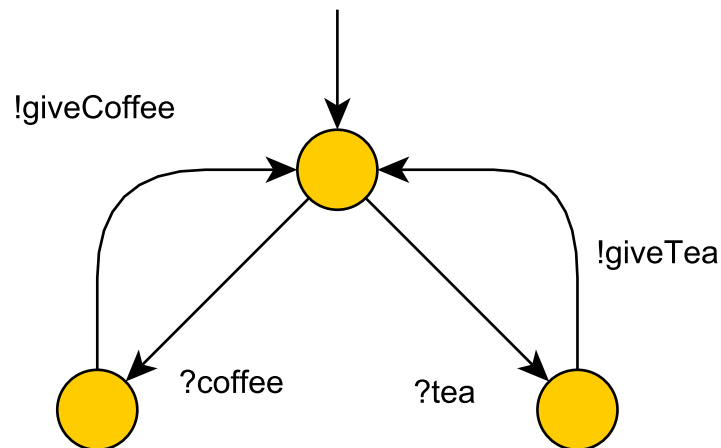


Figure 5.2: Yed LTS model, of a coffee machine.

The XML file from Yed is parsed, by using a standard Java library. Using this library for every tag “node” the ID value is saved. This is used as a reference to the node when adding the transition relations. Furthermore the visible parameter in the tag “y:NodeLabel” is checked to see if it is set to false indicating that the edge leading out of this state is the indicator of the initial state and thus this state is not part of the LTS. If the state is visible the node is added to the LTS. For every edge tag the ID’s in the source and target parameters are used to look up the source and target states in the LTS. If the source is the invisible state, the target state is used as the initial state of the LTS, if not then a transition relation is added between the two states in the LTS using the name of the edge as the action name for the transition relation.

¹http://www.yworks.com/en/products_yed_about.html

5.2 Generation of the CFG

Parsing the code and generating the CFG directly is a complicated task. This is why parsing code in general is done by first generating an abstract syntax tree (AST). By walking through the AST, it is then possible to generate the CFG of the application. An AST is a tree structure describing the structure of the source code in the tree form where the nodes represent the constructs in the source code. An example of such an AST and its matching code can be seen in Figure 5.3.

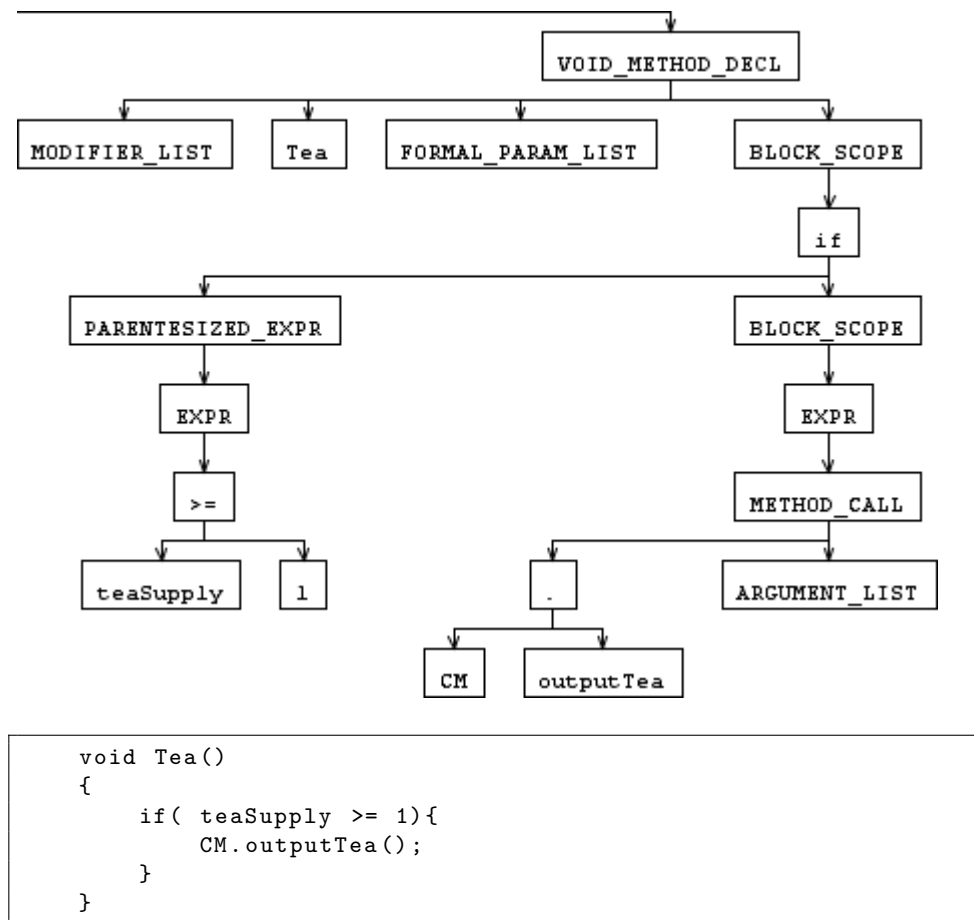


Figure 5.3: AST of the method declaration `tea()` .

Parsing and building an AST from the source code is not a trivial task especially for languages such as Java and C#. ANTLR tool is used to generate the AST and to walk the AST in order to generate the CFG of the code. ANTLR [9, 8] is described by its creator Terence Parr as follows:

“ANTLR (ANother Tool for Language Recognition) is a power-

ful parser generator for reading, processing, executing, or translating structured text or binary files. It's widely used to build languages, tools, and frameworks. From a grammar, ANTLR generates a parser that can build and walk parse trees.”

To generate a parser that is capable of building an AST, ANTLR needs a grammar describing the language, and structure of the AST. Building such a grammar is a project on its own, but fortunately there are grammars for various programming languages available on the ANTLR website² including one for Java 1.5³ which is used to build the AST.

Using the grammar, ANTLR builds a lexer and a parser. The lexer converts the stream of characters found in the source file into tokens and the parser parses the tokens and produces the AST. Once an AST is generated, the next step is to walk the AST and construct the CFG, in order to do this ANTLR uses a treewalker. A treewalker is again a grammar that matches the nodes in the AST and executes code when walking through the tree that is used to generate the CFG. The treewalker walks the tree from left to right and while walking parts of the CFG are constructed bottom up which are combined in every step to form in the end the complete CFG of the application.

In order to create a treewalker that is capable of generating the CFG custom code has to be added to the rules of the tree grammar. The code for these rules needs to take the parts that were already generated and combine them in the correct way. For example the if statement has a condition followed by two statements to be executed when the condition is evaluated, to true or false respectively. The condition is an expression, and the two statements can be anything from a single statement to a block of statements with while and for loops; the else statement may not be present at all. The statements will recursively be evaluated by the tree parser and will form a part of the CFG, according to the rules in Table 4.1. For all the supported statements and expressions code was added in order to facilitate the generation of the CFG of the java source code.

Currently the tree walker supports the following statements: *if*, *while* and *for*, statements such as: *assert*, *break*, *for each*, *do while*, *try – catch* and *switch* are currently not supported. All variable types except characters are supported, however *byte*, *short*, *int* and *long* are all interpreted as an integer and *float* and *double* are both interpreted as *double* this as the XML format used to save the final STS only supports a limited number of simple variable types. All expressions applicable for these variable types are supported however arrays of variables and enumerators are not supported.

Creation of the final parser based on the grammar files to build the CFG was done using the development tool ANTLRWorks [5].

²www.antlr3.org/grammar/list.html

³www.antlr.org/grammar/1207932239307/Java1_5Grammars/

5.3 Matching algorithm implementation

In order to implement the matching algorithm, first an algorithm is presented that is used to build the set of weak actions in the CFG leading out a state. Next the matching algorithm is presented.

Construction of weak traces

To construct a set of weak actions leaving a state the following rules have to be applied recursively for every outgoing action of a state, where “Action = τ ” indicates that the action leaving the state is a τ action. And Act indicates if already an action other than τ was encountered and thus the trace is already a weak action.

Action = τ	Act	Actions preformed
True	False	Add the action to the trace; Continue
True	True	Add the action to the trace; add the trace to the set of weakAct; Continue
False	False	Add the action to the trace; add the trace to the set of weakAct; set Act to True; Continue
False	True	Stop end of the trace no more weak actions in this branch.

Applying these results in Algorithm 1.

Algorithm 1 Generating the set set of all weak actions leaving state s

```

function WEAKACT( $s$ )
  return weakAct( $s$ ,  $\epsilon$ , false)
function WEAKACT( $s$ ,  $t$ , Act)
  for all  $a \in Act(s)$  do
    if  $a = \tau$  then
       $t' \leftarrow t \cdot a$ 
      if Act = True then
         $X \leftarrow X \cup \{t'\}$ 
       $X \leftarrow \text{weakAct}(s \text{ after } a, t', \text{Act})$ 
    else if Act = False then
       $t' \leftarrow t \cdot a$ 
       $X \leftarrow X \cup \{t'\}$ 
       $X \leftarrow \text{weakAct}(s \text{ after } a, t', \text{True})$ 
  return X

```

5.3.1 Matching algorithm

The matching algorithm is implemented as explained in Section 4.1.4. In order to implement the algorithm a couple of functions are used:

unrefinedState Returns a single relation in the relation graph that needs to be refined. If there are multiple states than can be refined an arbitrary state that needs to be refined is given.

createSTS Builds the final STS based on a relation graph or a set of STSs based on a set of relation graphs.

complete From a set of relation graphs the subset of complete relations is returned. Where a complete relation graph is defined as a relation graph for which all the states are refined.

With theses functions the complete algorithm is given in Algorithm 2. The input of the algorithm is a CFG in the form of an STS and an LTS, and the output is a set of STSs or an empty set when no solution is found.

Algorithm 2 The matching algorithm

Precondition: $S = \langle Q, q_0, V, i, I, A, \rightarrow \rangle$ is the CFG of the code as an STS, and $I = \langle Q, L_I, L_U, T, p_0 \rangle$ is an IOTS.

Postcondition: C is a set of STSs that are all solutions, if $C = \emptyset$ then no solution was found.

```
function MATCHINGALGORITHM( $S, I$ )
  if  $\text{out}(q_0) \neq \text{out}(p_0)$  then
    return  $\emptyset$ 
   $r \leftarrow \langle \{(q_0 \times p_0)\}, (q_0 \times p_0), \emptyset, \emptyset \rangle$ 
   $P \leftarrow \{r\}$ 
  while  $|P| > 0$  do
     $r \in P$ 
     $P \leftarrow P \setminus r$ 
     $(q \times p) \leftarrow \text{unrefinedState}(r)$ 
     $P' \leftarrow \text{refine}(\text{Act}(q), \text{WeakAct}(p), r, (q \times p))$ 
     $C \leftarrow \text{createSTS}(\text{complete}(P')) \cup C$ 
     $P \leftarrow (P' \setminus \text{complete}(P')) \cup P$ 
  return  $C$ 
```

Precondition: Act is the set of actions in the LTS that need to be refined, $weakAct$ is the set of actions in the STS, $R = \langle Q'', r_0, L, \rightarrow'' \rangle$ is the current relation graph that is refined and $(q \times p)$ the relation that is being refined.

Postcondition: P' contains all the refined solutions.

```
function REFINE( $Act, WeakAct, R, (q \times p)$ )
   $a \leftarrow \in Act$ 
  for all  $a' \in WeakAct$  do
    if  $a = a' \wedge \text{out}(p \text{ after } a') = \text{out}(q \text{ after } a)$  then
       $(q' \times p') \leftarrow (q \text{ after } a \times p \text{ after } a')$ 
      if  $\forall (q'' \times p'') \in Q'' : q'' = q' \implies p'' = p'$  then
         $E \leftarrow ((q \times p) \times WeakAct \times (q' \times p'))$ 
         $G' \leftarrow \langle Q'' \cup (q' \times p'), r_0, L \cup WeakAct, \rightarrow'' \cup E \rangle$ 
         $Act' \leftarrow Act \setminus a$ 
         $WeakAct' \leftarrow WeakAct \setminus a'$ 
        if  $|Act'| = 0$  then
           $P' \leftarrow P' \cup G'$ 
        else
           $P' \leftarrow \text{refine}(Act', WeakAct', G', (p \times q)) \cup P'$ 
  return  $P'$ 
```

5.4 Using the application

The application uses a simple graphical user interface (GUI) as can be seen in Figure 5.4 that allows selecting the input files and output directory for the

algorithm. By pressing the Load button a file selection dialog will appear

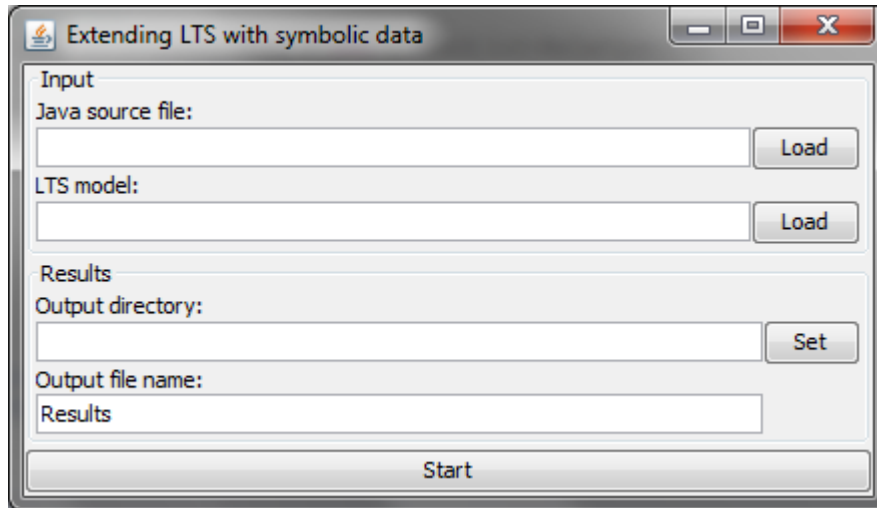


Figure 5.4: GUI of the application

allowing the user to select the Java (*.java) or Yed (*.graphml) file, alternatively the user can also type the path to the file manually. When selecting the LTS model, the output directory will automatically be set to the directory of the LTS model. If the user want to use a different directory pressing the set button will allow selecting an alternative directory for the results. After the user has supplied the correct files, pressing the start button will run the algorithm and when complete the output files will be saved in the output directory named according to the filename specified and appended with a sequence number. When the results are saved the application will show a pop-up informing the user that the process is complete and how many results where found. The STS models are saved as XML files according to the STSchema Version 111110⁴.

5.5 Example

As a demonstration of the application a version of the coffee machine example is used. The Yed LTS model van be seen in Figure 5.5 and the code is given in Listing 5.1.

⁴www.frantzen.info/archives/P20.html

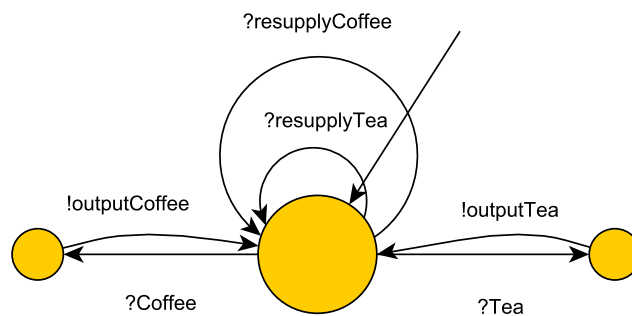


Figure 5.5: Yed model of the coffee machine example

```

public class CoffeeMachineController {
    private int coffeeSupply = 0;
    private int teaSupply = 0;
    private CoffeeMachine CM;

    public CoffeeMachineController(CoffeeMachine CM)
    {
        this.CM = CM;
    }

    public void resupplyTea(int tea)
    {
        if(teaSupply == 0) {
            teaSupply = tea;
        }
    }

    public void resupplyCoffee(int coffee)
    {
        if(coffeeSupply == 0) {
            coffeeSupply = coffee;
        }
    }

    public void Coffee(int coffee)
    {
        if(coffeeSupply >= coffee) {
            CM.outputCoffee(coffee);
            coffeeSupply -= coffee;
        }
    }

    public void Tea()
    {
        if( teaSupply >= 1){
            CM.outputTea();
        }
    }
}
  
```

Listing 5.1: Coffee Machine controller

Running the application with these files as input gives 4 different solutions displayed in Figure 5.6.

The difference between these models is the guards and update rules on the resupply actions. As the implementation in the code for both of these actions has an if statement this results in two weak actions in the CFG, and as a result there exist two different assignments for both actions resulting in four solutions. In order to test an application with these models a scaffolding has to be created in order to interface the code with an ioco test tool for example Jtorx [1]. Once done any of these models can be used to run tests on the implementation, however as not all the models implement the resupply function correctly some will end up in a unwanted state where the guards for the actions *?coffee* and or *?tea* and their respective resupply action are false. And thus when reaching the initial state these actions can not longer be executed, nor can their respective resupply function.

However assuming the code has not changed and testing the implementation based on any of these models using sioco none will reveal any faults. A domain expert should be able to inspect all the models for faults in the implementation and select the most appropriate one for regression testing. In this case the most appropriate model for regression testing would be the top most model in Figure 5.6

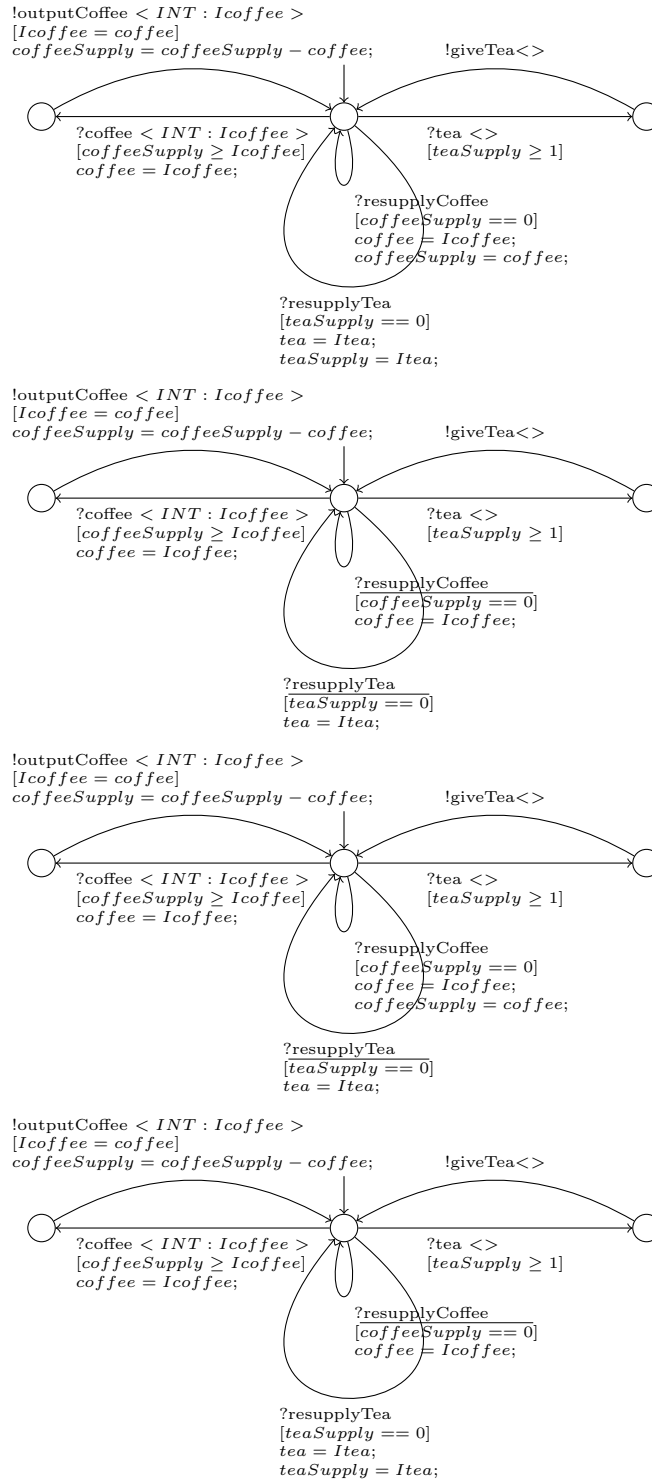


Figure 5.6: Results for the Coffee Machine Example

Chapter 6

Conclusions and Future Work

6.1 Conclusions

Symbolic Transition Systems (STSs) are behavioral models that represent the high-level behavior of systems with data STSs can be used as suitable models for mode-based testing of such systems. However, constructing STSs may be a complex task for domain experts.

One possible solutions to tackle the complexity of constructing an STS has been presented by using an LTS and the implementation to construct an STS. In order to achieve this a matching relation between the CFG of the implementation and the LTS model has been presented along with an algorithm to build this relation and the final STS. Furthermore theorems and proofs are proposed based on the ioco/sioco relations between the models and the implementation to witness the correctness of our approach. Using an example it has been demonstrated that it is possible to enrich an LTS model with data based on its implementation.

Several issues remain, open such as the error detection in the algorithm in order to find out why extracting the guards and update rules failed and extending the model when there are multiple actions in the implementation that can be related to the same action in the implementation.

6.2 Future Work

This section will given an overview of further work to improve and extend on the idea and implementation.

6.2.1 Improvements to the parser

Currently the parser is able to parse a subset of Java and can only build CFGs from a single class. This can be extended to allow parsing the complete Java language. Furthermore currently all functions in the code are interpreted as output actions, this can be changed where only a subset of the functions is seen as output actions. For the functions that are not seen as output actions a CFG can be build and placed where the function call was done in order to include the behavior of that function in the complete CFG. Another big improvement would be to allow filtering of the variables to remove variables and code branches that have no influence on the input output behavior of the system. One possible way of doing this would be using program slicing [4], by constructing a slice of the application that only effects the relevant variables.

6.2.2 Error detection and report

The current algorithm is capable of building a relation graph. But if it fails due to an inconsistency between the code and the model no error is reported. This means that the user has no idea why the process failed and thus the process of fixing the error is frustrating. Currently pinpointing the exact error is not trivial, it is possible to save the relation graphs with the largest set of relations but this set might contain completely different relation graphs. The question is now which of the relation graphs shows an error, and which of them failed because they have incorrect relations in between the states.

6.2.3 Model extension

Currently only one weak action in the CFG can be assigned to one action in the model. This means that if an action is present twice in the code, it also needs to be present twice in the model in order to properly place all the update rules. As this is not common in LTS a way to automatically duplicate the actions allows constructing a STS with both paths.

6.2.4 Merging internal actions

A trace with many internal actions, means that there are several possible variations on creating the relations in between the states. This in turn leads to many solutions in the output of the algorithm and a long processing time. To speed up the algorithm it is possible under certain conditions to merge internal actions, and thus remove possible variations in the assignments. In the end this will speed up the algorithm as less variations will need to be evaluated.

6.2.5 Formalize a proof for Theorem 4.2.3

An informal sketch of the proof is given, but this should be detailed in order to formally prove the theorem.

Bibliography

- [1] A. Belinfante. Jtorx: A tool for on-line model-driven test derivation and execution. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 266–270. Springer, 2010.
- [2] S. Khurshid C.S. Păsăreanu K. Sen N Tillmann C. Cadar, P. Godefroid and W. Visser. Symbolic execution for software testing in practice: preliminary assessment.
- [3] M. Yannakakis D. Lee. Principles and methods of testing finite state machines—a survey. *Proceedings of the IEEE*, 84(8):1090–1123, 1996.
- [4] Mark Harman and Robert Hierons. An overview of program slicing. *Software Focus*, 2(3):85–92, 2001.
- [5] T. Parr J. Bovet. Antlrworks: an antlr grammar development environment. *Software: Practice and Experience*, 38(12):1305–1332, 2008.
- [6] T. Willemse L. Frantzen, J. Tretmans. Test generation based on symbolic specifications. *Formal Approaches to Software Testing*, pages 1–15, 2005.
- [7] T.A.C. Willemse L. Frantzen, J. Tretmans. A symbolic framework for model-based testing. In *Formal approaches to software testing and runtime verification*, pages 40–54. Springer, 2006.
- [8] T Parr. The definitive antlr reference: Building domain-specific languages (pragmatic programmers). *Pragmatic Bookshelf*, May, 2007.
- [9] T Parr. Antlr. <http://www.antlr.org/>, 2013.
- [10] J. Tretmans. Model based testing with labelled transition systems. *Formal methods and testing*, pages 1–38, 2008.
- [11] J.C. Cherniavsky W.R. Adrion, M.A. Branstad. Validation, verification, and testing of computer software. *ACM Computing Surveys (CSUR)*, 14(2):159–192, 1982.
- [12] M. Young. *Software Testing and Analysis: Process, Principles, and Techniques*. John Wiley & Sons, 2008.