University of Twente
Faculty of EEMCS
Formal Methods and Tools
P.O. Box 217, 7500 AE Enschede
The Netherlands

SeCo

A Tool for Semantic Test Coverage

by

Jeroen Mengerink
j.mengerink@gmail.com
August 15, 2008

Committee

dr. M.I.A. Stoelinga ir. A.F.E. Belinfante dr.rer.nat. M. Weber

Abstract

This thesis aims at implementing a tool to calculate semantic coverage and select test suites. With this tool called SeCo, a set of tests can be selected based on the specification of a system according to the theory in Laura Brandán Briones' PhD thesis. Coverage measures are important, because they measure the quality of a test suite. We use semantic coverage measures, because these measures assign the same coverage to behaviorally equivalent systems. The specifications and tests that can be processed by SeCo are labeled transition systems in the Aldebaran format. SeCo is implemented in Java and able to process algorithms like:

- merging tests into a test suite
- calculating coverage for supplied test suites and specifications
- selecting tests according to coverage measures

A small part of the tool is implemented in Maple. This part uses matrix calculations, which are hard to implement in Java.

Table of Contents

1 Introduction

1.1 Motivation

Testing is mostly incomplete, so it is important to be able to do test selection. This thesis describes the design of a tool, SECO, used for test coverage and test selecting. Since coverage measures evaluate the quality of a test suite, the coverage values can aid in selecting the best tests or test suites. The reason for using semantic coverage is that this type of coverage assigns the same coverage value to behaviorally equivalent systems. The algorithms that are used for both the test coverage and the test selection have been developed by Laura Brandán Briones (Brandán Briones, Brinksma, & Stoelinga, 2006; Brandán Briones, 2007).

SECO is a tool, written in Java and Maple, that allows the user to execute the algorithms for semantic coverage and test selection as developed by Laura Brandán Briones. SECO is capable of:

- Merging tests
- Semantic coverage
- Test selection

SECO consists of seven main algorithms.

Merge combines tests into a special form of test suite. This suite allows inputs and outputs from the same state, and multiple inputs from the same state.

Then we have three algorithms for coverage.

- Absolute coverage
- Total coverage
- Relative coverage

Absolute coverage is the value we can calculate that indicates the coverage of a given test suite. The *total coverage* is the maximum coverage that can be achieved for a given specification and *relative coverage* is the absolute coverage of a test suite as a percentage of the total coverage.

The optimization part also contains three algorithms:

- Optimal coverage in a test case
- Optimal coverage in n test cases
- Optimal coverage in a test suite (with n test cases)

When given a specification, the first optimization algorithm returns the single test case for this specification that gives the highest coverage. The second algorithm returns the best n test cases for the specification. The last algorithm returns the test suite of n cases that gives the highest coverage value.

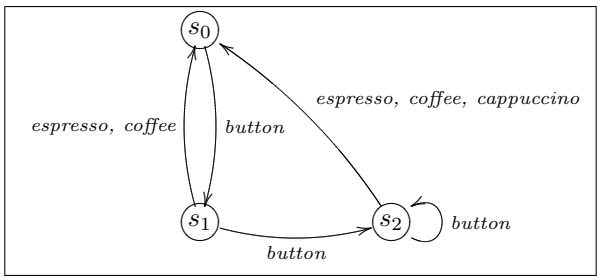


Figure 1.1: Specification of the coffee example

To evaluate the usefulness of the algorithms concerning semantic coverage, we compare this coverage to mutant coverage. Mutant coverage is also a type of semantic coverage and therefore a nice coverage measure to compare to. Mutant coverage is defined as the percentage of discovered IOCO incorrect mutants, where mutants are variations on the specification.

We used the simple specification of a coffee machine (Figure 1.1) to compare these two types of coverage. Six mutants of this specification already existed. We created four more to have more variance in the mutant coverage value. Sixteen test suites were generated for testing the mutants. To calculate the semantic coverage for these test suites, we used four sets of discount values. Discount values decrease the severity of an error according to the depth it is on. When an error happens very soon it is marked more seriously. The error weights that we used for the errors are all equal, this to give all mutants the same basic severity. When starting the tests it was not know which errors (if any) were present in the mutants.

None of the test suites marked correct implementations as incorrect. Three of the IOCO incorrect mutants were found by all of the suites. Indicating that the error in these mutants was quite easy to find. The rest of the mutants were found by a couple of the suites. The results of the tests showed that for discount values near to $\frac{1}{\text{outdegree}(s)}$ the semantic coverage was very low. When we took smaller discount values (indicating a larger discount) the semantic coverage value was

closer to the mutant coverage value. It also became clear that whenever the semantic coverage value increases, the mutant coverage value also increases. Indicating that adding tests or using better tests, might cover more of the specification, but does not always find more errors.

Semantic coverage proves to be a correct way to indicate the usefulness of a test suite, but I advise to look more into loops within the specification. Since loops tend to blow up the number of test cases that can be generated, where no new behavior of the system is tested.

1.2 Related Work

1.2.1 Background

A lot previous work on coverage has been done. Definitions on coverage have changed during the development of coverage measures. In (Williams & Sunter, 2000), coverage is defined as the number of faults detected divided by the number of potential faults. Now we define that even if no faults are detected there is coverage. Since with every time we successfully pass through a system, we are more assured that it is correct.

We see two main streams in coverage, *code coverage* and *specification coverage*. Code coverage is the type of coverage that is described in the majority of the papers about coverage. It defines coverage based on the implementation of a system. We can check which percentage of the code has been executed. The main coverage criteria for code coverage are:

- **Function coverage** - Has each function in the program been executed?
- **Statement coverage** - Has each line of the source code been executed?
- **Condition coverage** (also known as **Branch coverage**) - Has each evaluation point (such as a true/false decision) been executed?
- **Path coverage** - Has every possible route through a given part of the code been executed?
- **Entry/exit coverage** - Has every possible call and return of the function been executed?

The specification coverage can be split into state coverage and transition coverage. Where state coverage checks the number of states visited relative to the total number of states, transition coverage the number of transitions that have been used relative to the total number of transitions.

Next to these two main streams, we find a couple of other theories. The theory that is the basic for SeCo is about semantic coverage and is described in (Brandán Briones et al., 2006). This is a coverage measure that gives the same value for IOCO implementations of a system. Recently Mark Timmer developed a new theory about actual coverage (Timmer, 2008). It deals not only with test cases or test suites, but also with the number of executions planned and the probabilistic behavior of the system.

1.2.2 Tools

Many tool for coverage are available. To see which other tools we have, we used Google. For code coverage tools, we get more then 500.000 hits. We will discuss three programs with multiple hits.

- **Tcov** (Sun Microsystems, 2008) is an important tool that is used for code coverage. It produces a test coverage analysis of a compiled program. Tcov takes source files as arguments and produces an annotated source listing. Each basic block of code is prefixed with the number of times it has been executed. A basic block is a contiguous section of code that has no branches: each statement in a basic block is executed the same number of times.

- **EMMA** (Roubtsov, 2006) is a toolkit for measuring and reporting Java code coverage. EMMA supplies rich coverage analysis data without introducing significant overhead during either build or execution time. It also supports quick development and testing of small stand alone Java applications as well as massive enterprize software suites containing thousands of Java classes.

- **Clover** (Dawson, 2008) is also a Jave code coverage tool, providing detailed reports of method, statement and branch coverage. The nice features in Clover are per-test coverage, showing which test hit which statements, and identification of 'dead' code.

For specification coverage tools we have less hits, a little above 2.000. This number is very deceptive, since there are lots of links to the same programs. We will give an description of the two main tools for specification coverage.

- **Microsoft's Spec Explorer** (Campbell et al., 2005) is a leading tool in model-based testing. The core ideas behind Spec Explorer are to encode the system's intended behavior in a model program. This program only specifies what the system must and must not do. Then Spec Explorer can explore the possible runs of the model program and systematically generate test suites. In addition to testing, Spec Explorer also has abilities for calculating optimal strategies for testing nondeterministic systems (Nachmanson, Veanes, Schulte, Tillmann, & Grieskamp, 2004).

- **ADLscope** (Chang & Richardson, 1998) is an automated specification-based unit testing tool. It guides the testing activity based on an ADL specification of the units to be tested. ADL is a formal specification language developed at Sun Microsystems Laboratories to describe behavior of functions in the C programming language. ADLscope measures coverage of program units with respect to their ADL specifications. Any uncovered coverage condition reported to the tester usually means that certain aspects of the specification have not been tested adequately. This forces the tester to select additional test data to cover those aspects. ADLscope is most useful for conformance testing of an API. Many code-based coverage techniques require the source code of the implementation, which is often not available for conformance testing because of proprietary issues, whereas ADLscope only requires the object code or compiled libraries.

1.3 Organization of the Thesis

This thesis is organized into eight main chapters including this one. Chapter 2 explains some theoretic background needed to understand the contents presented in this thesis. In Chapter 3 we present an introduction SeCo. This chapter contains which file formats and data structures we used and the first algorithm with its implementation, *Merge*. Chapter 4 discusses the algorithms concerned with semantic coverage and their implementations.

- Absolute coverage
- Total coverage
- Relative coverage

The algorithms and implementations concerning test selection are presented in Chapter 5.

- Optimal coverage in a test case
- Optimal coverage in n test cases
- Optimal coverage in a test suite (with n test cases)

We present three extra features that have been implemented in Chapter 6.

- Explode - to make a test from a trace
- GenerateDiscounts - to generate a file representing a discount function
- GenerateMatrix - to generate Maple code for the total coverage algorithms

Chapter 7 deals with the comparison between semantic coverage as we use it and mutant coverage. Finishing with Chapter 8, where we present conclusions about semantic coverage together with some concluding remarks.

2

Background on Semantic Coverage

2.1 Introduction

The main focus of this thesis is implementing the algorithms about semantic test coverage and test selection as described in (Brandán Briones et al., 2006). To understand the algorithms that are used, it is needed to know the basic notations that will be used. The algorithms that are implemented deal with *weighted fault models*, *test cases* and *coverage measures*.

Weighted fault models are specifications which describe the severity of erroneous traces next to the normal behavior of the system.

The coverage measures we use abstract from the exact shape of test cases and test suites. The original tests for labeled input-output transition system state that at every moment you either get an input or all outputs. The change we made is that you can also get multiple inputs or inputs and outputs. Actually meaning the we use traces in stead of tests and trace sets in stead of test suites.

Since we need to calculate on the weighted fault models, it is needed to make them finite. To do this we provide two options: *finite depth* and *discounted*.

Finite depth achieves a finite weighted fault model by considering finitely many traces. It cuts of the infinite tree of traces at a given depth.

Discounted weighted fault models take in account the error weights of all traces, but discount the weight of a trace proportional to its length. Discount values need to be between zero and one to make sure the keep lowering the error weight. In this way a finite set of traces has a value greater than zero.

Organization of the Chapter

We start in Section 2.2 with the description and definition of *weighted fault models*. The next section deals with test cases in labeled input-output transition systems. We first define what labeled input-output transition systems are and after that we define what test cases for labeled input-output transition systems are. Section 2.4 defines coverage measures we mean to implement in SeCo.

The last section of this chapter defines fault automata and two ways to derive a weighted fault model from a fault automaton.

2.2 Weighted Fault Models

A weighted fault model specifies the desired behavior of a system. This model does not only describe the correct system traces, but it also describes the severity of erroneous traces.

Definition 1. *A weighted fault model (WFM) over L is a function $f : L \to \mathbb{R}^{\geq 0}$ such that*

$$0 < \sum_{\sigma \in L^*} f(\sigma) < \infty$$

Thus, a WFM f assigns a non-negative error weight to each trace in $\sigma \in L^*$. For correct traces $f(\sigma) = 0$, if $f(\sigma) > 0$ the trace σ represents incorrect behavior and $f(\sigma)$ denotes the severity of the error. The higher the value of $f(\sigma)$, the worse the error. The total error weight, $\sum_{\sigma \in L^*} f(\sigma)$, must be finite and non-zero in order to define coverage measures relative to the total error weight.

2.3 Test Cases in Labeled Input-Output Transition Systems

This section deals with some basic theory about test derivation for labeled input-output transition systems (LTS) following the IOCO testing theory (Tretmans, 1996).

2.3.1 Labeled Input-Output Transition Systems

A labeled input-output transition system (LTS) is a system that interacts with its environment through inputs and outputs. Input actions are driven by the environment, where output action are actions to the environment. Internal actions are not observable by the environment. We use L to denote the action set.

Definition 2. *A labeled input-output transition system (LTS) $\mathcal{A}$ is a tuple $\langle S, s^0, L, \Delta \rangle$, where*

- *S is a finite set of states*
- *$s^0 \in S$ is the initial state*
- *L is a finite action alphabet. We assume that $L = L^I \bigcup L^O$ is partitioned (i.e. $L^I \bigcap L^O = \emptyset$ into a set L^I of input labels (also called input actions or inputs) and a set L^O of output labels (also called output actions or outputs). We denote elements of L^I by $a?$ and elements of L^O by $a!$*
- *$\Delta \subseteq S \times L \times S$ is the transition relation. We require Δ to be deterministic, i.e. if $(s, a, s'), (s, a, s'') \in \Delta$, then $s' = s''$. The input transition relation Δ^I is the restriction of Δ to $S \times L^I \times S$ and the output*

transition relation Δ^O is the restriction of Δ to $S \times L^O \times S$. We write $\Delta(s) = (a, s')|(s, a, s') \in \Delta$ and similarly for $\Delta^I(s)$ and $\Delta^O(s)$. We denote by $outdeg(s) = |\Delta(s)|$ the out-degree of state s, i.e. the number of transitions leaving s

We denote the components of A by S_A, s_A^0, L_A and Δ_A. We omit the subscript A if it is clear from the context.

2.3.2 Test Cases

A test case is a trace through the system, ending in pass or fail. From each state in the trace, we can either get an input or the set of all outputs defined for the system.

Definition 3.

- *A test case (or test) t for a LTS A is a finite, prefix-closed subset of L_A^* such that*
  - *if $\sigma a? \in t$, then $\sigma b \notin t$ for any $b \in L$ with $a? \neq b$*
  - *if $\sigma a! \in t$, then $\sigma b! \in t$ for all $b! \in L^O$*
  - *if $\sigma \notin traces_A$, then no proper suffix of σ is contained in t*

  We denote the set of all tests for A by $T(A)$.
- *The length $|t|$ of test t is the length of the longest trace in t, i.e. $|t| = max_{\sigma \in t}|\sigma|$. We denote by $T^k(A)$ the set of all tests for A with length k.*

2.4 Coverage Measures

Coverage measures in this thesis abstract from the exact shape of test cases and test suites. Given a WFM f over action alphabet L, we only use that a test is a trace set, $t \subseteq L^*$; and a test suite is a collection of trace sets, $T \subseteq \mathcal{P}(L^*)$. In this way we define the absolute and relative coverage with respect to f of a test and for a test suite. Moreover, our coverage measures apply in all settings where test cases can be characterized as trace sets (in which case test suites can be characterized as collections of trace sets). This is true for tests in TTCN (Grabowski et al., 2003), IOCO test theory (Tretmans, 1996) and FSM testing (Ural, 1992; Lee & Yannakakis, 1996).

Definition 4. *Let $f : L^* \to \mathbb{R}^{\geq 0}$ be a WFM over L, let $t \subseteq L^*$ be a trace set and let $T \subseteq \mathcal{P}(L^*)$ be a collection of trace sets. We define:*

- $abscov(t, f) = \sum_{\sigma \in t} f(\sigma)$ *and* $abscov(T, f) = abscov(\bigcup_{t \in T} t, f)$
- $totcov(f) = abscov(L^*, f)$
- $relcov(t, f) = \frac{abscov(t,f)}{totcov(f)}$ *and* $relcov(T, f) = \frac{abscov(T,f)}{totcov(f)}$

2.5 Fault Automata

Weighted fault models (WFMs) are infinite, semantic objects. Here we introduce *fault automata*, which provide a syntactic format for specifying WFMs. A fault automaton is a LTS A augmented with a state weight function r. The LTS A is the behavioral specification of the system, i.e. its traces represent the correct behavior of the system. Hence, these traces will be assigned the error wight 0; traces not in A are erroneous and get an error weight through r according to Definition 5.

Definition 5. *A fault automaton (FA) F is a pair $\langle A, r\rangle$, where $A = \langle S, s^0, L, \Delta\rangle$ is a LTS and $r : S \times L^O \to \mathbb{R}^{\geq 0}$. We require that, if $r(s, a!) > 0$, there is no $A!$-successor of s in F, i.e. there is no $s' \in S$ such that $(s, a!, s') \in \Delta$. We extend r to a function $r : S \times L \to \mathbb{R}^{\geq 0}$ by putting by putting $r(s, a?) = 0$ for $a? \in L^I$ and define $\overline{r} : S \to \mathbb{R}^{\geq 0}$ as $\overline{r}(s) = \sum_{a \in L^O(s)} r(s, a)$. Thus, $\overline{r}$ accumulates the weight of al the erroneous outputs in a state. We denote the components of F by A_F and r_F and leave out the subscripts F if it is clear from the context. All the concepts that have been defined for LTSs will be used for FAs too.*

2.5.1 Finite Depth Weighted Fault Models

The finite depth model derives a WFM from an FA F, for a given $k \in \mathbb{N}$, by ignoring all traces of length longer than k, i.e. by putting their error weight to 0. For all other traces, the weight is obtained by the function r. If σ is a trace of F ending in s, but $\sigma a!$ is not a trace in F, then $\sigma a!$ gets weight $r(s, a!)$.

Definition 6. *Given a FA F and a number $k \in \mathbb{N}$, we define the function $f_F^k : L^* \to \mathbb{R}^{\geq 0}$ by*

$$f_F^k(\varepsilon) = 0 \quad f_F^k(\sigma a) = \begin{cases} r(s,\ a), & \text{if } s \in reach_F^l(\sigma) \wedge a \in L^O \wedge l \leq k \\ 0, & \text{otherwise} \end{cases}$$

2.5.2 Discounted Weighted Fault Models

Where finite depth WFMs achieve finite total coverage by considering finitely many traces, discounted WFMs take into account the error weights of all traces. To achieve this, only finitely many traces may have an error weight greater than ε, for any $\varepsilon > 0$. This can be done by discounting; lowering the weight of a trace proportional to its length.

To apply the discounting, we use a discount function $\alpha : S \times L \times S \to \mathbb{R}^{\geq 0}$ that assigns a positive weight to each transition of F. Then we discount the trace $a_1 \dots a_k$ obtained from the path $s_0 a_1 s_1 \dots s_k$ by $\alpha(s_0, a_1, s_1)\alpha(s_1, a_2, s_2) \dots \alpha(s_{k-1}, a_k, s_k)$.

Definition 7. *Let F be a FA, $s \in S$ and α a discount function for F. We define the function $f_F^\alpha : L^* \to \mathbb{R}^{\geq 0}$ by:*

$$f_F^{\alpha}(\varepsilon) = 0$$

$$f_F^{\alpha}(\sigma a) = \begin{cases} \alpha(\pi) \cdot r(s,a) & \text{if } s \in reach_F(\sigma) \wedge a \in L^O \wedge trace(\pi) = \sigma \\ 0 & \text{otherwise} \end{cases}$$

3 Introduction to SeCo

3.1 Introduction

SeCo is a tool used to calculate semantic coverage and perform test selection. Several algorithms for doing this are presented in the dissertation of Laura Brandán Briones (Brandán Briones, 2007). To use these algorithms in practice, we developed SeCo. For the coverage part of the tool, we have three modules:

- **AbsCov** - for calculating the absolute coverage of a test suite
- **Total coverage** = for calculating the total coverage of a specification
- **RelCov** - for calculating the relative coverage of a test suite with respect to the total coverage of the specification

The test selection part of the tool consists of the following three modules:

- **Optimize** - for getting the best test of a certain depth for a specification
- **OptNTests** - for getting the best n tests of a certain depth for a specification
- **OptSuite** - for getting the best test suite of a certain depth for a specification

These modules will be explained into more detail in respectively Chapter 4 and Chapter 5. In the chapter after that, we will present modules that helped us to quickly get input files or other inputs.

- **Explode** - to make a test from a trace
- **GenerateDiscounts** - to generate a file representing a discount function
- **GenerateMatrix** - to generate Maple code for the total coverage algorithm

Before we can start explaining all the above mentioned modules, we first need to know how we present the input to the modules and which data structures we use. Therefore we present the file types that are used as inputs for SeCo in this chapter, along with data structures we use to interpret the files.

The first algorithm and it's implementation (*Merge*) will be presented in this chapter too. *Merge* is used to combine several tests into test suite. Here we will encounter the example that we will use throughout the thesis. We will be using the simple specification of a coffee machine as shown in Figure 3.1

An overview of all these modules is presented in Figure 3.2. Notice that total coverage has not been implemented in Java like the rest of the tool, but in Maple. We explain more about Maple in the section that covers total coverage (Section 4.3).

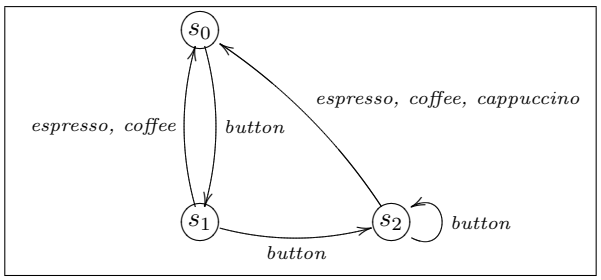


Figure 3.1: Specification of the coffee example

Organization of the Chapter
We start this chapter with explaining the file formats and data structures we use to represent them in Section 3.2. Then we will explain the module *Merge* in Section 3.3. This section is divided into four parts.

- Merge algorithm - presents the merge algorithm
- Merge with SeCo - shows how we programmed the module
- Example - clarifies the module with the use of an example
- Implementation issues - informs about some implementation choices

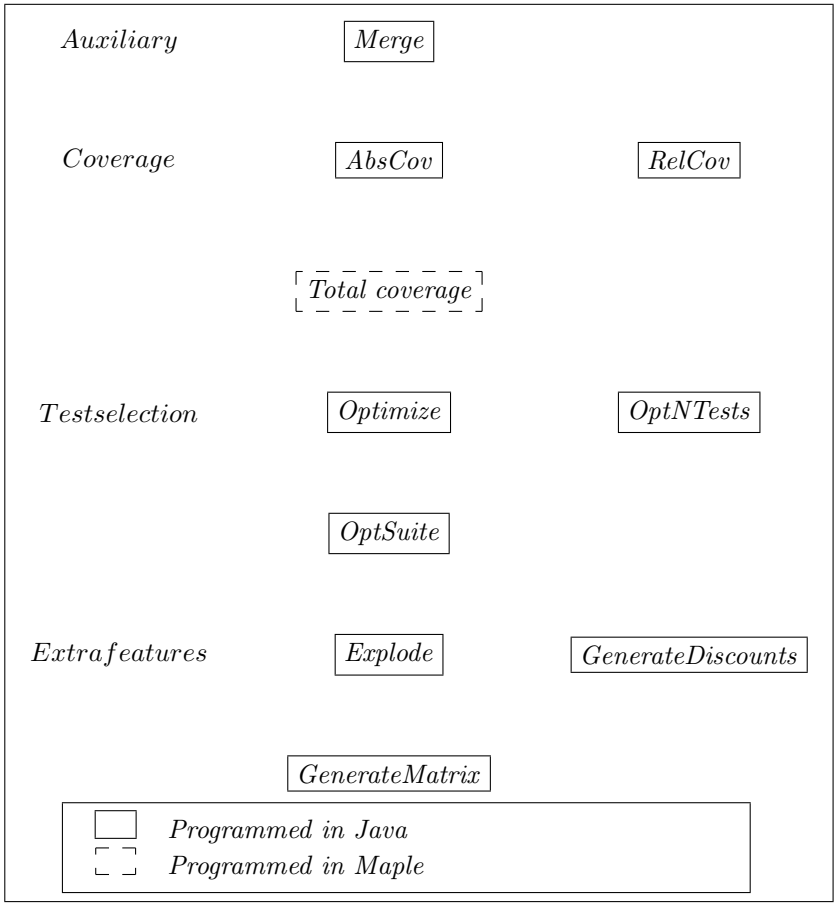


Figure 3.2: Overview of the modules in SeCo

3.2 File Formats and Data Structures

We need to use specifications, tests and test suites as inputs and outputs for SeCo. The format for these files is the Aldebaran format for LTS (.aut), as also used in the tool TorX (Belinfante et al., 1999). To be able to use fault automata (Section 2.5) we introduce two new file formats, the discount file format (.disc) and the error file format (.err). For the ease of use, we based these new file formats on the Aldebaran file format.

Input	**File type**	**Data structure**
Specification, test, Supertest	Aldebaran format (.aut)	Hashtable(state, [state, action, state])
Discount function	Discount file format (.disc)	Hashtable(state, [state, action, state, discount_value])
Error weights	Error file format (.err)	Hashtable(state, [state, action, error_weight])

Table 3.1: Data structures in SeCo

3.2.1 Aldebaran File Format

CADP ("Construction and Analysis of Distributed Processes", formerly known as "CAESAR/ALDEBARAN Development Package") (Garavel et al., 1997) is a popular toolbox for the design of communication protocols and distributed systems. From this package I have taken the Aldebaran format for LTS. A .aut file describes a LTS in the Aldebaran format. Each state is represented by a natural number. A .aut file exists of a descriptor line, followed by lines that describe transitions. The structure of a .aut file is described by the following grammar:

```
AUTFILE : DESCRIPTOR TRANSITIONS;

DESCRIPTOR : DES LPAREN FIRST-STATE COMMA NUMBER-OF-TRANSITIONS
    COMMA NUMBER-OF-STATES RPAREN;
NUMBER-OF-TRANSITIONS : NUMBER; NUMBER-OF-STATES : NUMBER;

TRANSITIONS : (TRANSITION)+; TRANSITION : LPAREN STATE COMMA
ACTION COMMA STATE RPAREN;

DES : 'des';
COMMA : ',';
DQUOTE : '"';
LPAREN : '(';
RPAREN :')';
FIRST-STATE : '0';
NUMBER : (DIGIT)+;
DIGIT : ('0'..'9');
STATE : NUMBER;
ACTION : DQUOTE String DQUOTE;
```

Figure 3.3a depicts a small example file which describes the automaton shown in Figure 3.1.

```
des (0, 8, 3)
(0, "button", 1)
(1, "button", 2)
(1, "espresso", 0)
(1, "coffee", 0)
(2, "button", 2)
(2, "espresso", 0)
(2, "coffee", 0)
(2, "cappuccino", 0)
```

(a) .aut file

Key	Value
0	[”0”, ”button”, ”1”] [”0”, ”delta”, ”0”]
1	[”1”, ”button”, ”2”] [”1”, ”espresso”, ”0”] [”1”, ”coffee”, ”0”]
2	[”2”, ”button”, ”2”] [”2”, ”espresso”, ”0”] [”2”, ”coffee”, ”0”] [”2”, ”cappuccino”, ”0”]

(b) Hashtable

Figure 3.3: Specification file and its data structure

As can be seen in Table 3.1, this file will be read into a vector of string arrays. The result will be a vector of transitions, where the transitions are string arrays containing start state, action and end state. The next step is creating a hash table, where we store the start states as keys and the transitions from the states as values. The hash table can be seen in Figure 3.3b.

Notice that we have one transition more in the hash table than we have in the file. This is because we automatically add delta transitions to specifications.

3.2.2 Discount File Format

This format looks a lot like the Aldebaran file format, but the transitions contain discount values. We use the extension .disc for files in this format. Every line describes an edge and the discount value belonging to that edge. This file format does not contain the descriptor line and the transitions now follow this grammar:

```
TRANSITION : LPAREN STATE COMMA ACTION COMMA STATE COMMA DISCOUNT
RPAREN;

DISCOUNT : DOUBLE;
DOUBLE : NUMBER PERIOD NUMBER;
PERIOD :'.';
```

Figure 3.4 shows the file that describes the automaton from Figure 3.5.

In Table 3.1 we state that this file will be represented as a hashtable with states as keys and transitions as values. The representation will look like the hashtable we present in Figure 3.3b, only with the discount value also present in the transition.

3.2.3 Error File Format

This file format describes the error transitions with their corresponding error weights, we use the extension .err for files with this format. The first line describes the outputs that are present in the specification, including ”delta”. Each other line of the file represents an error transition. The first line will look like:

```
(0, "button", 1, 0.499)
(1, "button", 2, 0.3323333333333333)
(1, "espresso", 0, 0.3323333333333333)
(1, "coffee", 0, 0.3323333333333333)
(2, "button", 2, 0.249)
(2, "espresso", 0, 0.249)
(2, "coffee", 0, 0.249)
(2, "cappuccino", 0, 0.249)
(0, "delta", 0, 0.499)
```

Figure 3.4: Discount file

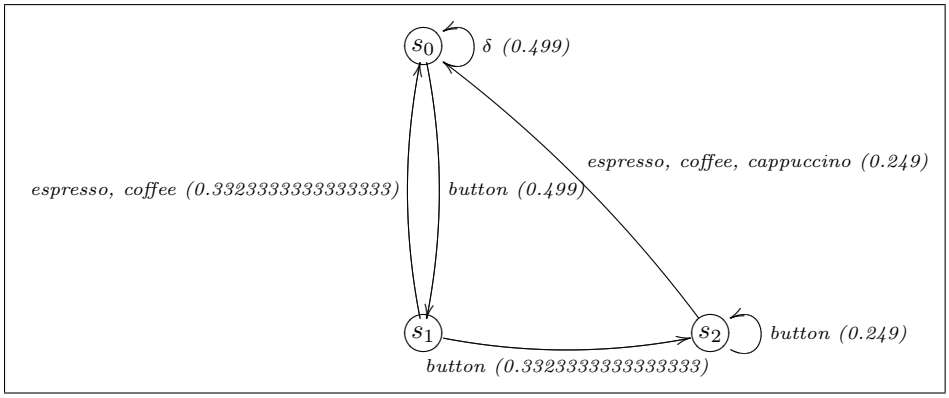


Figure 3.5: Coffee example extended with quiescence and discounts

```
OUTPUTS : LBRACK ACTION (COMMA ACTION)* RBRACK;

LBRACK : '[';
RBRACK : ']';
```

The grammar for an error transition is:

```
TRANSITION : LPAREN STATE COMMA ACTION COMMA ERROR RPAREN;

ERROR : NUMBER;
```

`STATE` is a number or the character '*', if `STATE` is '*', it means that the transition counts for all states that do not have the action `ACTION`. In Figure 3.6 we can see the file that in combination with the specification describes the fault automaton which can be seen in Figure 3.7.

We described that the transitions in the error file can contain '*' in stead of a state. So internally the example file in Figure 3.6 will be expanded to the file in Figure 3.8. After this, we once again represent the file as a hashtable like in Figure 3.3b.

```
 ["delta", "espresso", "coffee", "cappuccino"]
(*, "delta", 1)
(0, "espresso", 1)
(0, "coffee", 1)
(*, "cappuccino", 1)
```

Figure 3.6: Error file

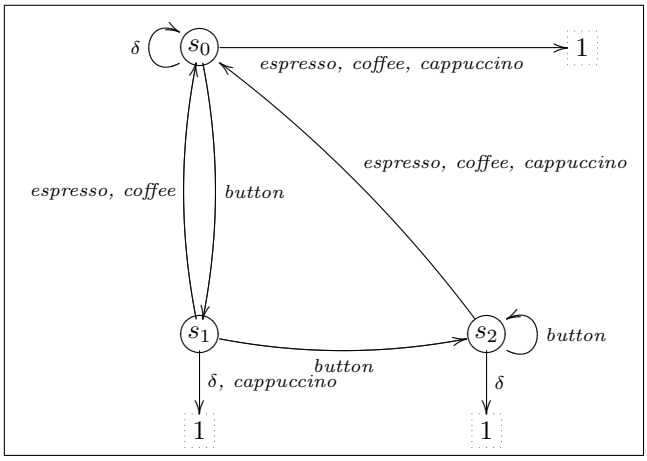


Figure 3.7: Fault automaton of the coffee example

```
 ["delta", "espresso", "coffee", "cappuccino"]
(0, "espresso", 1)
(0, "coffee", 1)
(0, "cappuccino", 1)
(1, "cappuccino", 1)
(1, "delta", 1)
(2, "delta", 1)
```

Figure 3.8: Expanded error file

3.2.4 Fault Automata

We will be working with fault automata which are a combination of a specification (extended with quiescence) and error transitions. Both are supplied to SeCo as input files. The vectors that contain the data form these files are internally used as a fault automaton.

Combining the specification and the error transitions internally is done for ease of use. Most of the time you will already have a specification and no error transitions. In this way you only need to create a list of error transitions to add for the fault automaton and you can keep your original specification.

3.3 Merge

The module *Merge* deals with combining several tests into a *Supertest*. The combined test suite has the possibility of having multiple inputs or inputs and outputs leaving a state. Since tests normally do not allow multiple inputs or inputs and outputs leaving the same state, this suite is called a Supertest.

The test that is an input for the algorithm is represented as a .aut file for the module. Since the module is implemented to merge several tests, it executes the algorithm for each test to be merged. It concatenates the equal parts of the tests and leaves the rest unchanged and then writes the Supertest to a file.

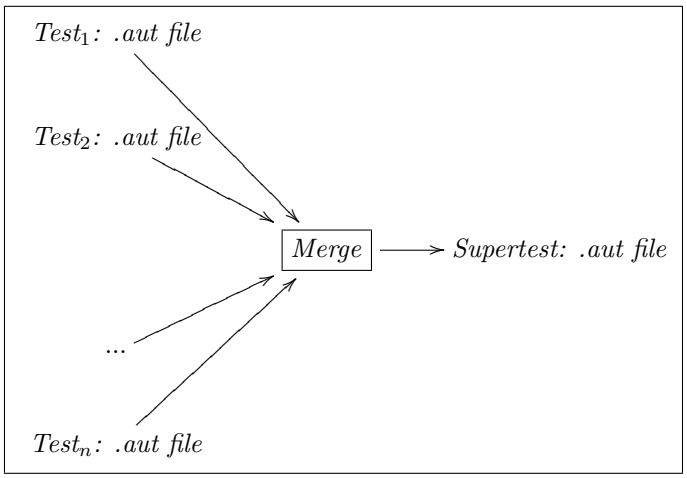


Figure 3.9: Architecture of the Merge module

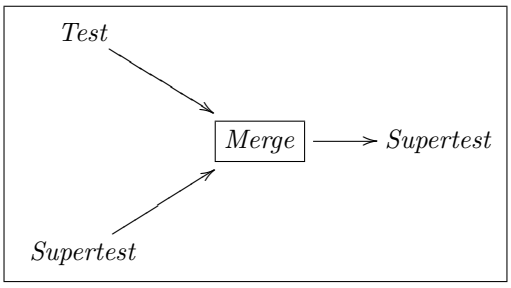


Figure 3.10: Architecture of the Merge algorithm

3.3.1 Merge Algorithm

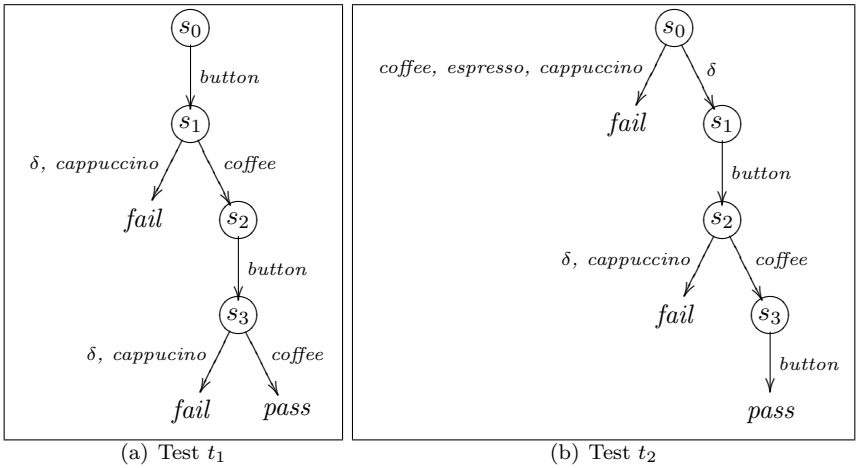
Let $t' = a_1t'_1 + \ldots + a_kt'_k + b_1t'_1 + \ldots + b_nt''_n$ be a Supertest and t be a test. Then $t = \varepsilon$ or $t = at_1$ or $t = b_1t'_1 + \ldots + b_nt'_n$. Then we define:

$$t' \uplus t = \begin{cases} a_1t'_1 + \ldots + a_j(t'_j \uplus t_1) + \ldots + a_kt'_k + b_1t''_1 + \ldots + b_nt''_n & \text{if } t = at_1 \wedge a = a_j; \\ a_1t'_1 + \ldots + a_kt'_k + b_1(t''_1 \uplus t_1) + \ldots + b_n(t''_n \uplus t_n) & \text{if } t = b_1t'_1 + \ldots + b_nt'_n; \\ t' + t & \text{otherwise.} \end{cases}$$

Now we execute the algorithm on the tests t_2 and t_3 of Figure 3.11, seeing t_2 as the current Supertest. The first step in merging the Supertest (t_2) with test t_3 is checking the transitions from state s_0. These are of the type considered in the second line of the algorithm (outputs). So the algorithms first step will make a Supertest as can be seen in Figure 3.12.

s_1 now has a dotted line, meaning that this state is the start state for the next step. The next step is once again the second line of the algorithm, this time the outputs are not yet in the Supertest. The result of step 2 can be seen in Figure 3.13.

The final step is made by using the third line of the algorithm. Since the transition to be added is an input, it should apply to the first rule, but the current state of the Supertest does not have any transitions. Therefore the rest of the test is now added. The result of this final step can be seen in Figure 3.14.



(a) Test t_1

(b) Test t_2

(c) Test t_3

Figure 3.11: Example tests

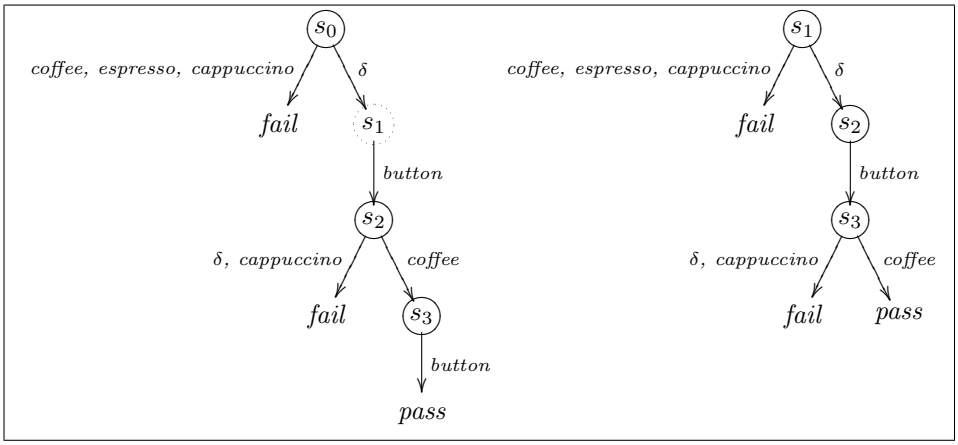


Figure 3.12: Merged with algorithm: step 1

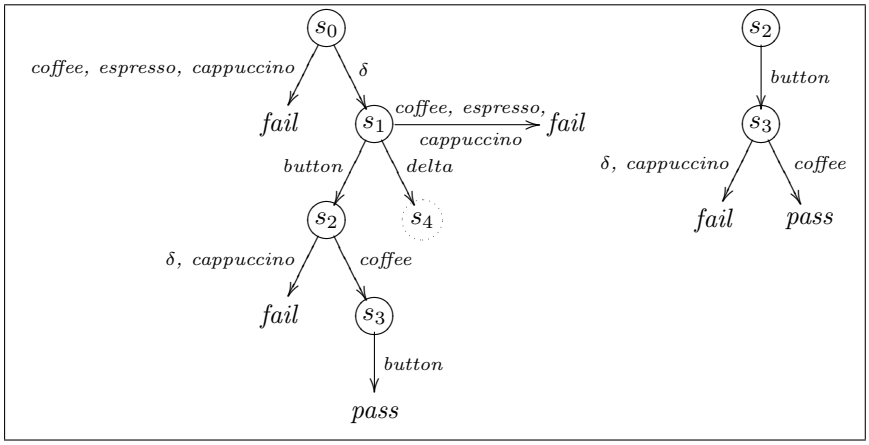


Figure 3.13: Merged with algorithm: step 2

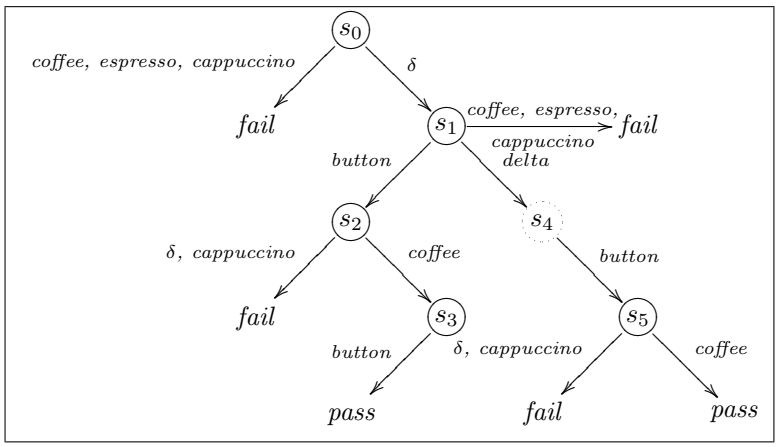


Figure 3.14: Merged with algorithm: step 3

3.3.2 Merge with SeCo

Merge should get at least three command line arguments, two tests and a filename for the resulting Supertest, when using the class.

Two methods perform the merge algorithm: `doMerge` and `addSubtree`. `doMerge` gets a test t and a Supertest S. The initial Supertest is provided by the constructor, which used the first test that was given as an input as the initial Supertest. Now `doMerge` acts in the following manner:

`For test` t `and Supertest` S`, both starting at state 0:`

1. `Get all transitions from current state of` t`.`
2. `For each transition check if it is an transition from the current state of` S`.`
3. `If it is, goto 1 with the next state of` t `and` S`.`
4. `If it is not, call addSubtree.`

When `addSubtree` is called, it adds the current transition and all the transitions following it to the Supertest.

3.3.3 Example

Using the coffee specification from Figure 3.1, we derived the tests shown in Figure 3.11. When *Merge* is used to combine these three tests, it results in the Supertest in Figure 3.17. This result can be obtained by the following command, where the files Test1.aut, Test2.aut and Test3.aut are the test files (Appendix A):

```
java main.auxiliary.Merge Test1.aut Test2.aut Test3.aut
                          Test123.aut
```

Merge will use t_1 as the initial Supertest. After that it will start to add t_2 to this Supertest. Starting in state "0", it checks all transitions from this state of t_2. Coffee, espresso and cappuccino are not in the Supertest yet, so they will be added along with their subtree (which in this case is empty). This step is shown in Figure 3.15. The other transition from this state, δ, is also not present in the current Supertest, but this transition does lead to a subtree. The addition of this transition and its subtree can be seen in Figure 3.16. Now no transitions are left in the current test, so it is time to add t_3 to the Supertest.

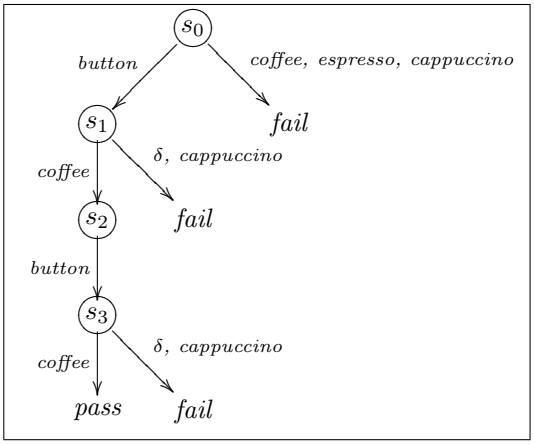


Figure 3.15: Merging of tests with SeCo: step 1

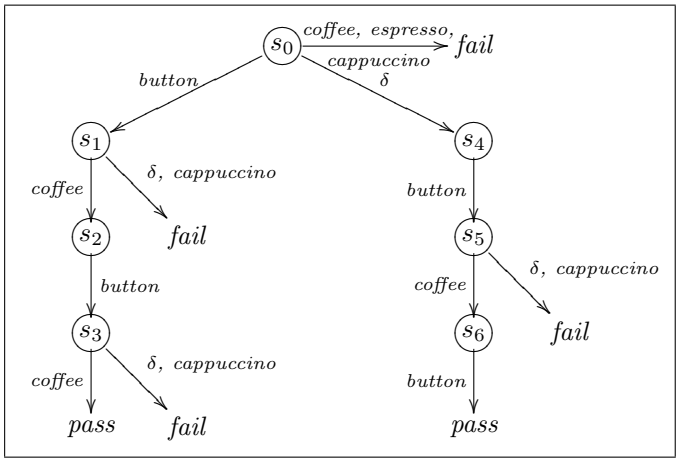


Figure 3.16: Merging of tests with SeCo: step 2

Once again start at state "0" and check all transitions. This time coffee, espresso and cappuccino are already in the Supertest. So we recurse to the next state and start the algorithm. The next state has no outgoing transitions, therefore this part of the transitions is ready. The other transition from state "0" is now evaluated. This also exists in the Supertest, so recurse. We are

now in state s_4 of the current Supertest and in state s_1 of t_3. Here we check all transitions again. The transitions coffee, espresso and cappuccino from this state are not in the Supertest yet, so they will be added along with their subtree (which in this case is empty). The other transition from this state, δ, is also not present in the current Supertest, this transition does lead to a subtree which needs to be added. This results in the Supertest in Figure 3.17. All transitions of the tests that needed to be merged are now in the Supertest. So the module writes this Supertest to the file `Test123.aut`.

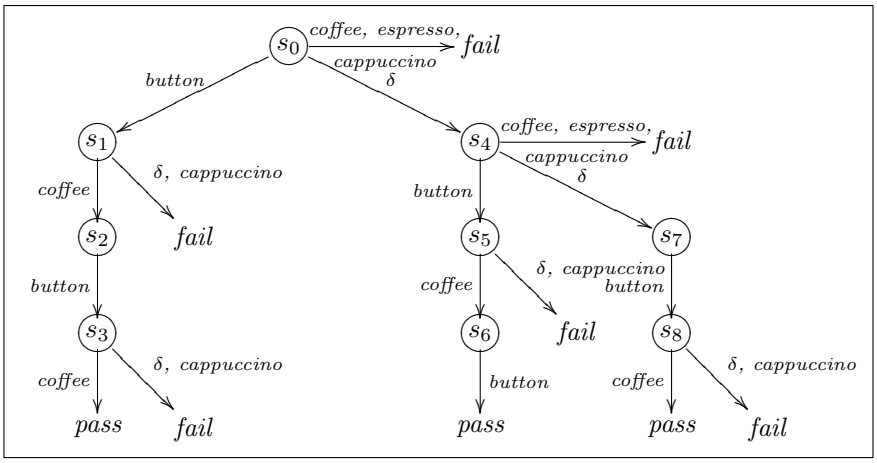


Figure 3.17: Supertest

3.3.4 Implementation Issues

All the transitions in the specification and tests are implemented as arrays of Strings. There might be better ways of implementing the transitions, but the String arrays are easy to handle.

SeCo does not use the descriptor line of the .aut files, in stead it calculates the number of transitions and states itself. The reason not to use this line is that a couple of files that we used had errors in that line. Wrong numbers of states and transitions were given in the descriptor, which messed up for-loops that were used in SeCo.

4 Coverage with SeCo

4.1 Introduction

Semantic coverage has the main focus for this thesis. This chapter will deal with the algorithms for semantic coverage and the implementation of these algorithms. We will present three types of coverage:

- Absolute coverage
- Total coverage
- Relative coverage

Absolute coverage can be calculated for Supertests (Section 3.3, giving a numeral indication of how much of the specification is covered by the test suite.

Total coverage gives the maximal coverage value for a specification. This part of SeCo has been implemented in Maple, a mathematics software package. We can calculate total coverage with two different algorithms:

- Total coverage in discounted FA
- Total coverage in finite depth FA

The difference in the algorithms is the way to make the FA (seem) finite.

The last coverage algorithm is *relative coverage*. This will return the absolute coverage of test suite relative to the total coverage of the specification.

Organization of the Chapter
In Section 4.2 we will describe the module *AbsCov* which is responsible for calculating the absolute coverage of a Supertest. In this section we start with a small introduction about absolute coverage, followed by the absolute coverage algorithm. After that, we will explain how we modeled the algorithm in SeCo and present an example of how the tool works.

The next section will provide information about total coverage. We start with a small introduction to total coverage and explain the two types of total coverage:

- Total coverage in discounted FA
- Total coverage in finite depth FA

Since this part of the tool is programmed in Maple, we will explain what Maple is an which parts of Maple we use. Next we will provide the algorithms for the two types of total coverage and we finish the section with an example where we will show how we programmed the algorithms in Maple.

We finish this chapter with a section about relative coverage. In this section we start with a small introduction about relative coverage, followed by the relative coverage algorithm. After that, we will explain how we modeled the algorithm in SeCo and present an example of how the tool works. We finish this section with an error we encountered during programming and how we resolved it.

4.2 Absolute Coverage of a Supertest

Absolute coverage of a Supertest is an indication of how much of the specification has been covered by the Supertest. This taking into account the severity of the errors that can be found and optionally the depth on which the errors are found. The inputs for the algorithm that calculates this value (Section 4.2.1) are a Supertest, a fault automaton (Section 2.5), a state on the fault automaton and optionally a discount function (Section2.5.2). The algorithm returns a value in $\Re$.

The inputs for the module, called *AbsCov* are a .aut file containing the Supertest, a .aut file containing the specification, a file containing the errors in the error file format and optionally a file containing discounts in the discount file format. The result is presented as a double.

This differs from the inputs of the algorithm. The algorithm takes a fault automaton, where the module takes a specification and a set of error transitions. These two inputs are internally combined into a fault automaton. The set of discount transitions correspond to the discount function. The module always starts with state "0" as the initial state that is needed for the algorithm.

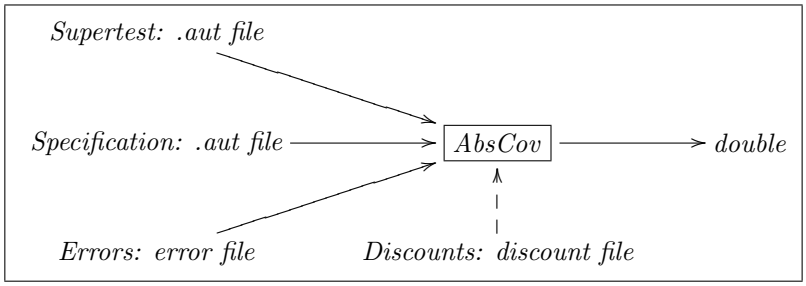


Figure 4.1: Architecture of the AbsCov module

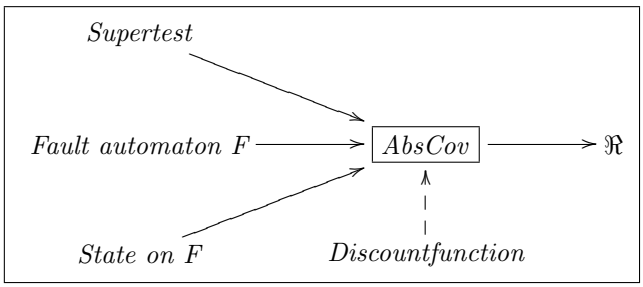


Figure 4.2: Architecture of the absolute coverage algorithm

4.2.1 Absolute Coverage Algorithm

Given a Supertest t, a fault automaton F, a state s on F and a discount function α for F , then

$$ac(\varepsilon, s) = 0$$

$$ac(t, s) = \sum_{i=1}^{n} aux(a_i t_i, s)$$

$$\text{where } aux(a_i \cdot t_i, s) = \begin{cases} \alpha(s, a_i, \delta(s, a_i)) \cdot ac(t_i, \delta(s, a_i)) & \text{if } a_i \in \delta(s) \\ r(a_i, s) & \text{otherwise} \end{cases}$$

When executing the algorithm for the Supertest we created with *Merge* (Figure 3.17), the fault automaton from Figure 3.7, the state s_0 and discount function α (Figure 4.3), the following will happen.

$s_0 \times button \times s_1 \rightarrow 0.499$
$s_0 \times \delta \times s_0 \rightarrow 0.499$
$s_1 \times button \times s_2 \rightarrow 0.3323333333333333$
$s_1 \times coffee \times s_0 \rightarrow 0.3323333333333333$
$s_1 \times espresso \times s_0 \rightarrow 0.3323333333333333$
$s_2 \times coffee \times s_0 \rightarrow 0.249$
$s_2 \times espresso \times s_0 \rightarrow 0.249$
$s_2 \times cappuccino \times s_0 \rightarrow 0.249$
$s_2 \times button \times s_2 \rightarrow 0.249$

Figure 4.3: Discount function α for the coffee example

We start at node s_0 of the Supertest:
$ac(t, s_0) = \sum_{i=1}^{5} aux(a_i t_i, s_0)$
Where every a_i represents an action from state s_0. When we write out this sum, we get:
$ac(t, s_0) = 0.499 \cdot ac(t_1, s_1) + 1 + 1 + 1 + 0.499 \cdot ac(t_5, s_4)$
The three transitions leading to the fail state deliver their error weight, where the two other transitions multiply the subtree beneath them with their discount

value. We will only show the left path of the tree and leave the rest to the reader. The left path recurses as follows:
$ac(t_1, s_1) = 0.3323333333333333 \cdot ac(t1_1, s_2) + 1 + 1$
$ac(t1_1, s_2) = 0.499 \cdot (t1.1_1, s_3)$
$ac(t1.1_1, s_3) = 0.3323333333333333 \cdot 0 + 1 + 1$
Resulting in $0.499 \cdot (0.3323333333333333 \cdot (0.499 \cdot 2) + 2) \approx 1.164$
The absolute coverage for the total Supertest is approximately 6.407.

4.2.2 Absolute Coverage with SeCo

The module *AbsCov* should get three or four command line arguments, a Supertest, the specification, a file containing error transitions and optionally a file containing discount transitions. If no discounting is specified for a transition in the specification, SeCo will assume discount value 1 for that transition.

With SeCo we edited the algorithm a little bit by not using the error transitions separately, but by taking the accumulated error weight for a state. The error transitions therefore need not be in the Supertest, which results in the more readable Supertest in Figure 4.4.

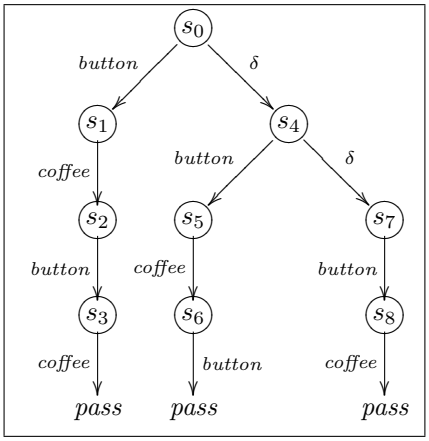


Figure 4.4: Supertest without error transitions

There are two functions called when using *AbsCov*, namely `alpha` and `doCalculate`. The function `alpha` returns the discount value given a state and an action, where `doCalculate` performs the algorithm according to the following steps:
`For Supertest` t `and state` s_0`:`

1. `Get the accumulated error weight for the current state.`
2. `Get all transitions from current state of` t`.`
3. `For each transition add its discount value multiplied by the absolute coverage of the next state.`
4. `If there are no transitions from this state, return.`

4.2.3 Example

To demonstrate how SECO handles the calculation of the absolute coverage, we will use the Supertest from Figure 4.4, the specification from Figure 3.1, the error file from Figure 3.6 and the discount file from Figure 3.4. To execute this, we use the command:

```
java main.coverage.AbsCov test123.aut coffee.aut coffee.err
                          coffee.disc
```

AbsCov will start the calculation with calling the method `doCalculate` for state s_0. It will start examining the transitions from the current state of the Supertest. First we find the transition with action `button`, since this is an input, we will do `alpha(`s_0`, button) * doCalculate(`s_1`)`. So the next step is to get the value for `doCalculate(`s_1`)`. We find the output `coffee`, so we mark that we found an output and do: `alpha(`s_1`, coffee) * doCalculate(`s_2`)`. From s_2 we get the input `button` and we will do `alpha(`s_2`, button) * doCalculate(`s_3`)`. When in s_3 we once again find the output `coffee` we store that we found an output on this level and try to calculate the coverage of the next step. Since this is a pass state, it will return the value zero. Back up one level to see if we find more transitions. There are no more transitions from s_3, but we have marked that we found an output here. Since there was an output, we add the $\overline{r}$ for the state of the specification that belongs to this state of the Supertest, which is 2. Up one level, where we will find no more transitions and no marked output. So up another level, where we also find no more transition, but we do have an output. So once again we add 2 to our result. Another level up and we are back in s_0, here we do find another transition, the output δ. So we mark that we found an output and we will do `alpha(`s_0`, `δ`) * doCalculate(`s_4`)`. From here on we leave the rest of the example to the reader. The result that the module *AbsCov* is approximate 6.407. This is the same value as we got with the algorithm.

4.3 Total Coverage

As mentioned earlier, not all of SECO is programmed in Java. The total coverage part has been programmed in Maple. Since specifications most of the times contain loops the number of tests you can derive are infinite, but we still want to know what the maximal coverage is we can achieve for that specification. To do this we use two way to make the state space finite: *discounting* and *finite depth*.

4.3.1 Maple

Maple is a general purpose commercial mathematics software package. Users can enter mathematics in traditional mathematical notation. Some examples are:

```
3*2^4;
5!;
A:=Matrix([[1, 1, 0], [2, 0, 1], [3, 0, 1]]);
```

These inputs generate the expected outputs: 48, 120 and the three-by-three matrix A, containing the rows (1,1,0), (2,0,1) and (3,0,1). In a same manner vectors can be created too.

Maple is a very big tool and only a little part of it is used for the calculation of total coverage. We use the basics of Maple, extended with the package *linalg*. This package has functionality for using linear algebra. Since we need to do matrix calculations, this package comes in very handy.

4.3.2 Total Coverage in Discounted FA Algorithm

Given $F = \langle A, r\rangle$ a FA with $A = \langle S, s^0, L, T\rangle$, $r : S \times O \to \mathbb{R}^{\geq 0}$, a state $s \in S$ and a discounting function α for F, we desire to calculate:

$$totcov(f_s^\alpha) = \sum_{\sigma \in L^*} f_s^\alpha(\sigma)$$

We assume that from each state in F we can reach at least one error state, then we get:

$$\forall s \in S : \exists s' \in reach_{F|s|} : \overline{r}(s') > 0$$

With some rewriting we can come to a matrix-vector notation which is as follows:

$$tc = \overline{r} + A^\alpha \cdot tc = (I - A^\alpha)^{-1} \cdot \overline{r}$$

When we use the fault automaton from Figure 3.7 and the discount function α from Figure 4.3, the algorithm can be executed.

$$A^\alpha = \begin{pmatrix} 0.499 & 0.499 & 0 \\ 0.6646666666666666 & 0 & 0.3323333333333333 \\ 0.747 & 0 & 0.249 \end{pmatrix}$$

$\overline{r} = [3, 2, 1]$

Results in $tc(s_0) \approx 963.141$

4.3.3 Total Coverage in Finite Depth FA Algorithm

Given $F = \langle A, r\rangle$ a FA with $A = \langle S, s^0, L, T\rangle$, $r : S \times O \to \mathbb{R}^{\geq 0}$, a state $s \in S$ and a depth $k \in \aleph$, we desire to calculate:

$$totcov(f_s^k) = \sum_{\sigma \in L^*} f_s^k(\sigma)$$

We assume that from each state in F we can reach at least one error state in k steps, then we get:

$$\forall s \in S : \exists s' \in reach^k_{F|s|} : \overline{r}(s') > 0$$

With some rewriting we can come to a matrix-vector notation which is as follows:

$$tc_k = \sum_{i=0}^{k-1} A^i \cdot \overline{r}$$

We once again use the FA from Figure 3.7 and we use depth $k = 5$.

$$A = \begin{pmatrix} 1 & 1 & 0 \\ 2 & 0 & 1 \\ 3 & 0 & 1 \end{pmatrix}$$

$\overline{r} = [3, 2, 1]$

Results in $tc_5(s_0) = 133$

4.3.4 Example

Since I used the matrix notation to calculate the total coverage, the algorithm could be used exactly like it was in Maple. Only some intermediate steps are present for the matrix multiplications. Hence we use the example direct in stead of first explaining how we programmed the algorithm in SeCo. To be able to use the matrix calculation, we use the package `linalg`. To do this, the first line of the Maple sheet needs to be:

```
with(linalg):
```

After this, we can specify the matrix A and the vector with total error weights per state ($\overline{r}$). For the example we use the fault automaton from Figure 3.7. In Maple this will be:

```
A:=Matrix([[1, 1, 0], [2, 0, 1], [3, 0, 1]]);
R_:=vector(3,[3,2,1]);
```

When we want to use discounting, we need a discount vector too:

```
discount:=vector(3,[0.499,0.3323333333333333,0.249]);
```

So now we have all the inputs that we need and we can start on the algorithm. Since it is a 3×3 matrix, we need I_3:

```
id3:=Matrix(3, 3, shape=identity);
```

Now lets create A^{α};

```
for i from 1 by 1 to 3 do A:=mulrow(A,i,discount[i]) end do:
```

And finish with the calculation of the total coverage in discounted FA:

```
totcov:=evalf(multiply(matadd(-1*A, id3)\^{}(-1),R\_{}));
```

So now our complete program for calculating total coverage in discounted FA looks like:

```
with(linalg):
A:=Matrix([[1, 1, 0], [2, 0, 1], [3, 0, 1]]);
R_:=vector(3,[3,2,1]);
discount:=vector(3,[0.499,0.3323333333333333,0.249]);
id3:=Matrix(3, 3, shape=identity); for i from 1 by 1 to 3 do
    A:=mulrow(A,i,discount[i]) end do:
totcov:=evalf(multiply(matadd(-1*A, id3)\^{}(-1),R\_{}));
```

The result given by Maple after execution of this program is:

```
totcov := vector([963.1406123, 960.9888714, 959.3422602])
```

We see that this returns 963.141, like the algorithm did. The other two values that we can see are the total coverage values when starting in state s_1 and in state s_2.

The matrix A and the vector $\overline{r}$ are the same for total coverage in finite depth FA. Of course we do not need the discount vector here, so we can start with the algorithm directly and the complete program for total coverage in finite depth FA will be:

```
with(linalg):
A:=Matrix([[1, 1, 0], [2, 0, 1], [3, 0, 1]]);
R_:=vector(3,[3,2,1]);
tc:=k->multiply(sum('A^i', 'i' = 0..(k-1)),R_);
```

When we execute the program for $k = 5$ with the command `tc(5);`, Maple returns: $\begin{pmatrix} 133 \\ 203 \\ 254 \end{pmatrix}$
And again we see the same result as with the algorithm, namely 133. The other two values are the total coverage values when starting in state s_1 and in state s_2, like we saw with the discounted algorithm.

4.4 Relative Coverage

Relative coverage is the coverage of a Supertest with respect to the total coverage. So we need both the absolute coverage (Section 4.2) and the total coverage (Section 4.3) value to calculate this.

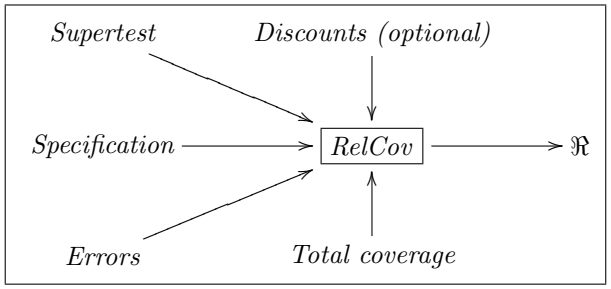


Figure 4.5: Architecture of the RelCov module

4.4.1 Relative Coverage Algorithm

Combining the algorithms for computing total coverage and absolute coverage , it is easy to determine the relative coverage for a test suite T and $f = f_s^k$ or $f = f_s^\alpha$:

$$relcov(T, f) = \frac{abscov(T,f)}{totcov(f)}$$

We might multiply with 100% to get a percentage, but this value is also clear. When we use the algorithm with the values for absolute coverage and total coverage we got in the earlier Sections, it results in:
T is the Supertest from Figure 3.17, the fault automaton f from Figure 3.7 and discount function α (Figure 4.3)
$relcov(T, f) = \frac{6.407}{963.141} \approx 6.652 \cdot 10^{-3}$

4.4.2 Relative Coverage with SeCo

The module that executes the relative coverage algorithm is called *RelCov*. Next to the inputs that *AbsCov* has, it also takes a double value to represent the total coverage as an input. When we execute the *RelCov* module, it creates an instance of the *Abscov* module to calculate the absolute coverage. After the absolute coverage value is obtained, this value is divided by the provided total coverage value.

4.4.3 Example

Total coverage of the coffee specification (Figure 3.1) is approximately 963.141, which we showed in Section 4.3, the absolute coverage of the example used in Section 4.2 is approximately 6.407, so the relative coverage of this example is $\frac{6.407}{963.141} \approx 0.006652$. To get this result, use the command:

```
java main.coverage.RelCov Test123.aut coffee.aut coffee.err
                    coffee.disc 963.141
```

4.4.4 Implementation Issues

The absolute coverage that was calculated by *RelCov* was different from the value that *AbsCov* returned if we ran it separately. The problem was that we supplied all the arguments of *RelCov*, including the total coverage, to *AbsCov*. The extra input caused errors in the *AbsCov* module. When we strip the last argument before we instantiate *AbsCov*, the correct value is returned.

5 Test Selection with SeCo

5.1 Introduction

In the previous chapter we described the algorithms for semantic coverage and the implementation of these algorithms in SeCo. The coverage values per test can help us select the best tests for a specification.

Test selection will be the scope for this chapter. We will present algorithms for test selection and their implementations. The test selection consists of three algorithms:

- Optimal coverage in a test case
- Optimal coverage in n test cases
- Optimal coverage in a test suite (with n test cases)

Optimal coverage in a test case produces a test of given length with highest absolute coverage for a specification. We will be able to use this for both the discounted and the finite depth model by creating a discount function for the finite depth model. This function uses the discount value 1 for each transition.

Optimal coverage in n test cases is actually applying the previous algorithm, repeating the algorithm but excluding the tests we already selected. If we execute the algorithm n times, we will have the best n tests.

Optimal coverage in a test suite (with n test cases) differs from the previous algorithm, since we want to produce a test suite with high coverage here. To get high coverage in a suite, we need to have little (or no) overlap between the test cases we use for the suite.

Organization of the Chapter
In Section 5.2 we start with description of *optimal coverage in a test case*. Next we present the algorithm and after that we show how we implemented

the algorithm. We will make clear how the module *Optimize* works by giving a example. We finish the section with remarks about the implementation.

The next section will give a short description of *optimal coverage in n tests cases*, followed by a short example.

The last section deals with *optimal coverage in a test suite (with n test cases)*. After a short introduction, we present the algorithm. The implementation is then presented with a small example.

5.2 Optimal Coverage in a Test Case

Optimal coverage in a test case can be used to select the best test of length k for a specification or to select the shortest test case with high coverage. The algorithm for the best test of length k takes a FA, a discount function and a natural number as inputs. The result of the algorithm is a single test of the length specified. The algorithm for the shortest test with high coverage uses a rational number in stead of a natural number.

Since we combined these two algorithms in the module *Optimize*, the module also needs to know which of the algorithms is used. This is resolved by using a command line argument to indicate which algorithm is used.

5.2.1 Optimal Coverage in a Test Case Algorithms

The first algorithm is the best test of length k, after that we will describe the shortest test case with high coverage algorithm. Since we want to use the algorithms for both the finite depth and the discounted model, we need to define a discount function for the finite depth model.

Given a specification A with $A = \langle S, s_0, L, T\rangle$, we create the following discount function α for the finite depth model:

$$\alpha(s,l,s') = \begin{cases} 1 & (s,l,s') \in T \\ 0 & \text{otherwise.} \end{cases}$$

Now we can define the algorithms using the above specified discount function when working with finite depth without discounting. The algorithm for the best test of length k can now be specified.
Let $F = \langle A, r\rangle$ be a FA with $A = \langle S, s_0, L, T\rangle$ and $r : S \times O \to \mathbb{R}^{\geq 0}$, α be a discount function and $k \in \mathbb{N}$ be a test length. Then $acopt_k$ satisfies the following equations:

$$\begin{array}{lcl} acopt_0(s) & = & 0 \\ acopt_{k+1}(s) & = & \max\Big(\overline{r}(s) + \sum_{(b!,s')\in T^O(s)} \alpha(s,b!,s') \cdot acopt_k(s'), \\ & & \qquad \max_{(a?,s')\in T^I(s)} \alpha(s,a,s') \cdot acopt_k(q')\Big) \end{array}$$

The algorithm to get the shortest test case with high coverage is actually an LP problem.
Let $F = \langle A, r\rangle$ be a FA with $A = \langle S, s_0, L, T\rangle$ and $r : S \times O \to (R)^{\geq 0}$, and α be a discount function. Then $\mathtt{mw}$ is the optimal solution of the following LP problem:
minimize $\sum_{s\in S} \mathtt{mw}(s)$ subject to

$$\begin{array}{ll} \mathtt{mw}(s) \geq \alpha(s,a?,s') \cdot \mathtt{mw}(s') & (a?,s') \in T^I(s) \\ \mathtt{mw}(s) \geq r(s) + \sum_{(b!,s')\in T^O(s)} \alpha(s,b!,s') \cdot \mathtt{mw}(s') & s \in S \end{array}$$

5.2.2 Optimal Coverage in a Test Case with SeCo

SeCo can return the test case with highest coverage for a specification with the module *Optimize*. The command line arguments needed for this module are: a specification file, an error file, the String "depth", a depth for the test and optionally a discount file. The method `doOptimize` performs the algorithm, but with the help of a couple of other methods.

- `alpha` returns the discount value for a given state and action
- `inputs` returns all transitions from a given state that use input actions
- `outputs` returns all transitions from a given state that use output actions
- `r_` returns the sum of all errors for a given state

Now some more detail on how we programmed the method `doOptimize`. The method has two parameters, the current state and the current depth. Therefore we initially call it for state s_0 and depth 0 and then works according to the following steps for state s and depth d:

1. for each output transition $(s, b!, s')$, multiply $\alpha(s, b!)$ with `doOptimize`$(s', d+1)$, sum the results and add $\overline{r}(s)$
2. for each input transition $(s, a?, s')$, multiply $\alpha(s, a?)$ with `doOptimize`$(s', d+1)$
3. if the requested depth is reached, return 0 for the current state and the action used to get here
4. else return the trace with maximal coverage

5.2.3 Example

As an example we will get the best test case of length 3 for the specification in Figure 3.1 and the error values presented in Figure 3.6.

We execute the module with the command:

```
java main.testselection.Optimize coffee.aut coffee.err depth 3
```

Optimize will now start at state s_0 of the specification in `coffee.aut` on depth 0 and recovers all the transitions from this state. So we get the input `button` and the output `delta`. For the output part, we need to do $\overline{r}(s_0) + \alpha(s_0, \texttt{delta}) \cdot$ `doOptimize`$(s_0, 1)$. Where for the input part, we need to do $\alpha(s_0, \texttt{button}) \cdot$ `doOptimize`$(s_1, 1)$. We keep calling `doOptimize` until we reach depth 3 and then the coverage values and the actions performed will be passed up the tree.

In 5.1 we see the Supertest containing all tests of depth 3 (without the transitions leading to a fail state). Since we use no discounting, all the function `alpha` always returns 1. Next a list of the values that `doOptimize` will return for each state.

- `doOptimize`$(s_2, 2)$ = max(0, 1+0+0+0) = 1, the path leading to the optimal trace from here is the one with the outputs `coffee`, `espresso` and `cappuccino`.

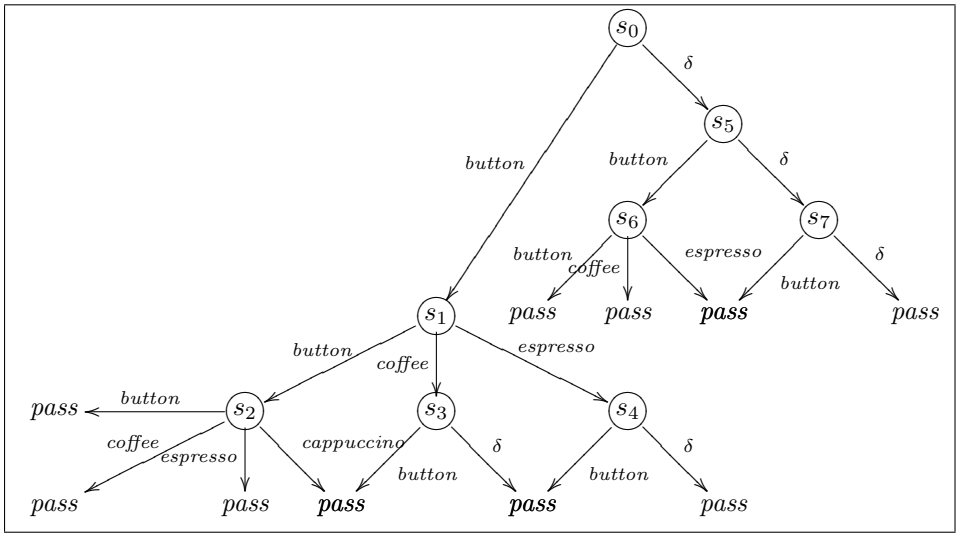


Figure 5.1: Test suite with all tests of length 3 (without error transitions)

- `doOptimize`$(s_3, 2)$ = max(0, 1+0) = 3, the path leading to the optimal trace from here is the one with the output δ.
- `doOptimize`$(s_4, 2)$ = max(0, 1+0) = 3, the path leading to the optimal trace from here is the one with the output δ.
- `doOptimize`$(s_1, 1)$ = max(1, 2+3+3) = 8, the paths leading to the optimal trace from here is `coffee,` δ combined with `espresso,` δ.
- `doOptimize`$(s_6, 2)$ = max(0, 2+0+0) = 2, the path leading to the optimal trace from here is the one with the outputs `coffee` and `espresso`.
- `doOptimize`$(s_7, 2)$ = max(0, 1+0) = 3, the path leading to the optimal trace from here is the one with the output δ.
- `doOptimize`$(s_5, 1)$ = max(2, 3+3) = 6, the path leading to the optimal trace from here is δ, δ.
- `doOptimize`$(s_0, 0)$ = max(8, 3+6) = 9, the path leading to the optimal trace from here is δ, δ, δ.

So the best test of length 3 is the one shown in Figure 5.2 with the absolute coverage value of 9.

5.2.4 Implementation Issues

We now return tests, where we first only returned traces. To achieve this we use the module *Explode* (Section 6.2).

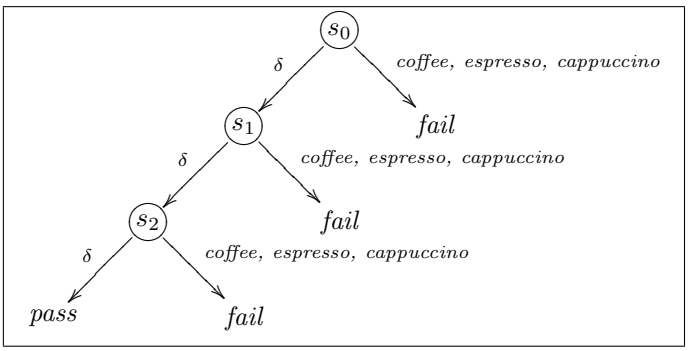


Figure 5.2: Test of length 3 with the highest coverage

5.3 Optimal Coverage in n Test Cases

The algorithm for computing the best test case of length k can be extended to a method for computing the best n test cases with optimal coverage. To pick the second best test case, we apply the same procedure as to get the best test, except that we exclude the first from all possible options. To get the third best test, exclude the first two, and so on. In this way we can get the first n separate tests with the best coverage values.

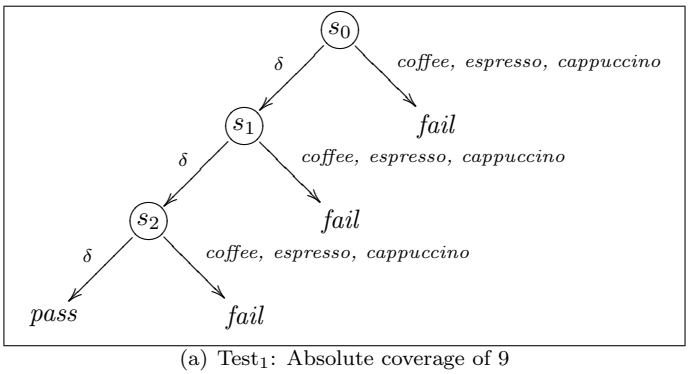
5.3.1 Optimal Coverage in n Test Cases with SeCo

Since this is actually returning the best n traces in stead of only the best trace, we take the $\max_n$ in stead of the max. We then return a set of traces in stead of a trace. If we execute this for the specification in Figure 3.1, the error values presented in Figure 3.6, length 3 and $n = 3$, we get the three test cases in Figure 5.3.

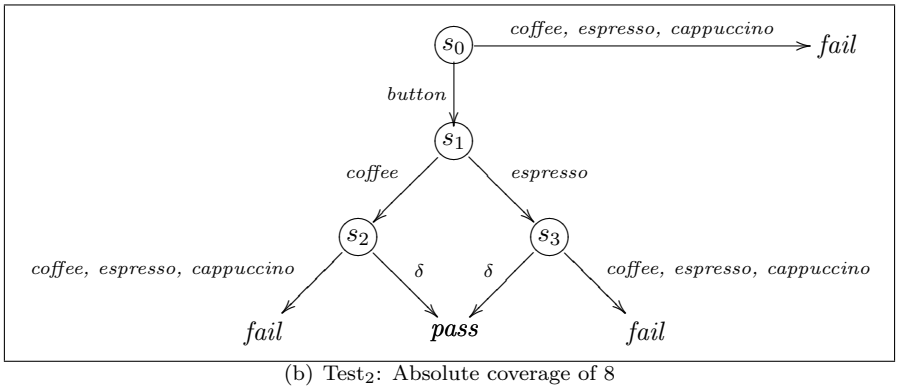
We see here that we have the best test case that we obtained with the example in Section 5.2 along with the next two best tests. To take the best 3 in the same manner as used in Section 5.2.3 is left as an exercise to the reader.

The command to execute the example with SeCo is:

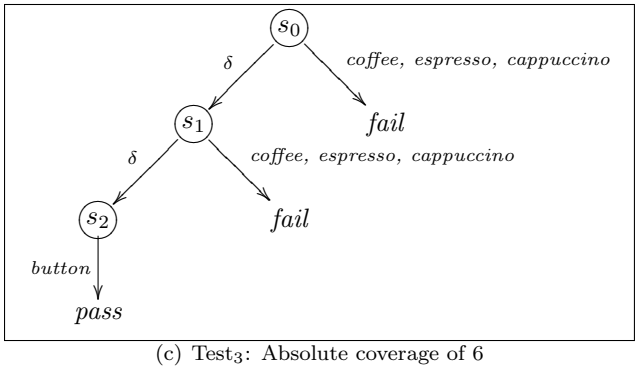
```
java main.testselection.OptNTests coffee.aut coffee.err 3 3
```



(a) Test_1: Absolute coverage of 9



(b) Test_2: Absolute coverage of 8



(c) Test_3: Absolute coverage of 6

Figure 5.3: Best 3 tests of length 3

5.4 Optimal Coverage in a Test Suite

Where we in the previous two sections got single tests from our algorithm, we now aim to get a test suite. The difference with the optimal coverage in n tests is that we want as less overlap as possible in the tests we select. In this way the combined tests make a suite that covers more than the best tests combined.

5.4.1 Optimal Coverage in a Test Suite Algorithm

Let $F = \langle A, r\rangle$ be a FA with $A = \langle S, s_0, L, T\rangle$, $r : S \times O \to \mathbb{R}^{\geq 0}$, α be a discount function for F, $k \in \mathbb{N}$ be a test length and $n \in \mathbb{N}$ be a number. Then $\texttt{tc}_k$ satisfies the following equations

$$
\begin{aligned}
v_0(s) =& [0, 0, .., 0] \\
v_{j+1}(s) =& max_n\{\big[\alpha(s, a, s') \cdot v \mid (a?, s') \in T^I(s), v \leftarrow v_j(s') \wedge 0 \leq j \leq n\big] + + \\
& r(s) \oplus \left[\sum_{(b!,s') \in T^O(s)} \alpha(s, b!, s') \cdot l \mid l \leftarrow v_j(s') \wedge 0 \leq j \leq n\right]\}
\end{aligned}
$$

Here, $x \oplus l$ adds the number $x \in \mathbb{R}^{/geq0}$ to each element of the list l (i.e., $x \oplus [e_1, e_2, \ldots e_n] = [e_1 + x, e_2 + x, \ldots e_n + x]$). And max_n yields the n maximal elements in a list. By keeping the lists sorted (largest element first) we can efficiently implement the algorithm. To do so, it suffices that max_n returns a sorted list.

5.4.2 Optimal Coverage in a Test Suite with SeCo

When given a specification file, an error file, a test length k, a number of tests n and optionally a discount file, SeCo starts with generating al the tests of length k. It stores these test in an ordered set, with the test with highest coverage first.

To get the test suite of n tests with the highest coverage, *Merge* is used. We start with getting the test with highest coverage, then merge it with the next test. Then we calculate the absolute coverage for this suite and store the suite with its coverage value. We now merge our highest test with the next test and calculate the coverage for this suite. If the coverage is higher than the stored value, we update the new highest value and store the current suite. After we combined the best test with all the other tests, we have the test suite of 2 tests with highest coverage. We can repeat this process until we merged n tests where we now see the just created test suite as highest test.

To get the best test suite of 3 tests of length 3, we use the command: The command to execute the example with SeCo is:

```
java main.testselection.OptSuite coffee.aut coffee.err 3 3
```

Within a second, SeCo provided the test shown in Figure 5.4. We can see that the test with highest coverage for depth 3 is present in this suite. Test_3 (Figure 5.3c) is not in this test suite, since it has too much overlap with test_1 (Figure 5.3a). The example shows us that the best 3 test do not necessarily merge into the best test suite of 3 tests.

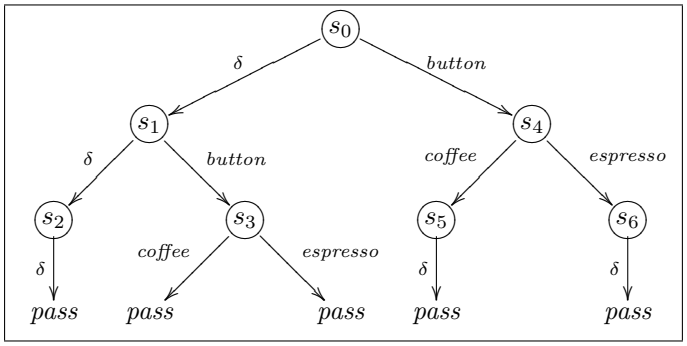


Figure 5.4: Best test suite of 3 tests of length 3 (without error transitions)

6 Extra Features

6.1 Introduction

To keep SECO user friendly, we implemented modules that help us create more readable output or generate input (files). We will present three modules in this chapter:

- Explode
- GenerateDiscounts
- GenerateMatrix

Explode is used to create a test from a trace or a test suite from a suite of traces. In this way we can present the results of algorithms from previous chapters as a test to the user.

GenerateDiscounts is a module that helps the user by creating a discount file (.disc) for a given specification by applying a basic algorithm to calculate discount values.

GenerateMatrix will help us to create parts of the Maple sheet we need to calculate total coverage (Section 4.3). It generates the adjacency matrix for a given fault automaton as well as the vector $\overline{r}$.

Organization of the Chapter
In Section 6.2 we will introduce the module *Explode*. The next section describes the module *GenerateDiscounts*. It starts with a short description, followed by the algorithm and an example. The final section covers an introduction to the module *GenerateMatrix* and an example of how to use the module.

6.2 Explode

The module *Explode* is used to create a test from a trace. Where traces are only paths, tests require to have all outputs possible, whenever an output is possible. This module is executed from within the *Optimize* module to make the best trace that is found into a test. The test in Figure 5.2 was obtained from using *Explode* on the trace in Figure 6.1.

The module can also be executed separately, to do this execute the command:

```
java main.extras.Explode <test|test suite> <specfile>
                         <outputfile>
```

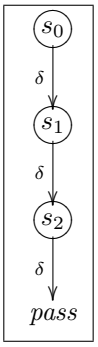


Figure 6.1: Trace that needs to be exploded to a test

6.3 Generate Discount Values

The discount function (alpha) needs to be presented to SeCo in a file. An example of such a file can be found in Appendix A. At first we made these files by hand, but that was very time consuming. Thus we created the module *GenerateDiscounts*. This module takes a specification, an error-file and a value epsilon and then generates a file with discount transitions and gives code we can use in Maple to represent the discount values there.

6.3.1 Discount Value Calculation Algorithm

The discount values for the transitions from state s are calculated by the following algorithm: $\frac{1}{outdegree(s)} - \epsilon$, where $0 < \epsilon < outdegree(s)$.

6.3.2 Example

To obtain the file containing the transitions representing the discount function we see in Figure 4.3, we use the command:

```
java main.extras.GenerateMatrix coffee.aut coffee.err
                 coffee_example.disc
```

SeCo first calculates the outdegree for every state. After this has been done, it calculates the discount value $\frac{1}{outdegree(s)} - 0.001$ for all the states and stores the pairs of state and discount value internally. When SeCo has all the discount values, it adds the appropriate discount value to all the transitions from the specification and adds the delta transitions with discount values too. The result in `coffee_example.disc` looks like:

```
(0, "button", 1, 0.499)
(1, "button", 2, 0.3323333333333333)
(1,"espresso", 0, 0.3323333333333333)
(1, "coffee", 0,0.3323333333333333)
(2, "button", 2, 0.249)
(2, "espresso", 0,0.249)
(2, "coffee", 0, 0.249)
(2, "cappuccino", 0, 0.249)
(0,"delta", 0, 0.499)
```

Apart from the file that is written, we also obtain the output: `discount:=vector(3,[0.499,0.3323333333333333,0.249]);`

6.4 Generate Maple Input

Since the matrix we need to supply to the total coverage algorithm can be very large, we provided a program that can generate the matrix A as Maple input and the vector $\overline{r}$, when given a file containing the specification, a file with error transitions and a target file. The target file will contain the Maple input. This part of SeCo is called *GenerateMatrix*.

6.4.1 Example

The input generation for the specification from Figure 3.1 and the error transitions from Figure 3.6 SeCo will work with the following command:

```
java main.extras.GenerateMatrix coffee.aut coffee.err maple.txt
```

First we calculate the adjacency matrix we need, by counting for every state the transitions to every other state. In our example that will be:

from/to	s_0	s_1	s_2
s_0	1	1	0
s_1	2	0	1
s_2	3	0	1

When this is finished, we will calculate the vector $\overline{r}$ which will be: $\langle 3, 2, 1 \rangle$
Now we format the calculated values into the Maple input we need and store this in the file `maple.txt`. This file will contain the following text:

```
A:=Matrix([
[1, 1, 0],
[2, 0, 1],
[3, 0, 1]
]);

R_:=Matrix([[3.0], [2.0], [1.0]]);
```

7 Semantic Versus Mutant Coverage: a Comparison

7.1 Introduction

To evaluate the usefulness of the algorithms presented in Chapter 4, we compare the algorithms to another form of coverage. This chapter deals with a comparison between absolute (Section 4.2) and mutant coverage. Mutant coverage is chosen based on it being a semantic type of coverage, like we use with SeCo. Using several mutated implementations is already used in practice and has proven itself useful.

Since we used the tool TorX to aid us in trace generation and conformance testing, a description of this tool is presented in this chapter.

We define mutant coverage as the percentage of IOCO incorrect mutants that have been discovered. To actually perform the comparison, we created ten mutants of the coffee specification shown in Figure 7.1. Seven of these mutants were IOCO incorrect. With the help of TorX and SeCo 16 test suites were created. We created four sets of discount values for the specification, so for each test suite we have four absolute and relative coverage values.

None of the test suites has marked a IOCO implementation as incorrect. The results show that when semantic coverage increases within a certain depth, the mutant coverage stays equal or increases too. It also became obvious that some errors cannot be detected with test suites of small depth. Discount values close to the one divided by the outdegree of a state give low coverage values.

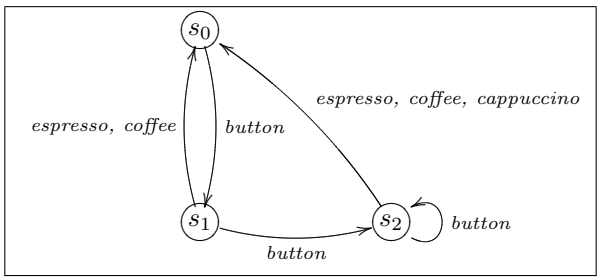


Figure 7.1: Specification of the coffee example

Organization of the Chapter
This chapter will start with a section that describes the tool TorX, which is used for creating traces and for performing the tests that will lead to the mutant coverage value.

Next we explain the background on mutant coverage, where we will provide a formal definition of mutant coverage. Following this, we will describe which mutants we have for the coffee example. All the mutants will be presented, along with where the errors are. After this, we present the test setup for the coffee example together with the results of the tests. We finish this chapter with an analysis of the results we achieved with our tests.

7.2 TorX

TorX is a prototype testing tool for conformance testing of reactive software. It requires an implementation and a (formal) specification of that implementation. The specification describes the system behavior for the implementation. TorX can check if the implementation behaves correctly according to the specification. TorX also allows to generate tests for a given specification.

In the process of testing SECO we used TorX to generate traces. To do this the command line option `-batch` was used. The creation of traces can be done in the following manner:

```
torx --batch --depth 10 coffee.if
```

TorX now creates a trace of depth 10 for the specification described in `coffee.if`. Since we needed a lot of tests, we used this command in a for-loop.

When we want to check if a mutant is IOCO with a test suite we created, we can use the conformance testing part of TorX. We make the test suite act as the specification for the mutant and then let TorX run. When an inconsistency is found, TorX will return where the difference between specification and implementation was.

7.3 Background on Mutant Coverage

Mutant coverage uses faulty variations of the specification that needs to be tested, called mutants. A suite of test cases based on the specification can

be applied to the mutant to see if the test suite can detect that it is a faulty variation. Mutant coverage is defined by the number of IOCO incorrect mutants that are detected by a test suite divided by the total number of IOCO incorrect mutants multiplied by 100%. In this way the mutant coverage will depict the percentage of detected IOCO incorrect mutants. Next is a formal definition for mutant coverage.

For a specification s, a test suite T and a set of mutants M:

$$mutantcov(T, s, M) = \frac{\{m | m \in M \wedge v(m,T)=\textbf{fail}\}}{\#\{\forall m \in M | m \not\mathit{IOCO}\ s\}} * 100\%$$

$$v(m, T) = \begin{cases} \textbf{pass}, & \text{if } \forall \sigma \in \ m : T \stackrel{\sigma}{\Longrightarrow} \textbf{pass}; \\ \textbf{fail}, & \text{otherwise.} \end{cases}$$

7.4 Mutant Coverage for the Coffee Example

The specification of the coffee example has ten mutants. Six of these mutants already existed as examples within the TorX distribution (Belinfante et al., 1999). We added four extra to have more variance in the mutant coverage. Table 7.1 shows for every mutant the number of states and the number of transitions (including delta transitions) and if the mutant is IOCO with the specification. The mutants are shown in Figures 7.2 until 7.11 and their specifications can be found in Appendix C. The mutants `impl1`, `impl2` and `impl4` are IOCO with the specification and should not be marked as erroneous implementations.

Mutant	States	Transitions	IOCO
impl1	3	8	✓
impl2	2	5	✓
impl3	4	12	
impl4	2	4	✓
impl5	4	9	
impl6	4	8	
impl8	9	34	
impl9	9	33	
impl10	17	58	
impl11	15	52	

Table 7.1: Mutant information

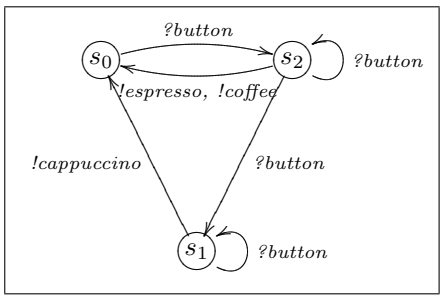


Figure 7.2: Mutant impl1: ioco

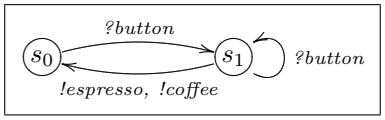


Figure 7.3: Mutant impl2: ioco

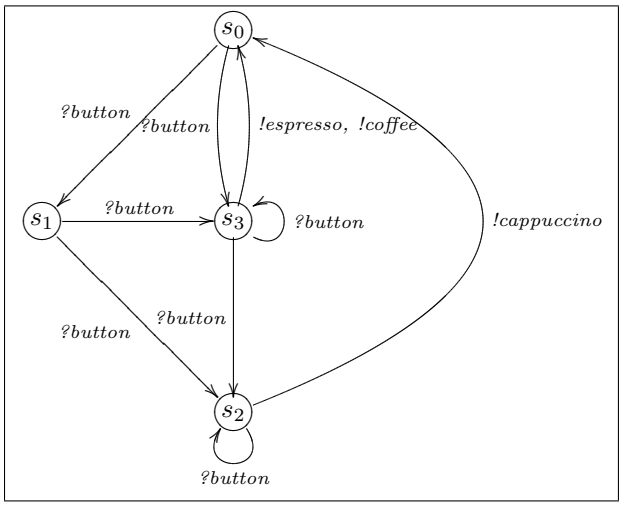


Figure 7.4: Mutant impl3: after ?button is a δ possible

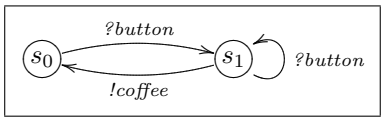


Figure 7.5: Mutant impl4: ioco

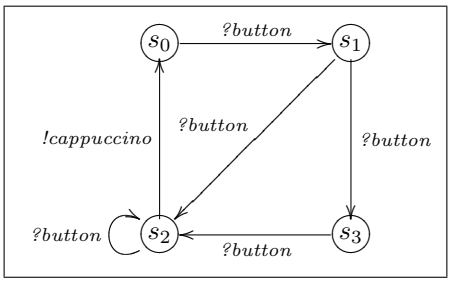


Figure 7.6: Mutant impl5: after ?button is a δ possible

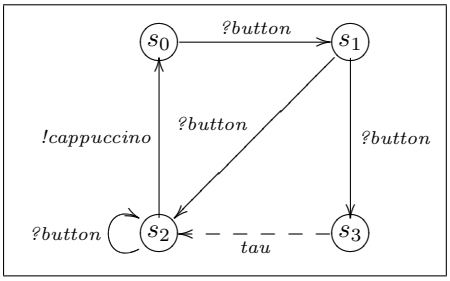


Figure 7.7: Mutant impl6: after ?button is a δ possible

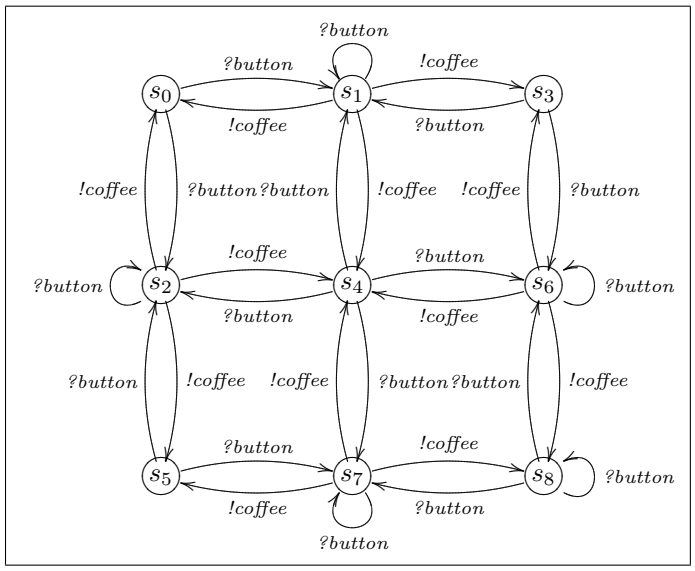


Figure 7.8: Mutant impl8: in s_8 is a self transition with ?button possible

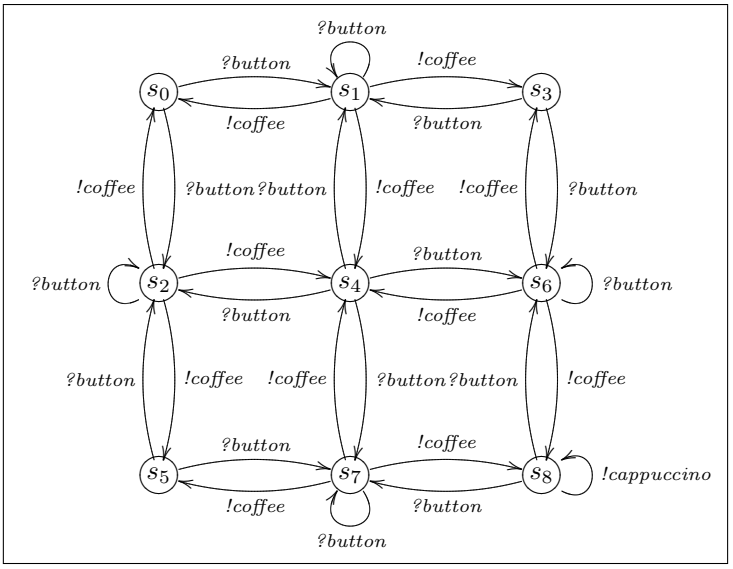


Figure 7.9: Mutant impl9: in state s_8 is !cappuccino possible in stead of δ

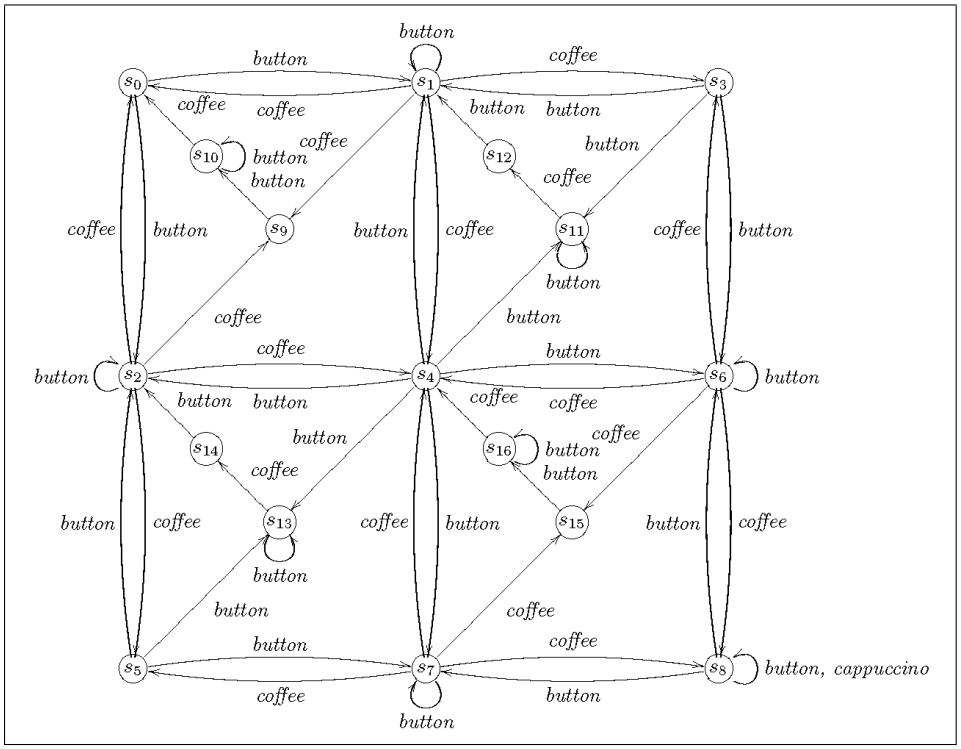


Figure 7.10: Mutant impl10

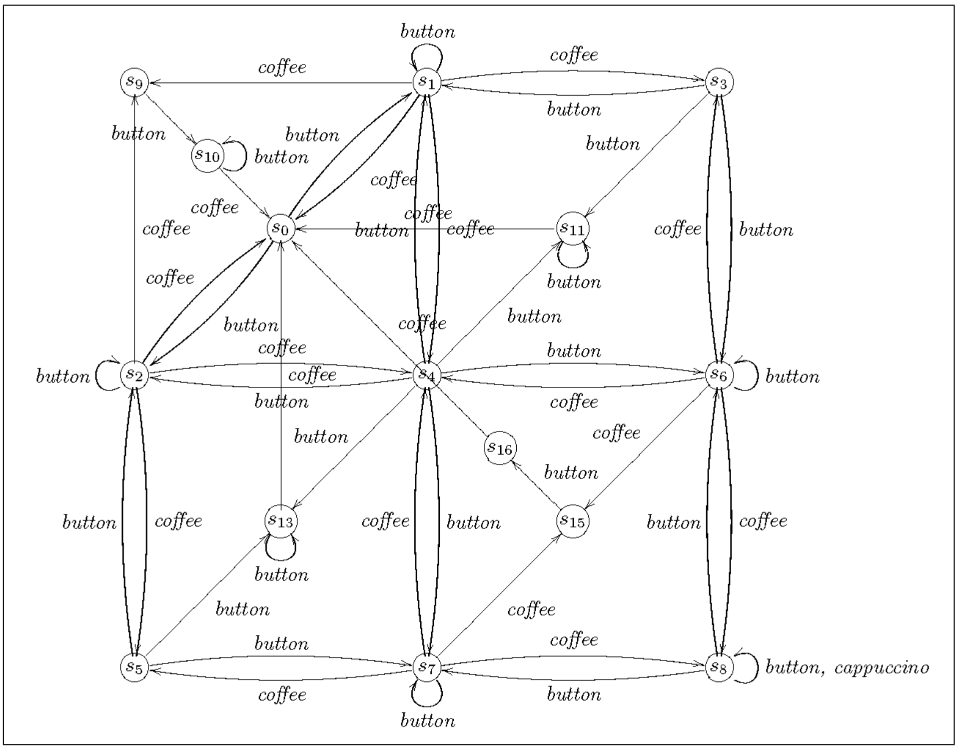


Figure 7.11: Mutant impl11

7.5 Test Setup

We want to compare semantic coverage values with mutant coverage values. To be able to calculate absolute coverage of test suites later on, we first need to calculate the total coverage of the specification in Figure 7.1. All the error values are set to one, and four sets of discount values are used. The discount values are shown in Table 7.2.

With the given specification, the error weights and the discount values, the total coverage can be calculated with Maple as we presented in Section 4.3. The results for the different values of α are shown in Table 7.3.

α	**State 0**	**State 1**	**State 2**
1	0.499	0.332	0.249
2	0.490	0.323	0.240
3	0.400	0.233	0.150
4	0.300	0.133	0.050

Table 7.2: Discount values for the coffee example

With TorX we generated 50 tests of depths 2, 5, 10 and 20. These tests have been merged into test suites with the module *Merge* (Section 3.3). We created 16 test suites, for all depths a suite with 5, 10, 25 and 50 tests. To calculate

Type	Total coverage
Discounting α_1	963.141
Discounting α_2	97.319
Discounting α_3	10.744
Discounting α_4	5.935

Table 7.3: Total coverage values for the coffee example (Figure 3.1)

the absolute and relative coverage, the two corresponding modules of SECO are used. The results for absolute and relative coverage are presented in Table 7.4.

For the calculation of the mutant coverage it is needed to use TorX (Section 7.2). TorX is not able to execute test suites, it only checks one trace in the test suite. To fix this, we checked each test suite against all the mutants with 50 different seeds. This was realized by executing a script for each test suite. The script that was used for test suite 6 can be found in Appendix D.

The resulting log-files show which mutants are detected. None of the test suites has marked a IOCO mutant to be incorrect with respect to the specification. So the number of detected mutants can be divided by 7 and multiplied by 100%. The values for mutant coverage are also incorporated in Table 7.4. Which specific mutants are discovered by the test suites can be seen in Table 7.5.

testsuite	depth	#tests	type	α_1 (tc: 963.141)	α_2 (tc: 97.319)	α_3 (tc: 10.744)	α_4 (tc: 5.935)	mutantcov
1	2	5	absolute	3.998	3.980	3.800	3.600	
			relative	0.4%	4.1%	35.4%	60.7%	42.9%
2	2	10	absolute	3.998	3.980	3.800	3.600	
			relative	0.4%	4.1%	35.4%	60.7%	42.9%
3	2	25	absolute	3.998	3.980	3.800	3.600	
			relative	0.4%	4.1%	35.4%	60.7%	42.9%
4	2	50	absolute	3.998	3.980	3.800	3.600	
			relative	0.4%	4.1%	35.4%	60.7%	42.9%
5	5	5	absolute	5.321	5.208	4.325	3.739	
			relative	0.6%	5.4%	40.3%	63.0%	71.4%
6	5	10	absolute	6.573	6.393	4.963	3.995	
			relative	0.7%	6.6%	46.2%	67.3%	71.4%
7	5	25	absolute	7.764	7.519	5.575	4.262	
			relative	0.8%	7.7%	51.9%	71.8%	71.4%
8	5	50	absolute	7.949	7.687	5.625	4.267	
			relative	0.8%	7.9%	52.4%	71.9%	71.4%
9	10	5	absolute	6.289	6.136	4.940	4.087	
			relative	0.7%	6.3%	46.0%	68.9%	85.7%
10	10	10	absolute	7.380	7.134	5.324	4.187	
			relative	0.8%	7.3%	49.6%	70.5%	100.0%
11	10	25	absolute	8.426	8.070	5.594	4.225	
			relative	0.9%	8.3%	52.1%	71.2%	100.0%
12	10	50	absolute	9.690	9.205	5.954	4.300	
			relative	1.0%	9.5%	55.4%	72.5%	100.0%
13	20	5	absolute	5.631	5.505	4.546	3.878	
			relative	0.6%	5.7%	42.3%	65.3%	100.0%
14	20	10	absolute	7.020	6.780	5.046	4.005	
			relative	0.7%	7.0%	47.0%	67.5%	100.0%
15	20	25	absolute	9.205	8.760	5.804	4.274	
			relative	1.0%	9.0%	54.0%	72.0%	100.0%
16	20	50	absolute	10.052	9.508	5.997	4.301	
			relative	1.0%	9.8%	55.8%	72.5%	100.0%

Table 7.4: Test results for the coffee example

testsuite	impl1	impl2	impl3	impl4	impl5	impl6	impl8	impl9	impl10	impl11	Mutant coverage
1	-	-	✓	-	✓	✓	-	-	-	-	42.9%
2	-	-	✓	-	✓	✓	-	-	-	-	42.9%
3	-	-	✓	-	✓	✓	-	-	-	-	42.9%
4	-	-	✓	-	✓	✓	-	-	-	-	42.9%
5	-	-	✓	-	✓	✓	-	✓	-	✓	71.4%
6	-	-	✓	-	✓	✓	-	✓	-	✓	71.4%
7	-	-	✓	-	✓	✓	-	✓	-	✓	71.4%
8	-	-	✓	-	✓	✓	-	✓	-	✓	71.4%
9	-	-	✓	-	✓	✓	-	✓	✓	✓	85.7%
10	-	-	✓	-	✓	✓	✓	✓	✓	✓	100.0%
11	-	-	✓	-	✓	✓	✓	✓	✓	✓	100.0%
12	-	-	✓	-	✓	✓	✓	✓	✓	✓	100.0%
13	-	-	✓	-	✓	✓	✓	✓	✓	✓	100.0%
14	-	-	✓	-	✓	✓	✓	✓	✓	✓	100.0%
15	-	-	✓	-	✓	✓	✓	✓	✓	✓	100.0%
16	-	-	✓	-	✓	✓	✓	✓	✓	✓	100.0%

Table 7.5: Detected mutants per testsuite

7.6 Analysis

In Table 7.5 we can see that the IOCO implementations were never marked as erroneous implementations of the specification.

The mutants 3, 5 and 6 were always detected, even with tests of depth 2. These three mutants are the smallest implementations, existing of only four states each and all three have an error after one step. The set of outputs after one step in the specification is {*!coffee, !espresso*}, where these three mutants all have δ in the output set. So the error in these specifications can be detected within two steps.

Another easy observation is that the mutant coverage in Table 7.4 is almost equal per depth of the test suite. The mutant coverage value therefore gives little information on how good a test suite is.

The results also show that if the semantic coverage value increases, the mutant coverage stays equal or increases too. Indicating that adding tests or using better tests, might cover more of the specification, but does not always find more errors. It also becomes clear that the coverage value is extremely low when we use discount values close to $\frac{1}{\text{outdeg}(s)}$. α_3 was $\frac{1}{\text{outdeg}(s)} - 0.1$, the coverage values we got with this α seem the most accurate, giving about 40 to 50% coverage for the test suites of depth 5. Compared to state and transition coverage this seems low, since we can get maximal state and transition coverage with this depth. Only then we completely disregard the loops that are present and assume that after one visit of a state or transition, it is working correct. So including the fact that we want to know if errors can occur later on, the 50% seems nice. Every other loop through the complete system would ensure us more of the correctness of the system, or discover an error.

The first two discount functions resulted in very low relative coverage. This seems odd for the higher depths, since we visit every state a couple of times then. The function α_4 shows, in my opinion, too high coverage for the test suites with depth 2.

I'd like to compare the results of the test suites to the coverage of the test suites containing all the tests of the lengths used (Table 7.6).

Depth	Number of tests	Absolute coverage with			
		α_1	α_2	α_3	α_4
2	5	7.901	7.760	6.453	5.230
5	84	15.191	14.439	8.969	5.852
10	9003	27.209	24.475	10.336	5.932
20	263247781	50.788	41.048	10.722	5.935

Table 7.6: Possible different tests per depth and absolute coverage values

Since there are only five tests of length two, I presumed that the first four test suites I created would have same coverage as the suite containing the five tests that are possible. This proved to be wrong. Since the tests I combined in the test suites were generated randomly, there was no guarantee that all five tests would be there. The test with the trace $\delta\delta$ is not in the first four suites. For the mutant coverage this proved to be no problem, since all errors that could be found within this depth were discovered.

The larger test suites could never be complete, since the test suites containing all tests of depth five and higher contain more than 50 tests. When we compare

the best coverage results with α_3 to the sets of all tests, we can see that the test suites score above 55% of the total possible for each depth. Where for depth 5 we have about 60% of al tests possible, but for higher depths a much smaller percentage of all possible tests. So we see that smaller test suites still have a high coverage relative all the test of their depth.

8 Conclusion and Recommendations

The goals for this project were to build a tool to calculate semantic coverage and to perform test selection. These goals were met by creating SeCo.

During the project, we thought it useful to compare the coverage measures used by the tool to another form of coverage. The coverage measure we chose was mutant coverage, since this is also a form of semantic coverage. Moreover, six mutants for the specification we used already existed.

The comparison shows us that if the semantic coverage value increases, it never occurs that the mutant coverage decreases. We also see that the mutant coverage in our case was highly dependant on the depth of the test suite. The relative coverage was highly dependant on the discount function.

The test results indicate that even with smaller test suites, a lot of semantic coverage can be reached. Making the test selection an important part of the tool. The main advantage we get, is that we can use smaller sets of tests in a test suite and still keep good coverage values, making testing a lot less time consuming.

To get higher coverage values, we should try to avoid that multiple delta transitions still can have high coverage. We achieve this by using per transition discounting in stead of the per state discounting we are using now. Then we can give a discount value closer to zero to the transitions with the action delta.

Further Research for the Theory

When doing further research into semantic coverage, I advise to look into the discounting more. Perhaps it is a good idea to give the transitions that cause a loop a very low discount value (indicating that all below this transition counts much less for the coverage). This should make it easier to get nice coverage values for low depths and in this way testing and test selection will be easier.

To have a smaller amount of possible tests, I advise to not allow multiple delta transitions in a row. It is not relevant to have several delta transitions performed in a row, since the delta transition stands for a time-out. Therefore we should change the way that we add delta transitions to the specification.

Further Research for SeCo

At the moment SeCo is only usable for systems that are defined in the Aldebaran format. To be able to use the tool for more types of specifications, we need to alter the read method of the modules and then try to parse the new format into the data structures that we use.

We still need to test SeCo for larger examples. To make SeCo less time consuming for larger systems, it might be useful to see if other data structures will perform better. Think of strings in stead of string arrays.

The test selection still needs to be tested extensively, since I performed only a couple of minor tests to see if it worked.

A graphical user interface might prove to be useful. At the moment the tool works from command line, which is not preferred by a lot of users.

References

Belinfante, A., Feenstra, J., Vries, R. de, Tretmans, J., Goga, N., Feijs, L., et al. (1999). Formal test automation: A simple experiment. In *Proceedings of the ifip tc6 12th international workshop on testing communicating systems* (pp. 179–196). Deventer, The Netherlands, The Netherlands: Kluwer, B.V.

Brandán Briones, L. (2007). *Theories for model-based testing: Real-time and coverage.* PhD thesis, University of Twente, The Netherlands.

Brandán Briones, L., Brinksma, E., & Stoelinga, M. (2006). A semantic framework for test coverage. In *Proceedings of the fourth international symposium on automated technology for verification and analysis (atva'06)* (Vol. 4218, p. 399-414). Springer-Verlag.

Campbell, C., Grieskamp, W., Nachmanson, L., Schulte, W., Tillmann, N., & Veanes, M. (2005, May). *Model-Based Testing of Object-Oriented Reactive Systems with Spec Explorer* (Tech. Rep.). Microsoft Research, Redmond.

Chang, J., & Richardson, D. (1998). Adlscope: An automated specification-based unit testing tool. In *Ase '98: Proceedings of the 13th ieee international conference on automated software engineering* (p. 289). Washington, DC, USA: IEEE Computer Society.

Dawson, E. (2008). *Atlassian clover - code coverage analysis.* `http://www.atlassian.com/software/clover/default.jsp`.

Garavel, H., Jorgensen, M., Mateescu, R., Pecheur, C., Sighireanu, M., Vivien, B., et al. (1997). Cadp97status, applications, and perspectives. In *in i. lovrek (ed.), proceedings of the 2nd cost 247 international workshop on applied formal methods in system design.*

Grabowski, J., Hogrefe, D., Réthy, G., Schieferdecker, I., Wiles, A., & Willcock, C. (2003). An introduction to the testing and test control notation (ttcn-3). *Comput. Netw.*, *42*(3), 375–403.

Lee, D., & Yannakakis, M. (1996). Principles and methods of testing finite state machines - a survey. In *Proceedings of the ieee* (pp. 1090–1123).

Nachmanson, L., Veanes, M., Schulte, W., Tillmann, N., & Grieskamp, W. (2004). Optimal strategies for testing nondeterministic systems. In *Issta '04: Proceedings of the 2004 acm sigsoft international symposium on software testing and analysis* (pp. 55–64). New York, NY, USA: ACM.

Roubtsov, V. (2006). *Emma: a free java code coverage tool.* `http://emma.sourceforge.net/index.html`.

Sun Microsystems, . (2008). *Man page tcov.1.* `http://docs.sun.com/source/820-4180/man1/tcov.1.html`.

Timmer, M. (2008). *Evaluating and predicting actual test coverage.* MSc thesis, University of Twente, The Netherlands.

Tretmans, J. (1996). Test generation with inputs, outputs, and quiescence. In *Tacas* (p. 127-146).

Ural, H. (1992). Formal methods for test sequence generation. *Comput. Commun.*, *15*(5), 311–325.

Williams, T., & Sunter, S. (2000). How should fault coverage be defined? In *Proceedings of the 18th ieee vlsi test symposium (vts'00)* (p. 325).

Appendices

A Example Files

This appendix shows all the input files that we created for the running example in Chapters 3 to 6.

A.1 Testfiles

Test1.aut	**Test2.aut**	**Test3.aut**
des (0, 4, 4)	des (0, 4, 4)	des (0, 4, 4)
(0, "button", 1)	(0, "delta", 1)	(0, "delta", 1)
(1, "coffee", 2)	(1, "button", 2)	(1, "delta", 2)
(2, "button", 3)	(2, "coffee", 3)	(2, "button", 3)
(3, "coffee", 4)	(3, "button", 4)	(3, "coffee", 4)

A.2 Specification

coffee.aut
des (0, 8, 3)
(0, "button", 1)
(1, "button", 2)
(1, "espresso", 0)
(1, "coffee", 0)
(2, "button", 2)
(2, "espresso", 0)
(2, "coffee", 0)
(2, "cappuccino", 0)

A.3 Errorfile

coffee.err
["delta", "espresso", "coffee", "cappuccino"]
(*, "delta", 1)
(*, "espresso", 1)
(*, "coffee", 1)
(*, "cappuccino", 1)

A.4 Discountfile

coffee.disc
(0, "button", 1, 0.499)
(1, "button", 2, 0.3323333333333333)
(1, "espresso", 0, 0.3323333333333333)
(1, "coffee", 0, 0.3323333333333333)
(2, "button", 2, 0.249)
(2, "espresso", 0, 0.249)
(2, "coffee", 0, 0.249)
(2, "cappuccino", 0, 0.249)
(0, "delta", 0, 0.499)

B
Generated Files

B.1 Supertest

This is the combination of the files in Appendix A.1 generated by the tool. The differences with figure 3.17 are: the state numbering, the pass states are not denoted as pass and the fail states are not present.

Test123.aut
des (0, 11, 12)
(0, "button", 1)
(1, "coffee", 2)
(2, "button", 3)
(3, "coffee", 4)
(0, "delta", 5)
(5, "button", 6)
(6, "coffee", 7)
(7, "button", 8)
(5, "delta", 9)
(9, "button", 10)
(10, "coffee", 11)

C

Files for the Comparison

This Appendix shows the files that we used in addition to the files in Appendix A to execute the comparison between semantic and mutant coverage (Chapter 7).

C.1 Discount files

coffee.disc2
(0, "button", 1, 0.49)
(1, "button", 2, 0.3233333333333333)
(1, "espresso", 0, 0.3233333333333333)
(1, "coffee", 0, 0.3233333333333333)
(2, "button", 2, 0.24)
(2, "espresso", 0, 0.24)
(2, "coffee", 0, 0.24)
(2, "cappuccino", 0, 0.24)
(0, "delta", 0, 0.49)

coffee.disc3
(0, "button", 1, 0.4)
(1, "button", 2, 0.2333333333333333)
(1, "espresso", 0, 0.2333333333333333)
(1, "coffee", 0, 0.2333333333333333)
(2, "button", 2, 0.15)
(2, "espresso", 0, 0.15)
(2, "coffee", 0, 0.15)
(2, "cappuccino", 0, 0.15)
(0, "delta", 0, 0.4)

coffee.disc4
(0, "button", 1, 0.3)
(1, "button", 2, 0.1333333333333333)
(1, "espresso", 0, 0.1333333333333333)
(1, "coffee", 0, 0.1333333333333333)
(2, "button", 2, 0.04999999999999999)
(2, "espresso", 0, 0.04999999999999999)
(2, "coffee", 0, 0.04999999999999999)
(2, "cappuccino", 0, 0.04999999999999999)
(0, "delta", 0, 0.3)

C.2 Mutants

impl1.aut	**impl2.aut**	**impl3.aut**	**impl4.aut**
des (0, 7, 3) (2, button, 1) (2, button, 2) (2, espresso, 0) (2, coffee, 0) (1, cappuccino, 0) (1, button, 1) (0, button, 2)	des (0, 4, 2) (0, button, 1) (1, button, 1) (1, espresso, 0) (1, coffee, 0)	des (0, 10, 4) (0, button, 1) (0, button, 3) (1, button, 3) (2, cappuccino, 0) (2, button, 2) (3, button, 2) (3, coffee, 0) (3, espresso, 0) (3, button, 3) (1, button, 2)	des (0, 3, 2) (0, button, 1) (1, coffee, 0) (1, button, 1)

impl5.aut	**impl6.aut**	**impl8.aut**	**impl9.aut**
des (0, 6, 4)	des (0, 6, 4)	des (0, 29, 9)	des (0, 29, 9)
(0, button, 1)	(0, button, 1)	(0, button, 1)	(0, button, 1)
(1, button, 3)	(1, button, 3)	(0, button, 2)	(0, button, 2)
(1, button, 2)	(1, button, 2)	(1, button, 1)	(1, button, 1)
(2, cappuccino, 0)	(2, cappuccino, 0)	(1, coffee, 0)	(1, coffee, 0)
(3, button, 2)	(3, tau, 2)	(1, coffee, 3)	(1, coffee, 3)
(2, button, 2)	(2, button, 2)	(1, coffee, 4)	(1, coffee, 4)
		(2, button, 2)	(2, button, 2)
		(2, coffee, 0)	(2, coffee, 0)
		(2, coffee, 4)	(2, coffee, 4)
		(2, coffee, 5)	(2, coffee, 5)
		(3, button, 1)	(3, button, 1)
		(3, button, 6)	(3, button, 6)
		(4, button, 1)	(4, button, 1)
		(4, button, 2)	(4, button, 2)
		(4, button, 6)	(4, button, 6)
		(4, button, 7)	(4, button, 7)
		(5, button, 2)	(5, button, 2)
		(5, button, 7)	(5, button, 7)
		(6, coffee, 3)	(6, coffee, 3)
		(6, coffee, 4)	(6, coffee, 4)
		(6, button, 6)	(6, button, 6)
		(6, coffee, 8)	(6, coffee, 8)
		(7, coffee, 4)	(7, coffee, 4)
		(7, coffee, 5)	(7, coffee, 5)
		(7, button, 7)	(7, button, 7)
		(7, coffee, 8)	(7, coffee, 8)
		(8, button, 6)	(8, button, 6)
		(8, button, 7)	(8, button, 7)
		(8, button, 8)	(8, cappuccino, 8)

impl10.aut	**impl11.aut**
des (0, 50, 17)	des (0, 47, 17)
(0, button, 1)	(0, button, 1)
(1, coffee, 3)	(0, button, 2)
(1, coffee, 4)	(1, button, 1)
(2, button, 2)	(1, coffee, 0)
(2, coffee, 0)	(1, coffee, 3)
(2, coffee, 4)	(1, coffee, 4)
(2, coffee, 5)	(2, button, 2)
(3, button, 1)	(2, coffee, 0)
(3, button, 6)	(2, coffee, 4)
(4, button, 1)	(2, coffee, 5)
(4, button, 2)	(3, button, 1)
(4, button, 6)	(3, button, 6)
(4, button, 7)	(4, button, 1)
(5, button, 2)	(4, button, 2)
(5, button, 7)	(4, button, 6)
(6, coffee, 3)	(4, button, 7)
(6, coffee, 4)	(5, button, 2)
(6, button, 6)	(5, button, 7)
(6, coffee, 8)	(6, coffee, 3)
(7, coffee, 4)	(6, coffee, 4)
(7, coffee, 5)	(6, button, 6)
(7, button, 7)	(6, coffee, 8)
(7, coffee, 8)	(7, coffee, 4)
(8, button, 6)	(7, coffee, 5)
(8, button, 7)	(7, button, 7)
(8, button, 8)	(7, coffee, 8)
(1, coffee, 9)	(8, button, 6)
(2, coffee, 9)	(8, button, 7)
(9, button, 10)	(8, button, 8)
(10, coffee, 0)	(1, coffee, 9)
(10, button, 10)	(2, coffee, 9)
(3, button, 11)	(9, button, 10)
(4, button, 11)	(10, coffee, 0)
(11, button, 11)	(10, button, 10)
(11, coffee, 12)	(3, button, 11)
(12, button, 1)	(4, button, 11)
(4, button, 13)	(11, button, 11)
(5, button, 13)	(11, coffee, 0)
(13, button, 13)	(4, button, 13)
(13, coffee, 14)	(5, button, 13)
(14, button, 2)	(13, button, 13)
(6, coffee, 15)	(13, coffee, 0)
(7, coffee, 15)	(6, coffee, 15)
(15, button, 16)	(7, coffee, 15)
(16, coffee, 4)	(15, button, 16)
(16, button, 16)	(16, coffee, 0)
(8, cappuccino, 8)	(8, cappuccino, 8)
(0, button, 2)	
(1, button, 1)	
(1, coffee, 0)	

D
Scripts

The script used to check mutants against a test suite:

```
#!/bin/sh first=1 beyond=50 depth=5 mutants="impl1 impl2 impl3
impl4 impl5 impl6 impl8 impl9 impl10 impl11"
MUTANTDIR=/torx-examples-3.9.1/coffee/SPEC/AUT export MUTANTDIR
i=$first while test $i -lt $beyond do
    for m in $mutants do
        MUTANT=$m export
        MUTANT torx --depth $depth --seed $i
            --log /testloop.$i.$m.log coffee5_10.if >
            /testloop.$i.$m.out 2>&1
        sleep 5
    done
    i=‘expr $i + 1‘
done
```