Master's Thesis

Model-based Mutation Testing with SMT Solvers

Matthias Sebastian Kegele[1]

Institute for Software Technology (IST)
Graz University of Technology
A-8010 Graz, Austria

Advisor: Ao.Univ.-Prof. DI Dr.techn. Bernhard Aichernig
Co-Advisor: DI Dr.techn. Elisabeth Jöbstl

Graz, December 9, 2014

[1] E-mail: matthias.kegele@student.tugraz.at

Masterarbeit

Model-based Mutation Testing with SMT-Solvers

Matthias Sebastian Kegele[1]

Institut für Softwaretechnologie (IST)
Technische Universität Graz
A-8010 Graz

Gutachter: Ao.Univ.-Prof. DI Dr.techn. Bernhard Aichernig
Mitbetreuer: DI Dr.techn. Elisabeth Jöbstl

Graz, 9. Dezember 2014

Diese Arbeit ist in englischer Sprache verfasst.

[1] E-Mail: matthias.kegele@student.tugraz.at

Abstract

Showing correctness of a piece of a program is cumbersome and very hard to achieve in a real world program. Tests can assure the correct working of a functionality in a certain context but not show the absence of errors. Test authoring may be as expensive in respect to effort, time and money as the development of the program or even more.

The present thesis applies a novel technique called model-based mutation to automatically generate test cases. An abstraction of a program - a model - is notated in a modeling language (Action Systems) which allows non-determinism. By applying mutation operators, the model is altered to generate mutated versions.

In model-based mutation testing, the original model and a mutated model are checked for conformance. Refinement has been selected as conformance relation as it has been defined for Action Systems. For being able to generate a distinguishing test case, a so-called unsafe state has to be identified. From this state, a transition can be enabled which is valid for the mutated model but not in the original. In fact, this represents the check for non-refinement. For Action Systems, unsafe states have also to fulfill the property of reachability which has to be checked separately.

In the course of this thesis, a prototype has been implemented called *as2smt*. The Action System (in a Prolog notation) models are translated to SMT-LIB, an input language for SMT solvers. An SMT solver serves as a computational back-end of the tool for computing refinement. Due to the standardised input language, different solvers can be configured. Furthermore, the application allows the selection of different algorithms for performing the reachability analysis. Finally, the tool is evaluated on different use cases.

Keywords: model-based mutation testing, automatic test generation, Action Systems, Satisfiability Modulo Theories (SMT) solver, SMT-LIB, refinement.

Kurzfassung

Die Korrektheit eines Teilbereichs eines existierendes Programms zu beweisen ist mühsam und schwierig. Tests können die Korrektheit einzelner Funktionalitäten in einem gewissen Zusammenhang zeigen, aber nicht die generelle Freiheit von Fehlern. Die Erstellung von Software Tests kann genauso aufwendig sein, wie die der Software selbst oder sogar noch aufwändiger.

Diese Diplomarbeit wendet eine neuartige Testfallgenerierungsmethode namens Modellbasiertes Mutationstesten an. Eine Abstraktion eines Programms - ein Modell - wird in einer Modellierungssprache (Action Systems), welche auch Nicht-Determinismus unterstützt, notiert. Durch sogenannte Mutationsoperatoren wird das bestehende Modell verändert. Daraus enstehen *Mutanten* des Modells.

Durch Modellbasiertes Mutationstesten werden das orginale und das mutierte Modell auf Konformität überprüft. Die Verfeinerung wurde als Konformitätsrelation gewählt, da diese bereits für Action Systems definiert wurde. Um Testfälle zu genierieren, die das ursprüngliche Modell von seiner Mutation unterscheiden, muss ein sogenannter *unsicherer Zustand* identifiziert werden. Von diesem Zustand aus kann ein Zustandsübergang gewählt werden, der valide für den Mutanten aber nicht für das Original ist. Diese Überprüfung stellt die Nicht-Verfeinerung dar. Für Action Systems müssen unsichere Zustände zusätzlich die Eigenschaft der Erreichbarkeit erfüllen, die separat überprüft wird.

Im Zuge dieser Diplomarbeit wurde ein Prototyp namens *as2smt* entwickelt. Die Action System-Modelle (in Prolog-Notation) werden übersetzt in SMT-LIB, einer Eingabesprache für SMT solver. Ein solcher SMT solver stellt das Back-end des implementierten Programms dar. Es dient zur Berechnung der Verfeinerung. Durch die standisierte Eingabesprache können verschiedene SMT solver als Back-end konfiguriert werden. Weiters erlaubt die Implementierung die Auswahl von verschiedenen Algorithmen zur Erreichbarkeitsanalyse. Zum Abschluss wird die Implementierung am Hand von Anwendungsfällen evaluiert.

Schlagworte: Modell-basiertes Mutationstesten, automatische Testfallgenerierung, Action Systems, Satisfiability Modulo Theories (SMT), SMT-LIB, Scala, kombinatorisches Parsen.

Statutory Declaration

I declare that I have authored this thesis independently that I have not used other than the declared sources/resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

. . . .

Graz, December 9, 2014 (signature)

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

. . . .

Graz, 9. Dezember 2014 (Unterschrift)

To Ludi

Acknowledgements

I would like to thank everybody who supported throughout my study and the writing of this thesis. I want to express my exceptional gratitude to my advisor Bernhard Aichernig. His actions have sparked my interest in the field of software testing. His open-mindedness and his ability to explain complex issues in an understandable way make him an excellent teacher.

Furthermore, my special thanks goes to my co-advisor Elisabeth Jöbstl who showed patience and understanding. Her advice, objections and remarks contributed to this thesis significantly.

I want to express my appreciation to my friends and fellow students. Without their support, the road through my study und to this thesis would have been much bumpier. I owe my deepest gratitude to my mother Christine, who supported my throughout my whole life.

I owe sincere and earnest thankfulness to my partner Lisa. She has given me the understanding and support that was necessary to come to this point.

Matthias Kegele
Graz, Austria, December 9, 2014

Danksagung

Ich möchte mich herzlich bei allen bedanken, die mich bei durch mein Studium und bei der Erstellung dieser Diplomarbeit unterstützt haben. Mein außerordentlicher Dank gilt meinem Betreuer Bernhard Aichernig. Sein Wirken hat mein Interesse am Gebiet des Testens von Software geweckt. Seine offene Art und seine Fähigkeit komplexe Zusammenhänge einfach zu erklären machen ihn zu einem exzellenten Lehrer.

Ebenso gilt meine Wertschätzung meiner Zweitbetreuerin Elisabeth Jöbstl, die viel Geduld und Verständnis gezeigt hat. Durch ihre Ratschläge, Einwände und Anmerkungen hat sie maßgeblich zur Qualität dieser Arbeit beigetragen. Auch Benedikt Maderbacher gilt mein Dank für die Analyse und Lösung eines Implementierungsproblems des im Zuge der Diplomarbeit entwickelten Programms.

Meine Anerkennung möchte ich hiermit auch der Unterstützung durch meiner Freunde und Studienkollegen ausdrücken, ohne die der Weg durch das Studium bis hin zu dieser Arbeit deutlich steiniger gewesen wäre. Außerordentlichen Dank gilt meiner Mutter Christine, dir mich immer in meinen Bestrebungen unterstützt hat.

Meinen besonderen Dank gilt meiner Partnerin Lisa. Sie hat mir den nötigen Ansporn und die Unterstützung gegeben, um diese Arbeit zu erstellen.

Matthias Kegele
Graz, Österreich, 9. Dezember 2014

Contents

List of Figures

List of Tables

Listings

List of Algorithms

1 Introduction

In 1965, the following statement from Gordon Moore has become Moore's law [65]: *the number of transistors on integrated circuits doubles approximately every two years*. In more abstract terms, this means that computational power increases at this ratio. In 1995, Niklaus Wirth added the argument that the increase in computational performance triggers also the increase in software complexity (not necessarily functionality) [87]. The more complex a software system is, the more effort has to be taken into account for testing it.

1.1 On the Importance of Testing Software

Identifier	**Description**	**Year**	**Est. damage**
Mars Climate Orbiter	Different sorts of measuring units (English vs. metric) used in two steering modules.	1998	245.6 million €
Pentium floating point calculation	A faulty hardware implementation of the math co-processor on Pentium chips resulted in a possible miscalculation of decimal numbers.	1994	356.1 million €
Knight Capital trading software	Trading at the Wall street in New York (US) was disrupted for 45 minutes. A new version of the trading software placed erroneous orders at the stock market.	2012	329.8 million €
Toyota braking system	Toyota had to recall about 400.000 of their cars. A software bug caused a lag in the anti-lock-brake system.	2012	2.249 billion €

Table 1.1: Costs of famous software bugs [78] [62]

In economical terms, testing of software can be as expensive as its development or even more. The world-wide costs of fixing software errors are enormous. Table 1.1 shows a small selection of famous software errors and the estimated economical damage they resulted in. Apart from statistics and numbers, the Ariane 5 incident in 1996 still serves as a classical example for the severity of lack of testing [25]. An integer overflow in the steering software of the rocket led to the destruction of the spacecraft. Furthermore, there exists a case with far higher severity as the incident nearly had provoked World War III. In 1983 - during the cold war - the Soviet satellite early-warning system signaled that five missiles were heading to the Soviet Union [51]. The person in charge, Lt. Col. Stanislav Petrov, did not react to it as he had been supposed to, as just five missiles were in his opinion a too low number for a preventive strike from a foreign aggressor. In fact, the detection software running on the satellites did not take into account that sun rays can be reflected by clouds which the system registered falsely as missiles.

For sure, in both cases, there had been some software testing involved. But the performed tests did not uncover the defects that were found at productional run time of the respective software. The later a bug is found, the more severe its results can become.

1.2 Motivation

The steering software of Ariane 4 was reused without modifications for Ariane 5, ignoring some technical differences which caused in the end the destruction of the space craft [25]. Nevertheless, the assumption is that some testing has been performed. But the collection of tests for the failed component did not uncover this severe error. How effective can a test suite be? Is it able to uncover all kind of possible programming errors? For this purpose, test criteria like statement coverage, path coverage, etc. have been defined [13]. With their aid, an evaluation of test suites can be performed. In general, these coverage criteria check if some part was executed by any test. But does the lack of a test of some part of programming code imply that a possible programming error exists there?

In addition to the criteria mentioned above, there exists *mutation coverage*. From the original program source code, a series of automatically modified programs - called mutants - are generated (by a mutant generator). The better the existing test suite is, the more of these mutants will be uncovered by failing (at least) one of the test cases the original program passes. This software testing technique evaluates test suites by a so-called *mutation (adequacy) score*. This represents the ratio of *identified mutants* over *total number of non-equivalent mutants* [13, p.181].

1.3 Problem Description

A novel testing technique called *model-based mutation testing* has been presented by Aichernig et al. [2]. It combines model-based testing with mutation testing. Instead of generating mutants from a concrete program, the mutant generator takes an abstract model of this program (notated in a modeling language). Furthermore, the mutants will not be executed against an existing test suite but actually generate it: if there is a semantic difference between the original model (specification) and the mutated model, this can be uncovered by a specific, simultaneous execution. In fact, this execution trace represents a test. Its result indicates if a program implements one of the generated mutants and not the specification. With model-based mutation testing, a set of tests can be generated that uncovers all possible errors in a present System under Test (SUT). This generation of a distinguishing trace/test case is exactly the problem this thesis faces.

Initially, this technique requires a model which conforms to the SUT. Furthermore, a mutation generator outputs mutants of this model. Then, the set of generated mutants and the specification are *compared* in terms of refinement. The present thesis focuses only on the last issue: the generation of test cases from an existing model and corresponding mutants. In the following sections, *model-based testing* (Section 1.4), *mutation testing* (Section 1.5) and *model-based mutation testing* (Section 1.6) are outlined. The described techniques form a basis for this thesis.

Speaking in technical terms, the models (both specification and mutants) are described in a modeling language (*Action Systems*). The application developed in course of this thesis translates these models to a standardised input language for a *Satisfiability Modulo Theories* (SMT) solver. Furthermore, this compiler is extended to map the problem of refinement of two models (a specification and a mutant) to an SMT problem. Eventually, an SMT solver evaluates the problem and outputs a distinguishing trace from which a test case can derived.

1.4 Model-based Testing

In classical testing, an SUT is already present at the time of testing. Tests are then specified on a certain feature or error case of an SUT. For instance, let there be a spacecraft steering software. A range of calculations and interactions are involved in this project. As it is far too complex to understand the whole, internal structure of such a program, a black-box approach is considered by the testing department. So, how would testing then proceed? Prenninger and Pretschner state that *abstraction is the most important tool computer engineers possess*. It may be claimed that a *computer engineer / a test case writer* creates an abstraction of the program in his mind. Then, the test cases would be *derived* from this *mental model*.

Model-based testing is a black-box testing technique. Like every member of this testing kind, it emphasises the external behaviour of the *System Under Test* (SUT). In contrast to the example mentioned before, the approach is much more formal. According to Utting et al. [84], the process of model-based

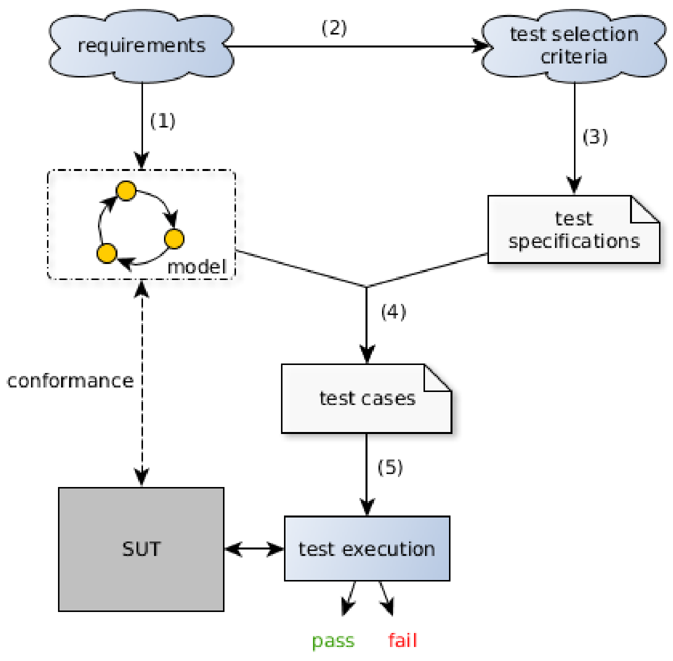


Figure 1.1: Model-based testing: the process[1] (inspired by Figure 1 in [84])

testing consists of the five steps. Figure 1.1 visualises these tasks. The description of these steps is listed below.

1. **Model creation:** initially, a model has to be extracted from the requirements of the SUT. It represents an abstract view on the SUT. The level of abstraction has to be chosen with care. It has to be *sufficiently precise to serve as a basis for generating 'meaningful' tests*. Furthermore, the authors remark that the model should be created with respect to the testing objectives. An existing design model of the SUT might not fit this task in general.

   For instance, the following requirements for an unmanned spacecraft steering software exist: the software adjusts speed and course of the rocket. After the ascend (to a certain level), angle corrections are possible. If this happens to be a dangerous correction (with respect to loss of steerability), an automatic self-destruction is triggered. Otherwise, the course is adjusted accordingly and the ascend continues. Additionally, a landing procedure can be initiated. When landed, the software is able to initiate an integrity check if another ascend is possible or not. From this specification, a behaviour model in form of a finite state machine like shown in Figure 1.2 can be constructed.

2. **Definition of test selection criteria:** the next step involves the definition (the choice, respectively) of the test selection criteria to be used. Utting et al. [84] state that criteria may focus on *system functionality (requirements-based test selection criteria), on the model structure (state and transition coverage), on data coverage heuristics, on environment properties or on stochastic heuristics such as pure randomness*. For the steering software, one criterion might be transition coverage. In this way, all possible transitions have to be tested by generated tests.

3. **Transformation to test case specification:** on the basis of an existing model and defined test selection criteria, test specifications can be derived. In case of a finite state machine like the model of the spacecraft steering software and the chosen transition coverage, test specifications are easily generated. Every transition has to be passed at least once: *engage, regular and dangerous course corrections, landing procedure, successful and failing integrity check.*

[1]All diagrams in this thesis have been created by the freeware tool yEd by the company yWorks.

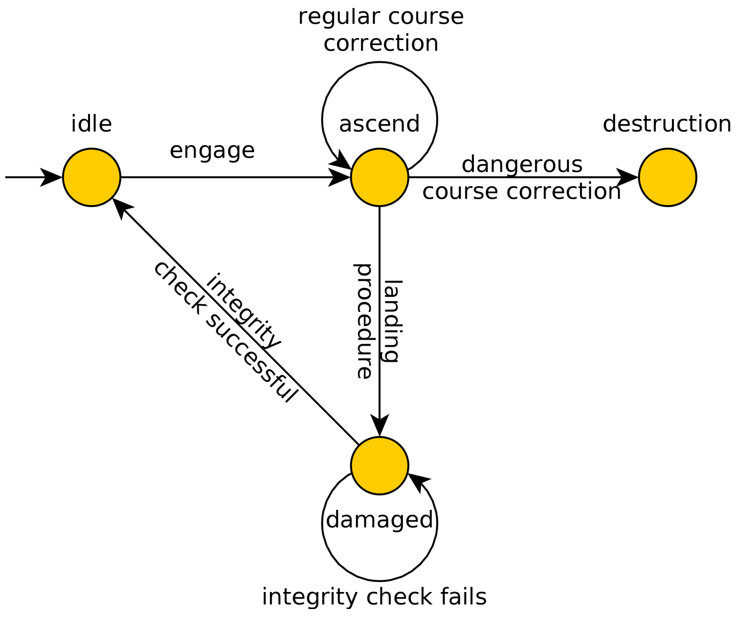

regular course correction
idle
engage
ascend
destruction
dangerous course correction
integrity check successful
landing procedure
damaged
integrity check fails

Figure 1.2: Example: a model of a spacecraft steering software

4. **Test case generation:** subsequently, test cases are generated from the test specifications by using a test generator. One test case specification will lead to a set of test cases. If the test specification is not satisfiable by the model, this set is empty. For example, if a finite state machine contains one state which is not reachable from the initial state, but there exists a transition from this state to a reachable one, no test may cover this transition. Still, a test specification for it might be generated. Utting et al. [84] argue that a test may cover more than one test specification. Some test generators even spend some effort on generating as few as possible test cases to cover all test specifications. The test specifications for the steering software example (cover all transitions) may then be covered by one single test like shown in Figure 1.3.
5. **Test case execution:** finally, the generated test suite is executed on the SUT. Test execution itself may be manual or may be automated by a *test execution environment*.

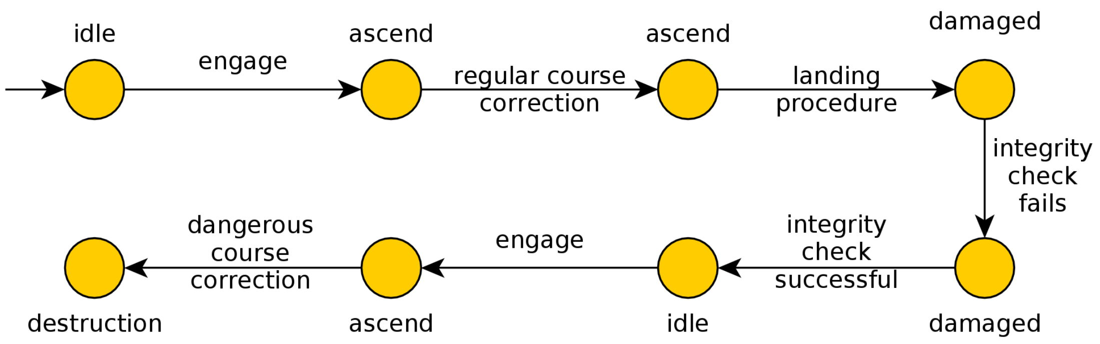


Figure 1.3: Example: a test case for the spacecraft steering software

Utting and Legeard [83] present six beneficial reasons for the appliance of model-based testing in software development:

- **SUT fault detection:** this is the most valued reason for using MBT. Comparative case studies indicate that fault detection of model-based testing performs equally good or better than manually designed test cases.

- **Reduced testing costs and time:** in general, if the time needed for *creating* and *maintaining the model* plus the time spent for *directing the test case generation* is less than the effort to manually design test cases, model-based testing is a valuable time and cost saver. Furthermore, the test generator may take care of finding the *shortest test sequences*. Hence, test execution time is kept as low as possible. Additionally, upon failure, not only the test case implementation itself can be inspected but also the abstract test. Using MBT over manual test creation may ease the search for and fixing of failures in the SUT.
- **Improved testing quality:** the quality of manually created test cases depends heavily on the skills of the engineer. A relation to the original system requirements is not easy to outline. With MBT, this process becomes more formal/systematic as test cases emerge from the existence of a formalisation of a requirement in form of a model. Also, the quality of the test cases itself depend much less on individual ingenuity. Moreover, the quantity of a series of generated test cases may be much higher than the number of manually crafted test cases.
- **Requirement defect detection:** requirements are often recorded in natural language documents. These recordings might contain *omissions, contradictions and unclear requirements*. In this sense, the construction of an abstract model may unveil these issues. Taking it to the next level, failing tests may indicate errors in the model or in the requirements.
- **Traceability:** MBT enables a relation of all items in the system: *each test case relates to the model, to the test selection criteria and even to the informal system requirements*. The model-test traceability enables

  – the extraction of transitions that are not covered by any tests,
  – the information which set of test cases covers a certain transition,
  – a visualisation of the test case as a sequence of transitions in the model (cmp. Figure 1.3).

  Moreover, the requirements-model traceability provides the information if a requirement is modeled and how this requirement influences the model. Eventually, the requirements-tests traceability

  – identifies untested requirements,
  – gives the information which set of test cases covers a certain requirement, and
  – shows the requirements that a test depends on.

- **Requirements evolution:** if there is some major change in an SUT, this has most certainly a big effect on its test cases. New requirements have to be tested, previously existing ones have to be adapted or dropped. By using MBT instead of manual test case creation, this effort can be kept on a lower level. Due to the traceability mentioned before, it is easy to determine which test case has to be *deleted* when a certain requirement is removed, for instance. The same holds also for *changed, unchanged* and *added* requirements.

Utting and Legeard [83] describe in their paper also some limitations of this testing technique. The authors state that it is not possible *to guarantee to find all non-conforming behaviour* of the SUT according to the model. Furthermore, the model creation requires some experience: the correct level of abstraction has to be chosen. Finally, the model creation itself takes some initial effort which has no counterpart in manual test case creation.

1.5 Mutation Testing

Back in 1978, DeMillo and Lipton [42] present their *competent programmer hypothesis*: *programmers do not create programs at random but ones that are close to correct*. The most common syntactical deviations from the initial code to the correct one can be recorded and generalised into code transformations. For instance, when looping through an array, the off-by-one error is classical: the chosen upper index bound exceeds the size of the array. This is simply caused by the fact that natural counting starts with the number one and indexing of a non-empty array starts with the number zero. Often, this is caused by the wrong comparison operator or the missing decrement over the length of the array. By changing the occurring *less than or equals to* relation to a *less than* relation, the resulting code might then become correct or incorrect (cmp. Listing 1.1).

```java
for(int i = 0; i <= array.size(); i++){
  System.out.println(array[i]);
}
```

Listing 1.1: Off-By-One error in Java

In general, proving correctness of a program is possible with tools like model checking and theorem proving but these often cause an unacceptable level of effort. In practice, it is often sufficient to show the absence of specific errors (and the presence of working functionality) rather than prove correctness. But what if tests do not uncover a major error like in the Ariane 5 steering software? Initially, there is an SUT which passes an existing test suite. By altering the source code of the SUT just the way previously described, a modified version of the SUT is created. This modified program represents a fault-injected SUT. The question at hand is how will the test suite behave for the modified program?

Mutation testing has been introduced by Hamlet [48] for comparing/improving the effectiveness of test suites. By looking at the behaviour of an existing test suite applied to a range of modified programs (further called mutants), this method can evaluate the quality of the test suite. With its aid, this can then be assessed directly by the so called *mutation (adequacy) score* (cmp. Equation 1.1 [54]). It represents the ratio of detected mutants (at least one of the tests fail) to the number of non-equivalent mutants. In this field of study, detected mutants are denoted as *killed* mutants hence they do not *survive* the test execution phase.

$$mas = \frac{\#killed}{\#non\text{-}equivalent} \tag{1.1}$$

The set of source code transformations - further called mutation operators - has to be chosen in a representative manner to cover classical errors like the off-by-one error (cmp. Listing 1.1). In this way, a series of mutants can be generated (manually or automatically) from the source code of the SUT. For instance, the tool *PIT mutation testing* for Java programs [37] defines a range of mutation operators (mutators) for its automated mutant generation: conditional boundary relations (contains the transformation from $\leq$ to $<$), mathematical mutators (for example from $+$ to $-$) and more (a complete list can be found on the tool's web site [37]).

When running a set of tests on a mutant (cmp. Figure 1.4), the result matches one of the following three cases:

1. If at least one test fails, the test suite *kills* the mutant. This means the test case is able to distinguish the mutant from the original SUT.
2. No test fails but the SUT and the mutants are semantically different. This indicates that a test case has to be added to the test suite. This test case should distinguish the SUT and the mutant. It passes for the correct implementation and fails for the incorrect one. There exists the possibility that not the original SUT is the correct implementation but the mutant.
3. No test fails because the original SUT and its mutant are syntactically different but equal with respect to semantics. This refers to the *equivalent mutant problem*. It will be further discussed in Section 1.5.1.

The determination between the last two cases may only be performed if the question of semantic equality of the mutant and the SUT can be answered. This will be discussed further later on (Section 1.5.1).

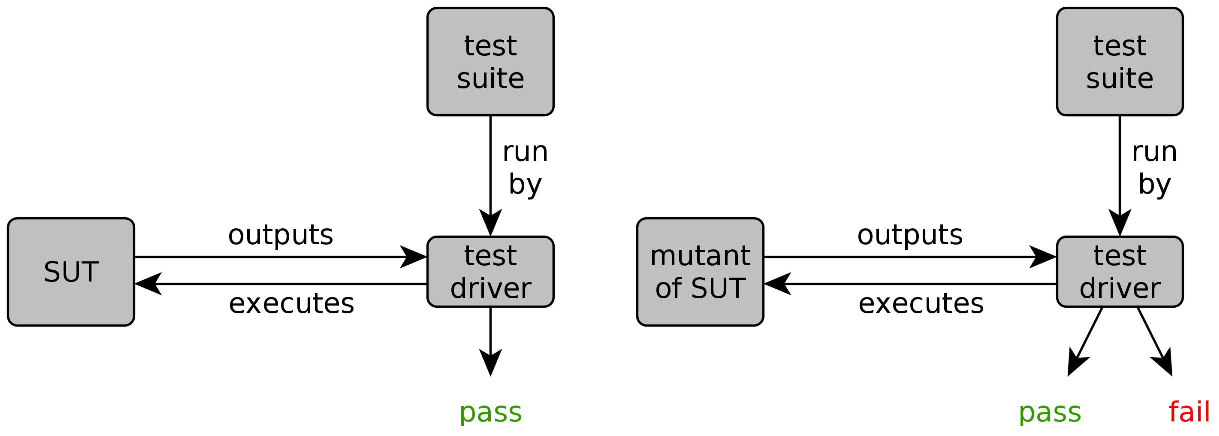


Figure 1.4: Mutation testing

In addition to the *competent programmer hypothesis*, DeMillo and Lipton [42] introduce the term *coupling effect*: *test data that distinguishes all programs differing from the correct one by only simple errors is so sensitive that it also implicitly distinguishes more complex errors*. They point out that they do not offer a prove for this effect but present empirical data to support the validity of this principle. Offutt [71] applies the idea of the coupling effect to the domain of mutation testing: in his experiments, a series of first-, second- and third-order mutants for three different programs is generated. This means that one, two or three different faults are injected into one mutant. First order mutants represent the simple errors, second and third order mutants resemble to more complex errors. His empirical results indicate:

- test data developed to kill first order mutants is very successful at killing second order mutants.
- the surviving second order mutants show no characteristics that would suggest the infeasibility of killing them by test data developed for first order mutants.
- the coupling effect increases with third order mutants. So, killing of third-order mutants with test data developed to kill first order mutants is even more successful than for second-order mutants.

In 2011, Yue and Harman [54] conducted a survey on recent papers about mutation testing. They identified three problems of this technique. First, its application is related to a *high computational effort*. Each generated mutant is run against a set of test cases. The authors give an overview on the various existing cost reduction techniques separating them into two classes: reduction of the number of mutants and reduction of the test execution time. Second, the *human oracle problem* defines the effort to check the output of the SUT to be correct. So, mutation testing might provoke a significant increase in oracle effort to be spent on further output checks, respectively test cases. Third, the equivalent mutants mark a computational, undecidable problem.

1.5.1 Equivalent Mutant Problem

One of the possible results of running a mutant against the set of existing test cases is that no test fails on the mutant. *Killing a mutant* is defined as being able to show a semantic difference between the original program and its mutant. The problem of detecting equivalent mutants is undecidable as program semantic equivalence is undecidable (classical halting problem; cmp. also [29]).

```
int greaterThanNull(int x) {
  if(x > 0){
    return 0;
  }
  return -1;
}
```

Listing 1.2: Original program

```
int greaterThanNull(int x) {
  if(x >= 1){
    return 0;
  }
  return -1;
}
```

Listing 1.3: Equivalent mutant

Listing 1.2 and 1.3 show an example for an original function and an equivalent mutant (written in the Java programming language). Both functions will return the same value for the same input. Hence, they are semantically but not syntactically equivalent.

1.6 Model-based Mutation Testing

Model-based mutation testing is a combination of the model-based and mutation testing approaches. It has been introduced by Aichernig et al. [2]. Figure 1.5 visualises the process of this novel testing technique. As in traditional model-based testing, a model is created from a set of (informal) requirements (1). Instead of deriving test selection criteria and further test specifications, mutation operators are applied to the model (2). Like in mutation testing, mutants are created this way (3). Furthermore, a test case generator tool takes the original model (further referred as *specification*) and a mutant as input (4). It checks the conformance of these two models. In case of a non-equivalent mutant, a distinguishing test case is generated (5). Finally, this test case can then be run against the SUT to show if the mutant was actually implemented or not (6 and 7).

Test case generation from two models has not occurred in neither model-based testing nor mutation testing. Therefore, this has to be discussed in more detail. Both models represent abstract versions of implementations. The intention of a generated test case is being able to distinguish an SUT implementing the correct model or the mutated model. If these two models are equal in semantics, this is not possible. In this thesis, the *Action System* formalism serves as the modeling language (cmp. Section 2.1.1). This enables also modeling of non-deterministic systems. Such systems require a different check than equivalence: a conformance relation. Aichernig and Jöbstl [9] show the inadequacy of equivalence for non-deterministic systems in a simple example: in a program, a variable out_o is set to either the value 1 or 2. A mutated version of this program sets the variable out_m to the value 2 or 3. Both decisions are chosen non-deterministically. As it is intended to check for different behaviour, the two programs should return different values. This can be fulfilled for the value combinations

1. $out_o := 1$ and $out_m := 2$
2. $out_o := 1$ and $out_m := 3$
3. $out_o := 2$ and $out_m := 3$

The combinations 2.) and 3.) mark definitely counter examples as the original system does not allow $out := 3$. But the combination 1.) is not a proper counterexample. The value $out := 2$ is allowed by the original system. These kind of false positives can be eliminated by applying a conformance relation which relies on an ordering from abstract to more concrete models. As this thesis is largely based on the work of Aichernig and Jöbstl, the same order relation is applied: refinement [7].

1.6.1 Refinement

In 1980, the refinement calculus has been introduced by Back [15]. He states that *the basic idea of this technique is to develop a program through a sequence of refinement steps, starting from a specification*

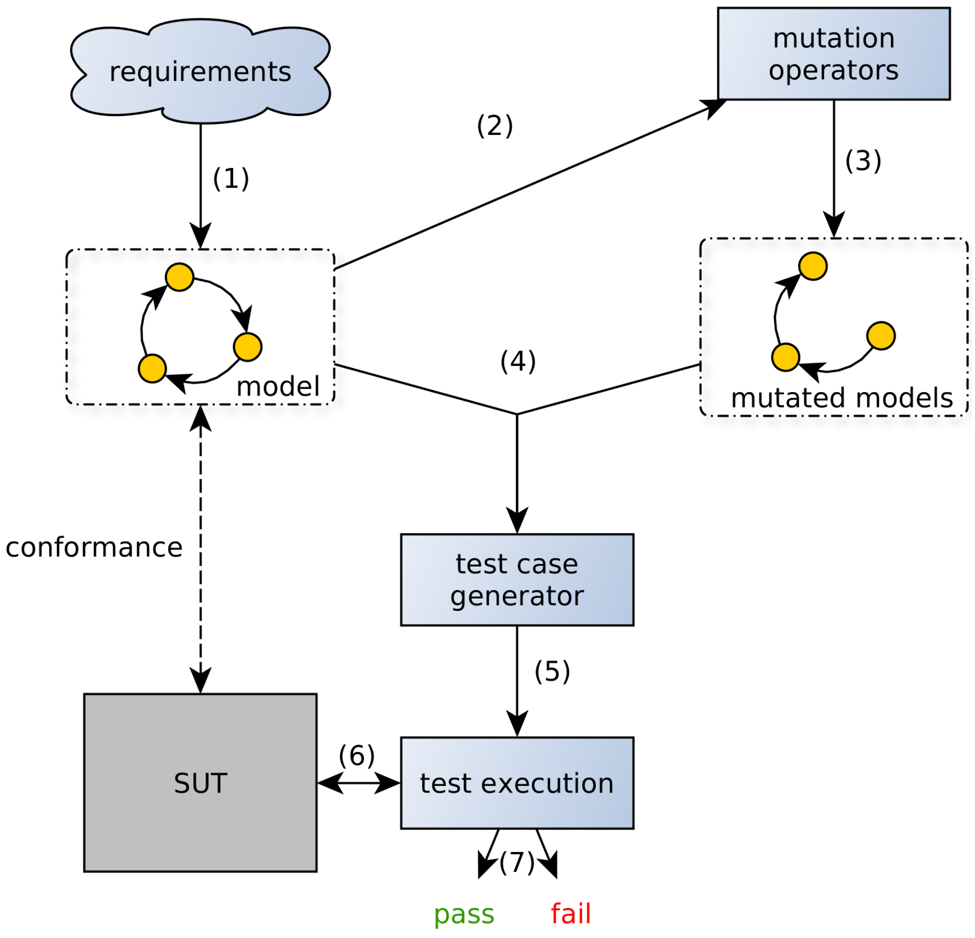


Figure 1.5: Model-based mutation testing (inspired by Figure 1 in [10])

of a program and ending up with an efficient program meeting the specification. Furthermore, Hoare and He give refinement in their *Unifying Theories of Programming* (UTP) [50] a predicative semantics. Additionally, Aichernig and Jifeng develop a mutation testing technique for UTP [6]. Based on this, Aichernig et al. give the following definition of refinement [10].

Definition 1. ***Refinement***

$$M \sqsubseteq I =_{df} \forall v, v' : I(v, v') \implies M(v, v')$$

Given a model M, an implementation I, the refinement denoted by the operator $\sqsubseteq$, and M and I being predicates over two sets of variables, refinement is defined as follows. The set $v = \langle x, y, ...\rangle$ denotes observations before execution and $v' = \langle x', y', ...\rangle$ denoting the observations afterwards. A concrete implementation I refines an abstract model M if and only if the implementation implies the model.

Furthermore, a mutant can be looked at as a concrete implementation of a model. In this way, the same conformance relation can be applied to the model of the SUT (further referred to as specification) and its mutated version: *the mutant refines the specification*. The goal of its usage in this thesis is to find a test case for every mutated model M^M which does not refine an original model M^o, so $M^o \not\sqsubseteq M^M$. For being able to get a distinguishing case, a counterexample to refinement has to be identified. This can only exist if and only if implication in Definition 1 does not hold.

Definition 2. ***Non-Refinement***

$$\exists v, v' : M^M(v, v') \land \neg M^O(v, v')$$

A mutated model M^M does not refine an original model M^O if there exist observations (i.e. state transitions) in M^M which are not allowed in M^O.

If the mutated model M^M shows a behaviour which is not allowed by the original model M^O, the state of the program in which non-refinement can be determined is called *unsafe state*.

Definition 3. ***Unsafe state***

$$u \in \{v | \exists v' : M^M(v, v') \wedge \neg M^O(v, v'))\}$$

A pre-state u identifies an unsafe state if it shows wrong (not conforming) behaviour in a mutated model M^M with respect to an original model M^O.

From these unsafe states, distinguishing test cases can be generated. An execution of such a test case will uncover if the SUT implemented the original model M^O or the fault-injected, mutated model M^M.

1.7 Thesis contribution

This thesis has been designed to be a part of the PhD thesis of Elisabeth Jöbstl[55]. Therefore, it contributes also to a project called *trust via failed falsification of complex dependable systems using automated test case generation through model mutation* (for short *TRUFAL*). This project has been funded by the national Austrian research promotion agency.

In addition, the findings of this thesis have been input for a paper called *Incremental Refinement Checking for Test Case Generation* by Bernhard Aichernig (supervisor), Elisabeth Jöbstl (co-supervisor) and Matthias Kegele (author of this thesis) [10]. In 2013, this paper had been accepted at the *Test & Proof* (TAP) conference in Budapest, Hungary.

1.8 Thesis Structure

In the previous section, the technique of model-based mutation testing has been explained. This includes the definition of refinement of models in general (Section 1.6.1). In Chapter 2.1, the modeling formalism *Action Systems* is described. It also illustrates the refinement of models described in this formalism. Subsequently, *Satisfiability Modulo Theories* are explained in short (Chapter 2.2). The standardised input language for SMT solvers (SMT-LIB) is discussed in more detail (Section 2.2.2). Then, the next three consecutive chapters document different aspects of the implemented application called *as2smt* (Action System to SMT-LIB): Chapter 3 outlines the general process the tool steps through and the modules it consists of. Chapter 4 concentrates on the translation from a Prolog notation for Action Systems to the SMT-LIB language. Chapter 5 shows how refinement of two models has been implemented. Moreover, the results of the performed case studies are presented (Chapter 6). This is affiliated by a discussion of related work (Chapter 7). Finally, a conclusion of the thesis is drawn (Chapter 8).

2 Prerequisites

As stated in the problem description (Section 1.3), a compiler is intended to be constructed. The input language for this compiler is an *Action System* notation in Prolog (cmp. Section 4.1). The modeling language is presented in Section 2.1. Then, a closer look is taken at *Satisfiability Modulo Theories* (SMT, Section 2.2), SMT solvers (Section 2.2.1) and the output language for the compiler - SMT-LIB (Section 2.2.2).

2.1 Modeling Language: Action Systems

The modeling formalism which has been chosen to be used in this thesis is the *Action System* formalism [17]. The following sections will discuss this formalism in general and its formal semantics in refinement.

2.1.1 Action System Formalism

Back and Kurki-Suonio introduced this formalism in 1983 [17] and revised it in 1988 [18]. Its intent is to describe process nets in form of joint actions. One action consists of processes directly linked together. The execution of an action results in value changes of the local variables of the process. Every action is accompanied by an enabling condition (so called *action guard*). It may depend on any variables of the system (global variables). By deciding that the evaluation of the enabling condition of every action is performed as an atomic operation, the actions can be executed sequentially. If there is more than one enabled action, it is chosen non-deterministically which action is executed. In case of Action Systems, formal semantics is defined by Back and Kurki-Suonio in terms of weakest precondition [17]. However, Aichernig and Jöbstl defined them by relational predicative semantics [8]. For the purpose of testing, this semantics also includes trace information of the executed action which does not exist in Back's original version. For better understanding, the semantics is explained in addition to the formal definition by an example.

Description of example Action System

```prolog
% type declarations
type(bool, X) :- X in 0..1, labeling([],[X]).
type(int_0_2, X) :- X in 0..2, labeling([],[X]).
type(int_0_4, X) :- X in 0..4, labeling([],[X]).
type(n, X) :- X in 0..1, labeling([],[X]).

% variable declarations
var([engine], bool).
var([integrity], int_0_2).
var([state], int_0_4).
var([x, y], n).

% definition of the state
state_def([engine, integrity, state, x, y]).
% initial state values
init([0, 0, 0, 0, 0, 0]).
```

Listing 2.1: Rocket steering software: type and variables declarations and initial values[1]

[1]The expression *labeling([],[X])* is an enumeration predicate of Prolog. It enables the search for values in bounded domain [30]. For this thesis, this has no meaning. Nevertheless, this is part of the input language definition (cmp. Section 4.1).

In the implemented tool, a Prolog notation for Action Systems is used (cmp. Appendix A). In general, this notation consists of five parts: type declarations, variable declarations, initial state assignment, action declarations and the do-od block definition. The example presented in Listing 2.1, 2.2 and 2.3 is written in a simplified version of this Prolog notation[2]. It shows the Action System of the rocket steering software presented in Figure 1.2.

```
% actions
actions (
  'engage' :: (integrity = 0 ∧ engine = 0 ∧ x = 0 ∧ y = 0) => (
    state := 1,
    engine := 1,
    x := 1,
    y := 1
  ),

  'course_correction'(x_next, y_next)::(integrity = 1 ∧ engine = 1) => (
    % irregular course correction, destruction
    (((x - x_next < 0) ∨ (y - y_next < 0)) =>
      state = 2,
      integrity = 2)
    ;
    % regular course correction
    (((x - x_next ≥ 0) ∧ (y - y_next ≥ 0)) =>
      state := 1,
      x := x_next,
      y := y_next
    )
  ),

  'land' :: integrity = 0 => (
    state := 3,
    engine := 0,
    (
      % perfect landing
      integrity := 0
      ;
      % damage on landing
      integrity := 1)
  ),

  'repair' :: (engine = 0 ∧ integriy = 1) => (
    % repair successful
    integrity := 0
    ;
    % repair fails
    integrity := 1
  ),

  'integrity_check' :: (integrity = 0 ∧ state = 4) => (
    % back to idle
    state := 0
  )
)
```

Listing 2.2: Rocket steering software: action declarations

[2]ignoring any Prolog code unnecessary for this illustration

Listing 2.1 shows the type definition and variable definitions and the initial values of the these variables. The used types are defined as subsets of Integer values. The Boolean type $bool$ maps $false$ and $true$ to the values 0 and 1. For enumeration variables, the types int_0_2 and int_0_4 allowing values between 0 and 2, 0 and 4 respectively. The Boolean value $engine$ indicates if the engine is turned on (0) or off(1; Line 8). $integrity$ signals the operability of the space craft: intact (0), damaged (1) and destroyed (2; Line 9). The state resembles to the state in the state graph (Figure 1.2): idle (0), ascend (1), destructed (2) and damaged (3; Line 10). x and y identify the coordinates of the current position of the rocket (Line 11). In Line 14, the set of variables is defined to form the state. Initially, this state is declared by all variables set to 0 (Line 16).

```
% do-od block
dood (
  engage
  ; [A:n,B:n] : course_correction(A, B)
  ; land
  ; repair
  ; integrity_check
)
```

Listing 2.3: Rocket steering software: do-od block

Predicative Semantics

For this thesis, the mentioned predicative semantics defined by Aichernig and Jöbstl has been chosen [8] which include traces. So, observations are represented by the system states (a vector of variables) and event traces before and after one execution of the do-od block. These states are denoted by

- $(\bar{v}, tr)$ to describe a pre-state
- $(\bar{v'}, tr')$ to describe a post-state

In the following, the semantics is explained per expression type. In the definitions, the left hand side shows the syntax of the element, the right hand side its semantic meaning.

Definition 4. ***Action***

$$l :: g => B \quad =_{df} \quad g \wedge B(\bar{v}, \bar{v'}) \wedge tr' = tr ^\frown [l]$$

An action consists of a label l, a guard g and a body B. Before execution, the trace tr represents the list of all called actions before. The conjunction of the guard g, the body of the action B and addition of its label l to the previously observed trace tr represent the guarded action's transition relation. The post-state value of tr is held in the trace variable tr'.

Listing 2.2 displays the action definitions for the space craft steering software example (cmp. Figure 1.2). The individual transitions are described as actions in an *actions* block. The first action in this block is the action labeled with `engage` (l, Line 3). At the same line, two colons separate it from the guard condition g `(integrity = 0 ∧ engine = 0 ∧ x = 0 ∧ y = 0)`. Subsequently, the body of the action is defined (Lines 4 - 7).

Definition 5. ***Action with parameters***

$$l(\bar{X}) :: g => B \quad =_{df} \quad \exists \bar{X} : (g \wedge B(\bar{v}, \bar{v'}) \wedge tr' = tr ^\frown [l(\bar{X})])$$

If an action is parameterised, the parameters $\bar{X}$ are added as local variables to the predicate. The assigned values to $\bar{X}$ are also added to the post-state trace variable.

Among the listed actions in Listing 2.2, there is also one action with parameters. It is labeled `course_correction` and takes two parameters identified by `x_next` and `y_next`. The type of the parameters is defined in the action call in the do-od block (Line 4, Figure 2.3).

Inside the body B of an action, four types of operations are permitted: assignment, sequential composition, non-deterministic choice and guarded command.

Definition 6. ***Assignment***

$$x := e \quad =_{df} \quad x' = e \wedge y' = y \wedge ... \wedge z' = z$$

An assignment updates a local variable x to a value e. All other variables are left unchanged.

Listing 2.2 contains a range of examples for assignments. For instance, at Line 4, the variable `state` is set to the value 1. In general, e can also be any arithmetic expression which is defined in Appendix A.

Definition 7. ***Sequential composition***

$$B_1, B_2 \quad =_{df} \quad \exists \bar{v_0} : B_1(\bar{v}, \bar{v_0}) \wedge B_2(\bar{v_0}, \bar{v'})$$

When two bodies of processes, each with a pre-state $\bar{v}$ and a post-state $\bar{v'}$ are concatenated by a sequential composition, the following transition relation applies: There exists an intermediate state between the two bodies. The first block changes the local variables from $\bar{v}$ to the intermediate state $\bar{v_0}$ and the second one from $\bar{v_0}$ to the post-state $\bar{v'}$.

In the Prolog notation for Action Systems, the comma symbol denotes a sequential composition. The body B of the `engage` action consists of sequential compositions of assignments (Lines 4 -7). The comma operator is left-associative. This means that the first sequential composition to be executed is `state := 1 , engine := 1`. Before the execution of the first assignment, the Action System has the state $\bar{v}$. When updating the value of $state$ to 1, the intermediate state $\bar{v_0}$ is reached. Then, the second assignments set the variable $engine$ to the value 1. After the execution of both assignments, Action System is in the post-state $\bar{v'}$. This procedure is carried on with the second composition, having the post-state after the execution of the first sequential composition as an intermediate state $\bar{v_0}$ for the second one. Section 4.3.4 lists an example for sequential composition.

Definition 8. ***Non-deterministic choice***

$$B_1; B_2 \quad =_{df} \quad B_1(\bar{v}, \bar{v'}) \vee B_2(\bar{v}, \bar{v'})$$

A non-deterministic choice between two bodies results in terms of predicative semantics in a disjunction of the two elements.

A non-deterministic choice operation is indicated by a semicolon. The body of the `repair` action contains this left-associative operator (Lines 36 - 40): `integrity := 0 ; integrity := 1`. It can not be determined which of the two bodies (consisting each of just one assignment) is executed at run time. For the rocket steering software example, the support for non-determinism in the modeling language is of importance. Modeling the possibility that the space craft is *possibly* damaged is difficult, otherwise.

Definition 9. ***Guarded command***

$$g => B \quad =_{df} \quad g \wedge B(\bar{v}, \bar{v'})$$

Any body of processes B can be guarded by an enabling condition g. Similarly to the guarded action's transition relation, for the guarded command transition relation, the conjunction of the body B and its accompanied guard g must hold. The only difference here is that no trace variable is involved as it is part of an action itself.

Just like for actions, a guard condition g and a body B is separated by an arrow symbol =>. Again, Listing 2.2 contains an example for this. In Line 12, there is the guard condition to check whether an irregular course correction occurred. If this condition g holds, then the two assignments (Lines 13 & 14; connected by sequential composition) are executed.

The previous definitions show the elements contained in an action. The do-od block provides the event-based view on the Action System: the labels (appended by possible parameters) of the actions are composed here to indicate the possible enabling of the action. In general, the do-od block consists of non-deterministic choices between action calls. If sequential composition was also allowed, the joined actions might be thought of one virtual action, combining the bodies of both action. In this thesis, only non-deterministic choice is considered in the do-od block.

Listing 2.3 presents the do-od block for the space craft example. As mentioned before, just one of the actions is chosen to be executed. This holds also if multiple actions are enabled. In this case, one action is chosen non-deterministically (i.e. interleaving semantics). If none of the actions listed in the do-od block can be enabled, the Action System terminates. Otherwise, the Action System continues with executing actions.

2.1.2 Refinement of Action Systems

In Section 1.6.1, refinement has been explained in general and how to use it as a criterion to identify unsafe states. Applying Definition 1, an Action System AS^m refines its original AS^o if and only if every observable behaviour of the mutant is allowed by the original. Like in a *traditional* program, an Action System may not reach all states (in the sense of value combinations of all variables). For this reason, it has to be dealt with reachability when defining refinement.

Definition 10. ***Refinement of Action Systems***

$$AS^o \sqsubseteq AS^m \quad =_{df} \quad \forall \bar{v}, \bar{v'}, tr, tr' : ((\bar{v} \in \mathit{reachable}(AS^o, tr) \wedge P(\bar{v}, \bar{v'}, tr, tr')^m) \implies P(\bar{v}, \bar{v'}, tr, tr')^o)$$

*The function **reachable** returns the set of reachable states for a given Action System AS^o and a corresponding event trace for each state. Let an Action System AS^m with the do-od block P^m refine the Action System AS^o with the do-od block P^o. Then, any reachable state change (v to v') with an event trace (tr to tr') that is valid in the implementation/mutant AS^m, has to hold also on the specification AS^o.*

So, every state has to be a reachable state (according to AS^o) and enable at least one action in P^m. This implies that also an action in the do-od block P^o is enabled. Considering the fact that a do-od block consists of a non-deterministic choice of actions A_i, P^o and P^m can be substituted. For example, let there be a mutation, called $rocket^m$, of the rocket steering software Action System (further referenced as $rocket^o$). The do-od block described in Listing 2.1 is changed slightly. In Line 4, the call to the action `course_correction` is modified to `[B:n] : course_correction(1, B)`. In this way, *course corrections* are only possible in y-axis direction as the parameter `x_next` is fixed to the constant value 1. Then, $rocket^m$ refines the original described one. Any event trace valid in $rocket^m$ holds also for $rocket^o$ as the mutant differs only in a limitation of a parameter.

The negation of Definition 10 leads to the non-refinement condition for two Action Systems.

$$\exists \bar{v}, \bar{v'}, tr, tr' : (\bar{v} \in \mathit{reachable}(AS^o, tr) \wedge (A_1^m(\bar{v}, \bar{v'}, tr, tr') \vee ... \vee A_n^m(\bar{v}, \bar{v'}, tr, tr')) \wedge \neg A_1^o(\bar{v}, \bar{v'}, tr, tr') \wedge ... \wedge \neg A_m^o(\bar{v}, \bar{v'}, tr, tr'))$$

By looking at each action A_i^m of AS^m individually (application of the distributive law), a set of constraints is created which have to be checked to detect non-refinement.

Theorem 1. ***Non-Refinement of Action Systems***

$$AS^o \not\sqsubseteq AS^m \quad \Longleftrightarrow \quad \bigvee_{i=1}^{n} \exists \bar{v}, \bar{v'}, tr, tr' :$$
$$(\bar{v} \in reachable(AS^o, tr) \wedge A_i^m(\bar{v}, \bar{v'}, tr, tr') \wedge \neg A_1^o(\bar{v}, \bar{v'}, tr, tr') \wedge ... \wedge \neg A_m^o(\bar{v}, \bar{v'}, tr, tr'))$$

A mutated Action System AS^m does not refine its original AS^o iff any action A_i^m of the mutant shows trace or state-behaviour that is not possible in the original Action System [8].

If such an event trace or state-behaviour is found as it is described in Theorem 1, then an unsafe state is identified. Let there be another mutant of the space craft steering software $rocket^o$ called $rocket^{m2}$. By changing the line 37 in Listing 2.2 to the same as Line 40, the action `repair` is not able to set the `integrity` flag back to 0. So, when landing resulted in a damage, we reach a state in which the guard of the `integrity_check` action may never again evaluate to true. Actually, this state marks the unsafe state. Hence, the Action System $rocket^{m2}$ does not refine $rocket^o$.

2.2 Satisfiability Modulo Theories

Determining, whether an arbitrary propositional formula is satisfiable (SAT), is a well-known NP-complete problem. Satisfiability modulo theories (SMT) generalises Boolean satisfiability (SAT) by adding equality reasoning, arithmetic, fixed-size bit-vectors, arrays, quantifiers, and other useful first-order theories [40].

The intention of the usage of SMT in this thesis is to map the problem of finding counterexamples from the modeling language Action System (cmp. Chapter 2.1) to the SMT-LIB language. An SMT solver is a tool for deciding the satisfiability of formulas in these theories. By using an SMT solver, the problem of finding counter examples when checking bounded refinement of two Action Systems is solvable in an automated way.

At first, this chapter outlines the basic algorithms used in SAT & SMT solvers. Then, the SMT-LIB language is explained thoroughly. Finally, the motivation for using SMT-LIB is explained.

2.2.1 SAT & SMT Solvers

As mentioned before, SMT solvers evolved from SAT solvers by enriching SAT problems by domain theories. SAT & SMT solvers may be based on various techniques. This section gives a first look on general algorithms implemented in SAT and SMT solvers. It has no claim for completeness but serves as a small overview of the techniques applied in such solvers. Huth and Ryan give more information on this in [52].

Linear SAT solving

Huth and Ryan describe a marking algorithm with unit propagation as an example for a simple SAT solver [52, p. 68-72]. With this algorithm, SAT solving can be performed. At first, the input constraint parse tree is transformed into a directed acyclic graph (DAG). It contains only one node per variable (all occurrences link to the same node), negations and conjunctions. Furthermore, the root of this DAG is marked with $1 : T$. This identifies the root node evaluated to true which is exactly the condition that has to be fulfilled to decide satisfiability. As a next step, this truth value is propagated by the following rules:

- **negation** $\neg p$**:** Negation propagation leads to a negation of the marked truth value. Propagation may be performed in both directions ($\neg p$ marked as truth value x, p is marked as $\neg x$).
- **conjunction** $p \wedge q$**:**

– A true conjunction $p \land q$ forces true conjuncts p and q.
– True conjuncts p and q force true conjunction $p \land q$.
– False conjuncts p and q force false conjunction $p \land q$.
– False conjunction $p \land q$ and true conjunct p (or q) force false conjunct q (or p).

These rules are applied until either all nodes in the DAG have been marked properly or a conflict has been determined. In case the marking has been completed, a valuation for all variables in the Boolean formula has been found. So, the SAT solver has proven the satisfiability of the given constraint. If the SAT solver stops by discovering a conflict or no further marking is possible, it simply fails. This is called a linear SAT solver as the running time of the solver scales linearly with the input formula.

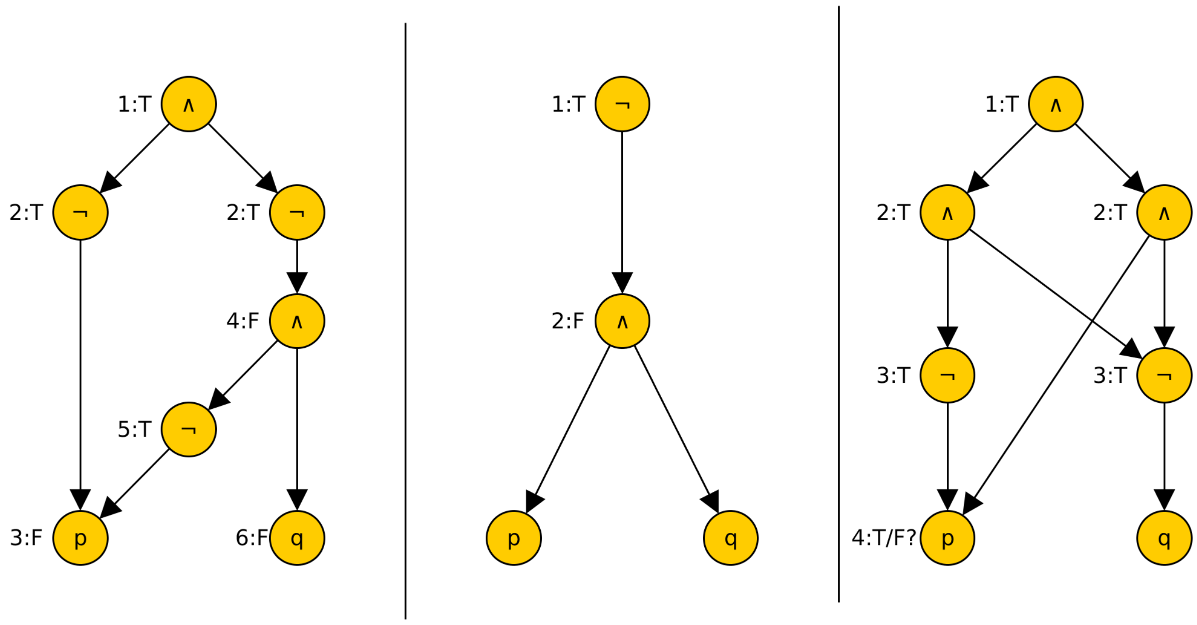


Figure 2.1: Linear SAT solver: SAT (left), STUCK (middle), UNSAT (right)

Figure 2.1 illustrates three examples. The first example on the left side presents the expression $\neg p \land \neg(\neg p \land q)$ as a DAG. At first, the rule *true-conjunction lead to true conjuncts* is applied. Then, the negation rule is applied three times ($\neg p$ marked as true leads to p marked false). Finally, all nodes have been marked with a value and so also the assignments for the Boolean variables have been found. So, a SAT solver (working with the described algorithm) returns *satisfiable* for the left example. The second example in the middle shows the Boolean expression $\neg(p \land q)$. It stops after marking the root node and applying the negation rule once because the expression $p \land q$ marked with false can not be marked any further (according to the defined rules). In this case, a solver returns UNKNOWN or STUCK as it is not able to valuate all nodes. The third example on the right side illustrates the DAG of the expression $(\neg p \land \neg q) \land (p \land \neg q)$. Here, the marking algorithms is applied as before but the marking leads to a conflict. By applying two rules, two different values should valuate a node. In this case, a solver determines that the expression at hand is unsatisfiable as this conflict can not be resolved.

Cubic SAT solving

A cubic SAT solver continues when the linear SAT solver is not able to apply any further markings. It temporarily sets a node to a concrete truth value and proceeds with the application of the marking

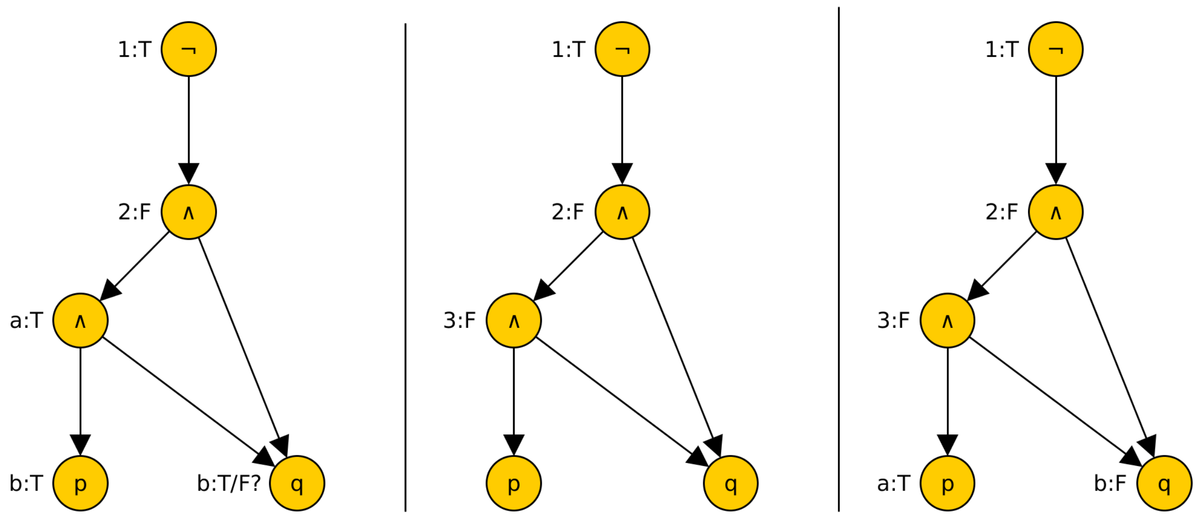


Figure 2.2: Cubic SAT solver: Conflict in temporary marking (left), making temporary marking permanent (middle), SAT with temporary markings (right)

rules. If the marking stops with a conflict, the opposite truth value is chosen as new temporary value. Otherwise, the marking becomes permanent. In this way, just the answers SAT or UNSAT are possible. Figure 2.2 shows possible cases for the cubic SAT solver [52, p. 72ff].

Davis-Putnam-Logemann-Loveland (DPLL) Algorithm

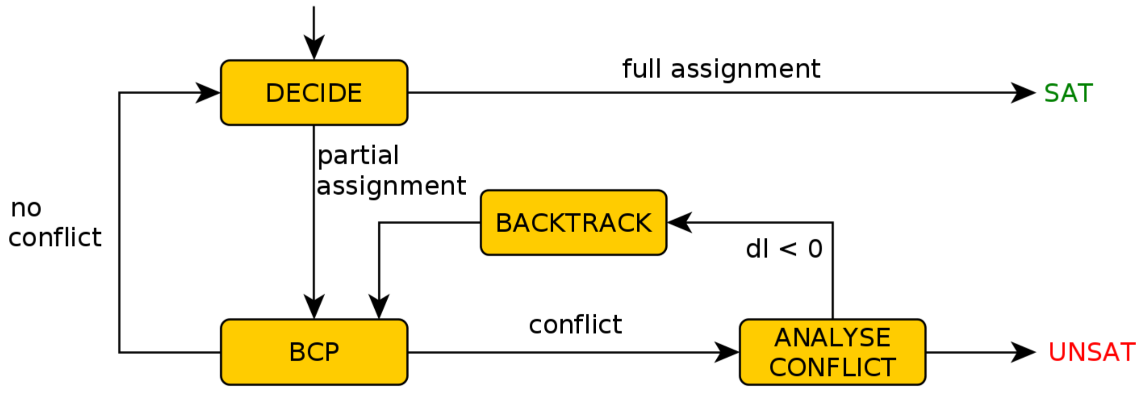


Figure 2.3: DPLL-SAT: high-level overview of the Davis-Putnam-Loveland-Logemann algorithm. The variable dl is the decision level to which the procedure backtracks [61]

In 1960, Davis and Putnam presented their *computing procedure for quantification theory* [39]. Furthermore, Davis, Logemann and Loveland suggested some modifications to the algorithm resulting in the Davis-Putnam-Logemann-Loveland algorithm [38]. It is able to decide satisfiability of propositional formulas in conjunctive normal form (CNF). Figure 2.3 illustrates its application:

- **DECIDE** assigns a truth value to a selected variable. The decision is made on the basis of heuristics. If no further decisions can be made because all variables have been set, the algorithm terminates and the result is SAT. Otherwise, the algorithm proceeds with BCP.

- **BCP** (Boolean constraint propagation) applies repeatedly the *unit clause*, i.e. assigns single unassigned variables to Boolean values with the goal to fulfill a formula. When propagation ends, it continues with DECIDE. If propagation uncovers a conflict, this is analysed (*ANALYSE-CONFLICT*).
- **ANALYSE-CONFLICT** investigates which backtracking point (decision level) has to be recovered to continue processing. If this action is performed at decision level 0, the algorithm terminates. This implies that the input formula has been found unsatisfiable. Otherwise, the decision level to backtrack to is taken as an input for the BACKTRACK action.
- **BACKTRACK** resets all assignments to the input decision level which becomes the current decision level.

1	$\emptyset$	$\neg a \vee b,$	$\neg b \vee \neg c \vee \neg d,$	$c \vee \neg d$	$\Rightarrow$	DECIDE
2	$\boldsymbol{a}$	$\neg \boldsymbol{a} \vee b,$	$\neg b \vee \neg c \vee \neg d,$	$c \vee \neg d$	$\Rightarrow$	BCP
3	$\boldsymbol{a}, b$	$\neg \boldsymbol{a} \vee b,$	$\neg b \vee \neg c \vee \neg d,$	$c \vee \neg d$	$\Rightarrow$	DECIDE
4	$\boldsymbol{a}, b, \boldsymbol{c}$	$\neg \boldsymbol{a} \vee b,$	$\neg b \vee \neg \boldsymbol{c} \vee \neg d,$	$\boldsymbol{c} \vee \neg d$	$\Rightarrow$	BCP
5	$\boldsymbol{a}, b, \boldsymbol{c}, d$	$\neg \boldsymbol{a} \vee b,$	$\neg b \vee \neg \boldsymbol{c} \vee \neg d,$	$\boldsymbol{c} \vee \neg d$	$\Rightarrow$	BACKTRACK
6	$\boldsymbol{a}, b, \neg \boldsymbol{c}, \neg d$	$\neg \boldsymbol{a} \vee b,$	$\neg b \vee \neg \boldsymbol{c} \vee \neg d,$	$\boldsymbol{c} \vee \neg d$		final state: model found $\square$.

Table 2.1: DPLL-SAT example

Table 2.1 applies DPLL (SAT) to the formula $(\neg a \vee b) \wedge (\neg b \vee \neg c \vee \neg d) \wedge (c \vee \neg d)$. On the left, there is the set of assigned values. If the variable has a prefixed $\neg$ symbol, the assigned value is *false*. Otherwise, true is assigned. Bold symbols represent variable values that have been decided (by the *DECIDE* action). Initially, there are no variables set to any values (Line 1). The algorithm starts with the DECIDE action and sets the variable a to true (Line 2). As DECIDE is not able to generate a full assignment, BCP is applied which leads to an assignment of b to true (Line 3). No further markings are possible with BCP. Therefore, the algorithm continues with DECIDE. Again, a variable is chosen. This time, c is set to true (Line 4). By executing the unit propagation, d is set to true (Line 5), but this leads to a conflict: the first and third term evaluate to true by setting d to true, but the second term would evaluate to false. Subsequently, the conflict is analysed and resolved by back tracking and setting c to false and also d to false. The solver outputs satisfiable and is able to present the model $a = true, b = true, c = false$ and $d = false$.

In 2004, Nieuwenhuis et al. [67] state that most state-of-the-art SAT solvers were based on different variations of the DPLL algorithm [66] [46]. Furthermore, they modeled the DPLL algorithm and its existing extensions as *transition systems* in an abstract version. The current implementations (2012) of the SMT solvers used in this thesis implement this algorithm framework [40] [33] [34] (with some adaptions). Among the extensions Nieuwenhuis et al. described, is clause learning: when backtracking from a level j to i, a temporary clause is added to the existing ones. This clause, called *back jump clause*, prevents decisions which led to backtracking. Furthermore, it enables the possibility to reverse more than one decision level. This back jump clause can determined for example by a conflict analysis (analysing the conflict graph,[63]). The algorithms implemented in SMT solvers are constantly evolving.

2.2.2 SMT-LIB

In 2003, Ranise and Tinelli proposed the initial version of the SMT-LIB format [75]. Its purpose has been to *produce* an online library of benchmarks for satisfiability modulo theories. Furthermore, a standardised input language for SMT solvers has been defined, in which these benchmark were written. A similar approach has been performed for SAT solvers. The standardised input language for SAT solvers is the *DIMACS* format[3]. In 2010, Barett et al. released the specification SMT-LIB in Version 2 [24]. This is the version that has been used in this thesis. The version number will not be further noted when referencing SMT-LIB in the following. It defines

- *a language for writing terms and formulas in a sorted (i.e., typed) version of first-order logic*
- *a language for specifying background theories and fixing a standard vocabulary of sort, function, and predicate symbols for them*
- *a language for specifying logics, suitably restricted classes of formulas to be checked for satisfiability with respect to a specific background theory*
- *a command language for interacting with SMT solvers* [24, p.13]

The syntax of SMT-LIB is structured exactly like so called *symbolic expressions* (s-expressions) originating from the LISP programming language. Every s-expression starts with an opening and ends with a closing parenthesis. Operators and operands are separated by a blank character, the operator notation is a prefix-based one. Example: `(and a b)`.

Underlying Logic

SMT-LIB logic is based on a *many-sorted first-order logic with equality*. It contains sorts (i.e. basic types) and sorted terms. Deviant from standard many-sorted first order logic, there is no syntactic category of formulas distinct from terms. Formulas are interpreted as sorted terms of a distinguished Boolean sort which is interpreted as a two-element set in every SMT-LIB theory. Moreover, the definition of sorts is not limited to constants like `Int`, but are defined as sort terms. This enables the use of composed data structures like `(List (Array Int Real))`. Another feature inspired by traditional programming languages is the *let* binder which can be seen as a local variable binder. Additionally, existential and universal quantifiers can be used to write formulas.

Background Theories

In SMT-LIB, there exist two kinds of theories: basic and combined. The standard explicitly declares six basic theories. In a use case, not a single basic theory but a combination of those theories might fit best for the problem at hand. This combination can be carried out implicitly by a general modular combination operator. The current available basic theories are listed on the SMT-LIB web page [23]:

- **Core:** core theory, defining the basic Boolean operators
- **Ints:** integer numbers with basic arithmetical (plus *div*, *mod* and *abs*) and relational operators
- **Reals:** real numbers with basic arithmetical and relational operators
- **Reals_Ints:** combination of *Ints* and *Reals* theories with additional functions (*to_real*, *to_int*, *is_int*)
- **FixedSizeBitVectors:** bit vectors with arbitrary but constant size
- **ArraysEx:** functional arrays with extensibility

Theory specifications have mostly documentation purposes [24]. Their intention is to be a standard reference for human readers. Only the set of sorts and the sorted function symbols are specified formally.

[3]The DIMACS acronym and the format origin from the *Center of Discrete Mathematics & Theoretical Computer Science* from the Rutgers University in New Brunswick, USA. [14]

Logics

Every SMT-LIB assertion is always interpreted and executed in the context of a logic ([23], tutorial document). In addition to the core SMT-LIB logic (cmp. Section 2.2.2), a *sub-logic* can be defined. Such a logic declaration includes apart from a textual description a reference to the used theories. Additionally, it may contain extensions in form of sorted function symbols. For example, the logic *AUFNIRA* is based on the theories *Reals_Ints* and *ArraysEx* and defines three sorted function symbols (intended to be syntactic sugar) as extensions. Like the theories described in Section 2.2.2, all supported logics are listed on the SMT-LIB web page. The naming of the logics seems initially a bit cryptic but the different capital letters mark either the implementation of a theory, a limitation or an extension:

- **QF:** Quantifier-Free (limitation)
- **UF:** Uninterpreted Functions (extension)
- **IA:** Integer Arithmetic (*Ints* (theory))
- **RA:** Real number Arithmetic (*Reals* (theory))
- **IRA:** Integer and Real number Arithmetic (*Reals_Ints* (theory))
- **L or N:** Linear or Non-linear
- **DL:** Difference Logic (limitation)
- **BV:** Bit Vectors (*FixedSizeBitVectors* (theory))
- **A:** Arrays (*ArraysEx* (theory))

Additionally, there might be further restrictions and extensions of a logic [36]. This is described in the textual logic description. Any combination of the listed limitations and extensions are possible per theory. The name of the theory then consists of a concatenation of the letters. In general, there exist the following categories:

- **Boolean logics:** a single listed logic supports only the *Core* theory: *QF_UF* - the logic of Quantifier-Free Uninterpreted Functions. It restricts the SMT-LIB logic (cmp. Section 2.2.2) to quantifier free expressions.
- **Logics with linear arithmetic:** a set of logics is defined which can deal with arithmetical expressions. Two of the most popular according to the SMT competition are listed bellow:
  - *QF_UFLIA* is limited to linear integer arithmetic without quantifiers: multiplication is only possible if one factor is a constant value.
  - *QF_UFLRA* represents *QF_UFLIA* with real numbers.
- **Logics for difference arithmetic:** instead of allowing all kinds of arithmetic expressions (cmp. previous category), difference arithmetic logics only allows the comparison between numeric values or the comparison of a difference between two numeric values to a positive or negative literal [23].
- **Logics with bit vectors:** instead of integers and real numbers, also bit vectors can be used. The most basic representative of this family is the *QF_BV* logic. Like for arithmetic logics, extensions exist like *QF_UFBV* (*QF_BV* plus the support of uninterpreted functions).
- **Logics with arrays:** this kind of logics enable the support of arrays of sorts, allowing to define a sort for the index and the value separately. Any arithmetic, difference or bit vector logic may serve as basis (for example *AUFLIA* or *AUFNIRA*).

Command Language

The command language of SMT-LIB conforms to the format of writing terms and defining logics. It defines various commands which have to be interpretable by a solver implementing the SMT-LIB language. There exist the following types of commands:

- **Solver options**: every solver may be configured by parameters. In respect to the previous section, the command `set-logic` is of importance: it sets the logic to be used. It is obligatory to set this command before any others (with few exceptions) as they might be dependent on the choice of the logic. Further configuration parameters can be set by `set-option` commands. It has to be distinguished between standard options defined by the SMT-LIB specification and solver-specific options. The implemented application makes only use of the standard option **produce-models** to enable the retrieval of a model from the solver in case the assertions are in a satisfiable state. The complete list can be obtained from the SMT-LIB specification document [24].
- **Definition and declaration of sorts:** apart from the standard sorts Boolean, Integer, Real and Bit Vector, additional sorts can be defined. This way, more complex data structures can be created.
- **Definition and declaration of symbols:** depending on the logic, function symbols may be declared by `declare-fun`. It is also used to declare a constant which is just a function without parameters [41]. *Microsoft's Z3* even introduces syntactic sugar for declaring constants - `declare-const` - to omit the braces of the empty parameter list (`(declare-const x Int)` instead of
`(declare-fun x () Int)`) [41].
- **Assertion commands:** a solver implementing SMT-LIB maintains a stack of the asserted formulas, declarations and definitions. Assertions are added through the `assert` command (example: `(assert (= x y))`). The satisfiability of all assertions on the stack can be issued by the command `check-sat`. It outputs the classical (SAT solver) response *SAT*, *UNSAT* or *UNKNOWN*. A feature of SMT-LIB compliant solvers is the support of *incremental solving*: it is not only possible to add assertions from the stack but also to remove some (in fact to backtrack to a previous state of the stack). With `push`, a backtrack marker is set on the assertion stack. By issuing a `pop` command, the state, when the most recent marker was set, is restored. This feature has been used heavily in the implemented application (cmp. Section 5.3.2).
- **Inspection of proofs and models:** the previously mentioned `check-sat` command tries to find an assignment to every symbol to satisfy all assertions on the stack. This assignment can be output by the `get-assignment` command. Furthermore, there exist the commands `get-proof` to get a proof of unsatisfiability and `get-unsat-core` to get the set of assertions that the solver determined to be unsatisfiable. These two commands have not been used in the course of this thesis.

3 The as2smt Tool: Overview and Architecture

This chapter is structured as follows: at first, the environment of the tool implemented in the course of this thesis will be outlined. Then, the different modules of the tool and their internal responsibilities will be discussed. Finally, an installation and usage guide is provided.

3.1 Environment

In this section, the tools used in the course of this thesis are listed. Furthermore, a closer look is done on the implementation language. At last, a process overview of the *Action System to Satisfiability Modulo Theories (as2smt)* application is presented.

3.1.1 External Tools Used for Implementation

The development process included the use of the following external tools and libraries:

- **Scala [69]:** the Scala programming language has been used in version 2.9.1. Most of the source code of the application is written in this language.
- **Java:** the Java programming language in version 1.6 has been used as well but just by referencing generated Java code in Scala.
- **ScalaTest [85]:** ScalaTest is a Scala library which enables the programmer to write unit tests. Version 1.6.1 has been used in this project.
- **Simple Build Tool [49] (SBT):** SBT is a build tool for Java and Scala projects (both at a time). Version 0.11.3 has been used to run tests, build the source code and generate the ScalaDoc (source code documentation for Scala).
- **ANTLR Works [74]:** this tool is an IDE for the ANTLR parser generator. It has not been used to actually generate the parser but to check if the parsed grammar fulfills the property of being an LL(1) grammar [1].
- **JNAerator [31]:** the Java Native Access (JNA) is a way to to call native code from the Java programming language. This library is part of the Java Development Kit (SDK) since version 1.4. JNAerator is a tool to generate JNA code from a C/C++ header file. This has been used to call the APIs (written in the C programming language) of the solvers which had no Java bindings available at the time of implementation.
- **Eclipse & Scala Plugin [43]:** Eclipse is a well-known IDE widely used for Java development. The Scala plugin enables approximately the same support for Scala projects as it already exists for Java projects.

3.1.2 Implementation Language: Scala

Scala in version 2.9.1 has been chosen as implementation language. It is a Java Virtual Machine (JVM) based language[2]). As it is JVM-based, any code written in Java is accessible natively. It is a multi-paradigm programming language merging object-oriented and functional style. In the reference book *Programming in Scala* [70], the authors Odersky et al. discuss the advantages of this language closely. Further books on Scala are listed on the Scala language page [68].

[1] A grammar being left to right and leftmost derivation with one look ahead token (LL(1)) avoids back-tracking of the parser (cmp. Section 4.1).

[2] More information about Scala can be found on its official site.

Object-oriented & Functional

On the one hand, Scala is a purely object-oriented programming language. Its main principle is that *every value is an object*. This is not the case in Java as there exist native types like *integer* and *null pointer*. There are also boxed versions of the native types (Integer instead of integer), but just the fact that there exists something not in accordance to the principle crosses the line of pure object-orientation. On the other hand, Scala is a functional programming language. This enforces two main principles: at first, *functions are first-class values*. They can be passed like values. Combined with the principle of pure object-orientation, this leads to the statement that *every function is an object*. The second principle states that *operations of a program map input values to output values*. It is important to notice that this mapping should not produce any side effects. This inherits that no changeable inner state of an object (which includes functions as previously stated) exists. Pure functional programming would enforce this, Scala nevertheless allows imperative-style programming (e.g. with side-effects), but encourages the programmer to program as functional as possible. For instance, there are two types of collections in the Scala library: mutable and immutable ones. Scala automatically imports the immutable versions. Hence, when using a *List* object, it will be used as an immutable List as long as the mutable implementation is not referenced explicitly.

Compatible & Concise

One very big advantage of Scala is its compatibility. As previously stated, it compiles to JVM byte code. This fact inherits that Java code can be called from Scala code and vice-versa without any additional code (apart from the standard Scala API). So in general the run-time performance should be very similar. To assure full interoperability, Scala types are using the standard Java types and wrap them with some higher level functionalities. The type conversion from a String object to an Integer object should show how this works exemplarily.

```java
Integer.parseInt("42");
```

Listing 3.1: Converting a String to an integer in Java

Listing 3.1 shows a Java code snippet: String *42* should be converted to the integer *42*. In Java, a static class called *Integer* has to be called to achieve this. Would it not be more comfortable if the conversion function exists within the *String* type?

```scala
"42".toInt
```

Listing 3.2: Converting a String to an integer in Scala

Usually the Scala compiler would work with the Java *String* implementation. But then it would not find the function *toInt* like it is shown in Listing 3.2. Scala will find an implicit conversion from a *Java.util.String* instance to an enriched String type which includes this method. This Scala compiler feature will be discussed in a more detailed fashion in Section 4.2.1.

```java
class SimpleClass {
  public final int intMember;
  public final String strMember;

  public SimpleClass(int intMember, String strMember) {
    this.intMember = intMember;
    this.strMember = strMember;
  }
}
```

Listing 3.3: Class definition in Java with two immutable members

Another benefit of the language is that it tries to avoid unnecessary code. Listing 3.3 shows a class definition in Java. The *SimpleClass* holds two kinds of values: a String and an integer. The Java code needs eight lines of code to declare this type. In Scala this declaration is done in one single line (cmp. Listing 3.4). Both source codes will be translated to the same JVM byte code.

```scala
class SimpleClass(intMember: Int, strMember: String)
```

Listing 3.4: Class definition in Scala with two members

3.2 Process Description

Figure 3.1 illustrates how the application steps through the process of translating the model-based mutation testing problem and outputs the solution. *Action Systems* are the input for the application:

- one specification
- one or multiple mutants of the specification

These Action Systems are notated in a *Prolog* source code file. Prolog is a logical programming language. A close look at the Action System description in Prolog is done in Section 4.1. This input type was chosen because it has been used by *Ulysses*[3]. The process starts by parsing the input files. This is achieved by a special parsing technique called *combinator parsing*. This topic will be discussed in Section 4.2.1 closely. Internally, the output of the parsing will be converted to an abstract syntax tree (AST). This and other used data structures will be explained in Section 3.3 thoroughly. Furthermore, this AST will be translated to SMT-LIB (cmp. Section 2.2.2). For refinement checking, the non-refinement formula is constructed (cmp. Section 2.1.2 and 5.1.1). As non-refinement checking for Action Systems includes the computation of reachability of the found (possible unsafe) state, the SMT solver has to be called at least a second time. The exact number of calls to the solver depends on the used strategy (cmp. Chapter 5).

3.3 Packages

The following section discusses the modules the *as2smt* application consists of.

3.3.1 Execute

The *Execute* module contains all classes and objects which are used for the execution of the application:

[3]Ulysses is an enumerative test case generator. It has been developed at the Institute for Software Technology (IST) at Graz University of Technology (cmp. [3], [4], [7]).

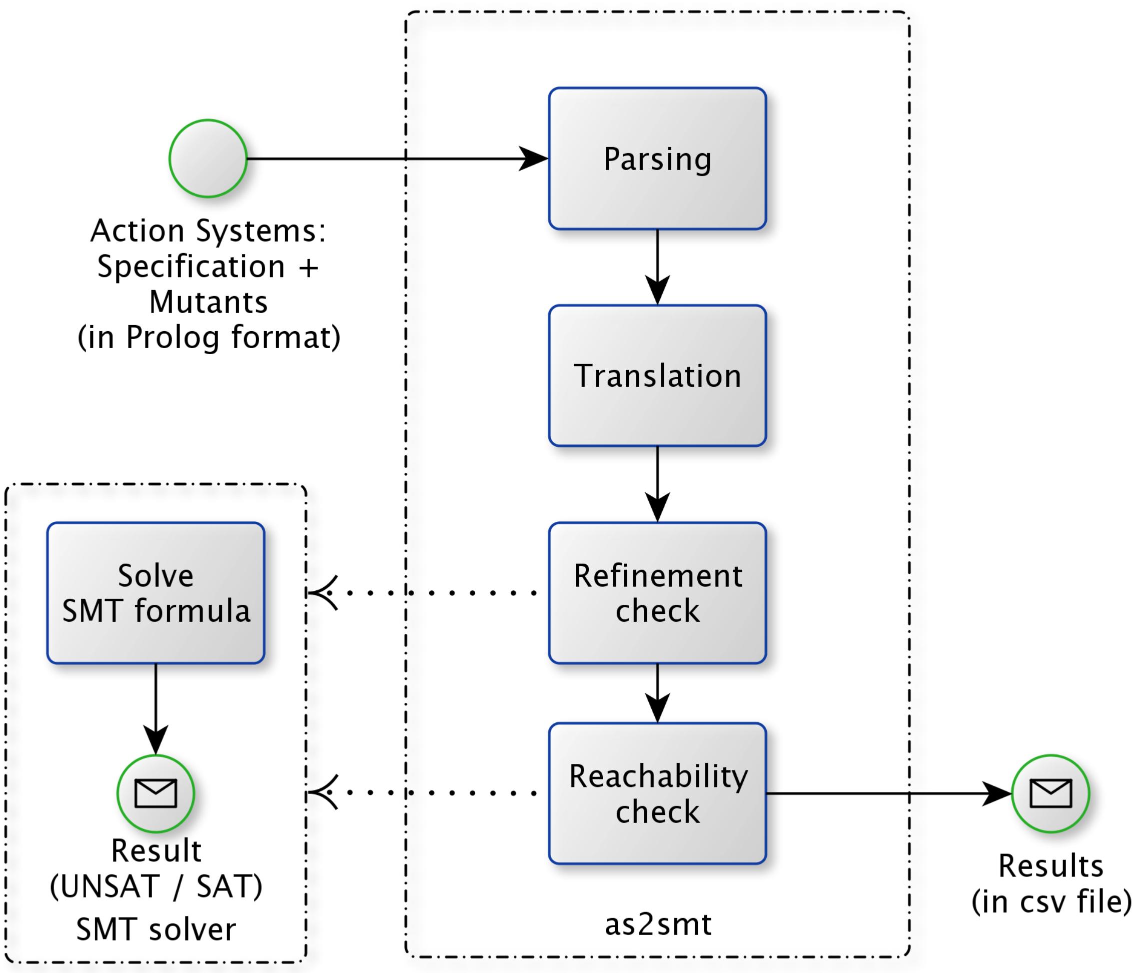


Figure 3.1: Simple process overview of the as2smt application

- **API:** the static *API* object enlists all algorithms to solve the problem of model-based mutation testing. During implementation various improvements have been done on the solution process. These solutions are reflected by the different methods defined in the *API* object.
- **Configuration:** the *Configuration* class holds different values which are used throughout all modules. The respective object holds the latest values and makes them accessible globally.
- **ModelBasedChecker:** the static *ModelBasedChecker* object is the entry point for starting the application. By simply executing the *runModelBasedChecker* method, the as2smt application is started.
- **Run:** the static *Run* object is a simple command line interface for starting some predefined configurations. In Section 3.4, its usage is outlined.

3.3.2 Parser

In the *Parser* module, all implemented parsers can be found. The parsers use combinatory parsing. Figure 3.2 shows the classes of this package in an UML class diagram. The implemented classes inherit from *JavaTokenParser* - a Scala API class which represents combinator parsers (with some previously defined combinators). More on this technique can be found in Section 4.2.1.

- **ActionSystemParser** The *ActionSystemParser* object defines the parser combinators to process a file containing an Action System in Prolog (cmp. section 4.1).

- **ResultParser** The *ResultParser* trait[4] specifies the two functions which every result parser has to contain. The *parseResult* function is already implemented as it is not result parser-specific. The *resultParser* function is the root parser combinator which is called by the implemented *parseResult* function. The return value is of type *ActionSystemModel* which is described closely in the *Translation* subsection. The following classes implement the *ResultParser* trait:
  - **CVC3ResultParser** parses the output of the CVC3 SMT solver.
  - **MathSATResultParser** parses the output of the MathSAT SMT solver.
  - **Z3ResultParser** parses the output of the Z3 SMT solver when called via C API.
  - **Z3ModelParser** parses the output of the Z3 SMT solver when called via command line.

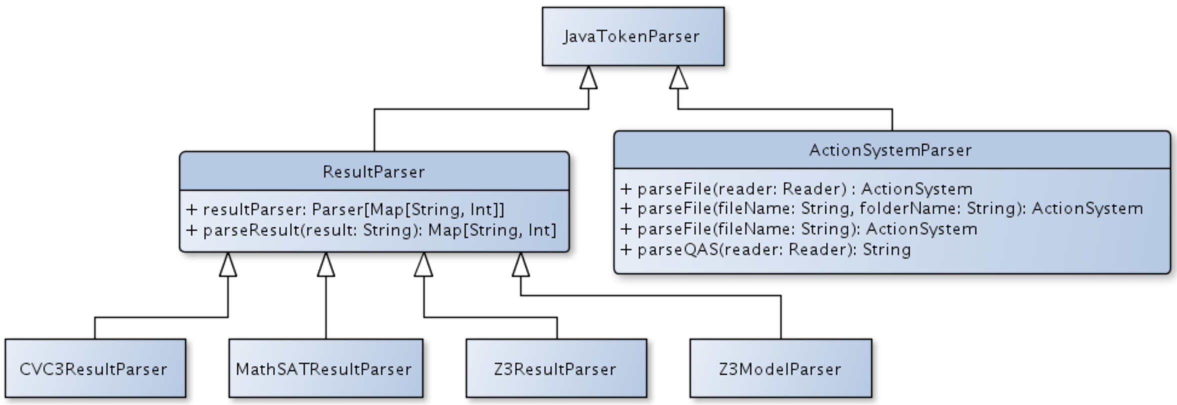

JavaTokenParser
ResultParser
+ resultParser: Parser[Map[String, Int]]
+ parseResult(result: String): Map[String, Int]
ActionSystemParser
+ parseFile(reader: Reader) : ActionSystem
+ parseFile(fileName: String, folderName: String): ActionSystem
+ parseFile(fileName: String): ActionSystem
+ parseQAS(reader: Reader): String
CVC3ResultParser
MathSATResultParser
Z3ResultParser
Z3ModelParser

Figure 3.2: Package *Parser* class diagram: combinator parsers extend *JavaTokenParser* (Scala API class)

3.3.3 Translation

The *Translation* package consists mainly of class declarations to define the abstract syntax tree (sub-classes of *Translation*, cmp. Figure 3.3) and a translator (*ActionSystemTranslator*). In case of *ANTLR*, the visitor pattern [73] is used when walking through an abstract syntax tree. Utilising this pattern in Scala is also undoubtedly possible. But this kind of functionality is achieved more concisely with a feature this language offers: pattern matching.

Abstract syntax tree

The *parseFile* function defined in the *ActionSystemParser* (cmp. Figure 3.2) returns an instance of an **ActionSystem**. This class and all other AST nodes are defined in the *Translation* file. All nodes extend from *Translation*. Nodes for arithmetic and Boolean expressions, statements, variable types and do-od block expressions (*Callable*) are grouped in an additional hierarchy level. The nodes *CallPrioComp* and *StmtPrioComp* are not listed because their translation has not been implemented yet.

In Scala, it is possible to define multiple classes in one file (which is not possible in Java). All AST nodes are sub-classes of the abstract *Translation* class. The *sealed* keyword assures that the class will not get extended by other classes which have not been declared in the same file. All of these classes are marked as case classes to enable class/object matching on them. The defined AST nodes are shown in Figure 3.3. Listing 3.5 shows a small subset of the class declarations in the *Translation.scala* file.

[4]A trait is a stackable, rich interface in Scala. For a detailed description, refer to [70, p.217 ff].

```

sealed abstract class Translation

sealed abstract trait ArithmeticExpression extends Translation

case class AEInteger(value: Int) extends ArithmeticExpression

case class AEVariable(value: String) extends ArithmeticExpression

case class AEAddition(left: Translation, right: Translation) extends
    ArithmeticExpression

case class AESubstraction(left: Translation, right: Translation) extends
    ArithmeticExpression
```

Listing 3.5: Definition of AST nodes (subset)

Translation to target language

When using the mentioned *Visitor* pattern solution [45], one would define a *translate* method on every AST node. The much cleaner and easier to refactor way is to implement it like it has been done in the *ActionSystemTranslator* class. Once again, the subset is sufficient to show the idea behind it. This is illustrated in Listing 3.6. The *translate* function takes as input a *Translation* object which origins from the parser and an integer indicating the current depth in the AST (only necessary pretty printing). The function simply applies the Scala *match* operator on the *Translation* object. Depending on the real type of this object, the code under the *case* object definition gets executed.

```
def translate(t: Translation = as, depth: Int = 0): String = t match {
  case AEInteger(value) =>
    ... //translation of an AEInteger object
  case AEVariable(name) =>
    ... //translation of an AEVariable object
  case AEAddition(left, right) =>
    ... //translation of an AEAddition object
  case AESubstraction(left, right) =>
    ... //translation of an AESubstraction object
}
```

Listing 3.6: Function to match AST node type (subset)

The marking as case classes enables the pattern matching on it (*case* keyword plus class name). The identifiers in parentheses after the class name indicate the member fields of the object (cmp. Section 3.1.2). As the *sealed* keyword assures no further extensions in other files, the compiler will notice if the matching is exhaustive or not e.g. if a case was forgotten [70, p.286ff].

Supplemental classes

Additionally to *ActionSystemTranslator* and the AST node definitions in *Translation* the following object definitions (classes) exist:

- **ActionSystemNormalizer:** this class is responsible to transform an abstract syntax tree into a certain normal form (cmp. Section 4.4.5).

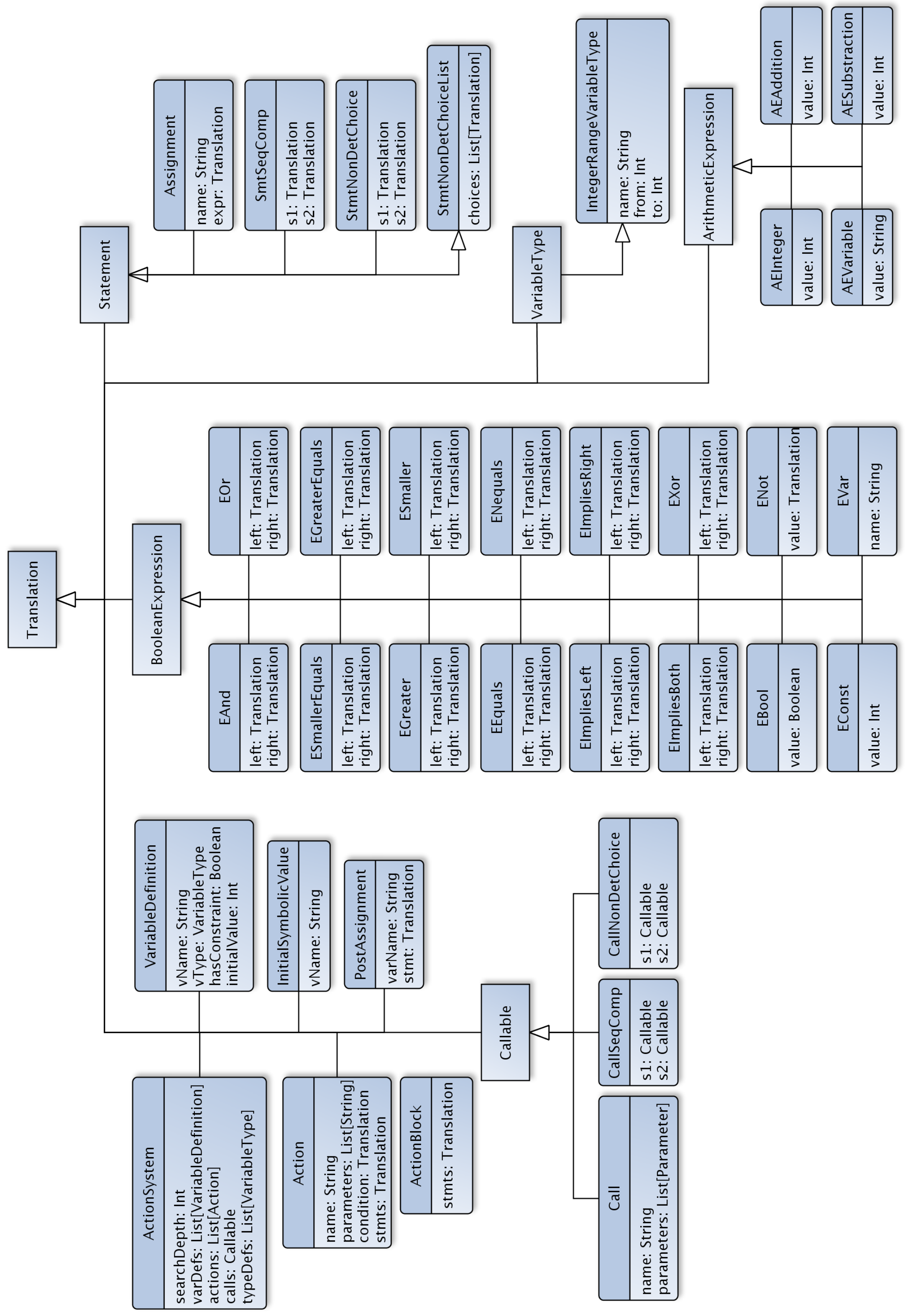


Figure 3.3: Class diagram of the AST nodes

- **SymbolTable & SymbolTableEntry:** during the translation process, a data storage is needed to translate the AST correctly. An instance of the *SymbolTable* class keeps track of all values necessary to compute the translation. Among those variables are instances of the *SymbolTableEntry*. One instance holds the current symbolic value of a SMT-LIB variable. This is necessary for applying the one-point rule (cmp. Section 4.4.1).

- **TranslationType:** the *TranslationType* is an enumeration (which is achieved in Scala by extending from the *Enumeration* class). It is used as a member to be set for an *ActionSystemTranslator* instance. For testing, the translator *SimpleTranslation* is used, which will produce a positive translation including the variable declarations. When choosing *NegatedSpecification*, the result will contain the SMT-LIB code in negated form (by adding a negation in front of the root node), the variable declarations will stay the same. By setting it to *Mutant*, the translation will be exactly like when using *SimpleTranslation* but without any variable declarations and leading negation.
- **Formatter sub-package:** the output of the translator will be in form of a string. By setting the TranslationFormatter, the shape of this string can be affected.
  - **TranslationFormatter:** this trait defines the functions which have to be implemented to construct a usable formatter. Both, the *SimpleTranslationFormatter* and the *IndentedTranslationFormatter* implement this.
  - **SimpleTranslationFormatter:** the *SimpleTranslationFormatter* results in a compact string output.
  - **IndentedTranslationFormatter:** the *IndentedTranslationFormatter* adds indentations to make the SMT-LIB code better readable for humans.

The ActionSystemTranslator and its dependencies are illustrated in Figure 3.4. A UML class diagram representation of the *formatter* sub-package is shown Figure 3.5.

3.3.4 Solver

Figure 3.1 shows the interaction with a back-end: an SMT solver. SMT-LIB (version 2) language has been implemented as target language of the translator, which is a standardised input language for SMT solvers [23] (cmp. Section 2.2.2). In general, SMT solvers support just a subset of the defined theories. Initially, integers, real numbers, arrays and quantification were required by the tool. Due to the elimination of quantifiers (cmp. Section 4.4) and the drop of real number and array support, just integers have been needed by the as2smt tool. At the time of the implementation, the following solvers have been considered as back ends for the implemented application:

- **Z3** [76] developed by Microsoft Research (Redmond, United States)
- **MathSAT** [47] developed at University of Trentino, Italy
- **CVC3** [21] developed at New York University, US
- **SMTInterpol** [32] developed at University of Freiburg, Germany

These solvers are accessed by the *as2smt* application in different ways. At an early development stage, this was achieved by a command line call for all solvers. During the first experiments this was one major performance bottleneck. Therefore a solution for a native access has been implemented: *Z3* - the most promising solver when it comes to performance [79] - had no official Scala/Java interface available at the time of implementation. The external library $Scala^{Z3}$ [81] (a *Google Code* project of Phillip Suter, a member of Martin Odersky's team) was tested, but at the time of implementation the latest version of *Z3* was not supported. Hence, the usage of the *Java Native Interface* (JNI) and *Java Native Access* was evaluated. Through these Java frameworks (both part of the standard Java API) it is possible to call native interfaces. It would have been quite an effort to implement the glue code in one of the frameworks. The tool *JNAerator* [31] is a generator of JNA bindings based on C-header files[5]. Through this method *Z3*, *MathSAT* and *CVC3* are accessed in a native way with good performance. *SMTInterpol* is written itself in Java so the matter of accessing it was simple.

[5]Thanks to the author *Oliver Chafik* and also *Angelo Gargantini* for sharing his experience with the tool.

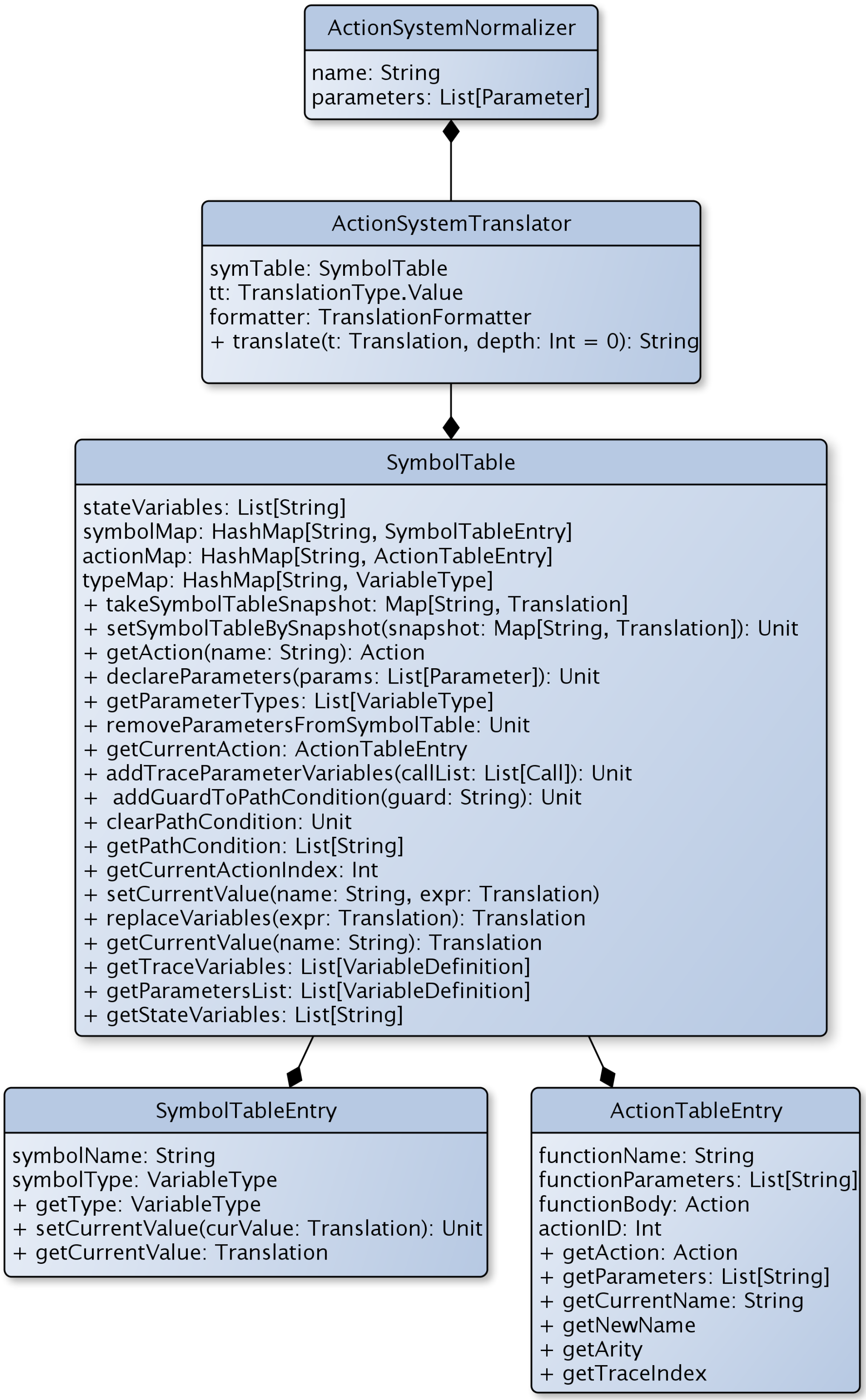

ActionSystemNormalizer
name: String
parameters: List[Parameter]
ActionSystemTranslator
symTable: SymbolTable
tt: TranslationType.Value
formatter: TranslationFormatter
+ translate(t: Translation, depth: Int = 0): String
SymbolTable
stateVariables: List[String]
symbolMap: HashMap[String, SymbolTableEntry]
actionMap: HashMap[String, ActionTableEntry]
typeMap: HashMap[String, VariableType]
+ takeSymbolTableSnapshot: Map[String, Translation]
+ setSymbolTableBySnapshot(snapshot: Map[String, Translation]): Unit
+ getAction(name: String): Action
+ declareParameters(params: List[Parameter]): Unit
+ getParameterTypes: List[VariableType]
+ removeParametersFromSymbolTable: Unit
+ getCurrentAction: ActionTableEntry
+ addTraceParameterVariables(callList: List[Call]): Unit
+ addGuardToPathCondition(guard: String): Unit
+ clearPathCondition: Unit
+ getPathCondition: List[String]
+ getCurrentActionIndex: Int
+ setCurrentValue(name: String, expr: Translation)
+ replaceVariables(expr: Translation): Translation
+ getCurrentValue(name: String): Translation
+ getTraceVariables: List[VariableDefinition]
+ getParametersList: List[VariableDefinition]
+ getStateVariables: List[String]
SymbolTableEntry
symbolName: String
symbolType: VariableType
+ getType: VariableType
+ setCurrentValue(curValue: Translation): Unit
+ getCurrentValue: Translation
ActionTableEntry
functionName: String
functionParameters: List[String]
functionBody: Action
actionID: Int
+ getAction: Action
+ getParameters: List[String]
+ getCurrentName: String
+ getNewName
+ getArity
+ getTraceIndex

Figure 3.4: Class diagram of *ActionSystemTranslator* and its dependencies

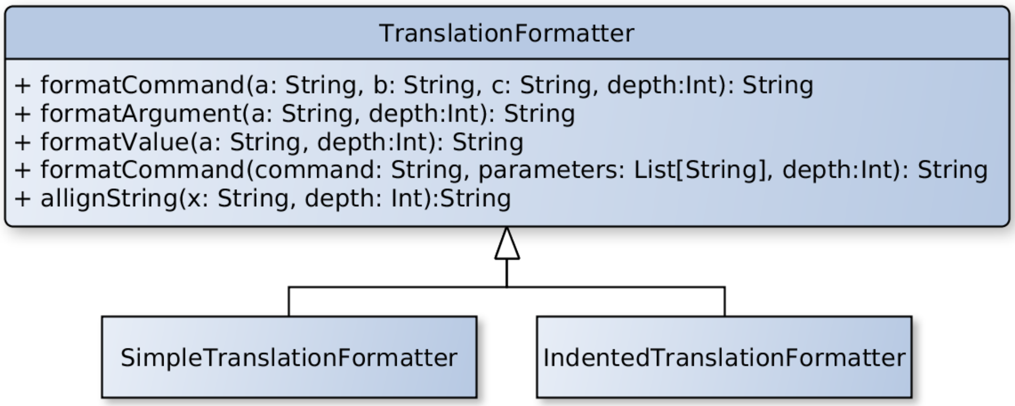

TranslationFormatter
+ formatCommand(a: String, b: String, c: String, depth:Int): String
+ formatArgument(a: String, depth:Int): String
+ formatValue(a: String, depth:Int): String
+ formatCommand(command: String, parameters: List[String], depth:Int): String
+ allignString(x: String, depth: Int):String
SimpleTranslationFormatter
IndentedTranslationFormatter

Figure 3.5: Class diagram of package *Formatter*

The package *Solver* defines the following elements:

- **SMTSolver** is an enumeration of all currently supported SMT solvers.
- **SMTSolverAPI** defines functions which the as2smt application needs when interfacing to an SMT solver.
- **SMTIncrementalSolver** defines functions to enable incremental solving support for a specific solver.
- **SMTSolverConfiguration** specifies the configuration parameters which can be set for a specific solver.

In the sub-packages *cvc3*, *mathsat*, *smtinterpol* and *z3* the implementations of the previously listed interfaces can be found. Figure 3.6 shows a class diagram with the needed interfaces to access an SMT solver in the as2smt application.

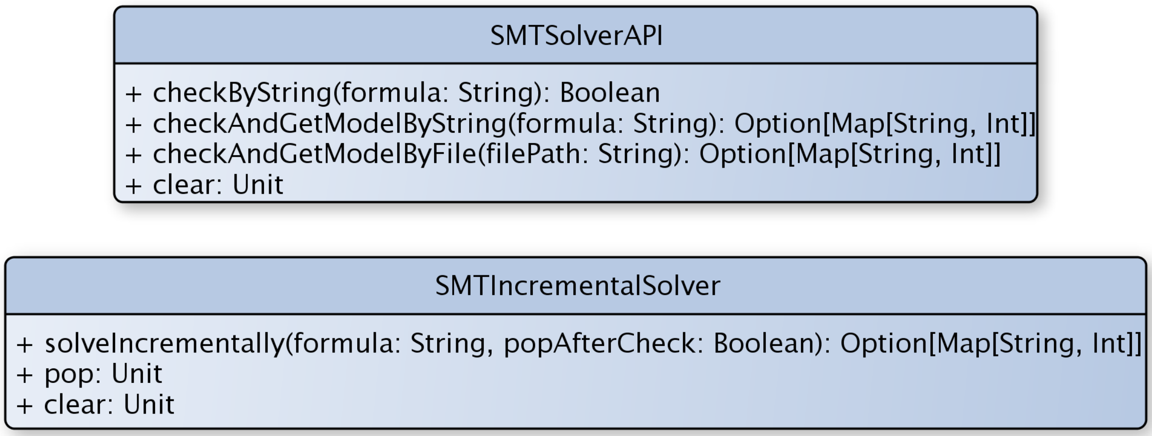

SMTSolverAPI
+ checkByString(formula: String): Boolean
+ checkAndGetModelByString(formula: String): Option[Map[String, Int]]
+ checkAndGetModelByFile(filePath: String): Option[Map[String, Int]]
+ clear: Unit
SMTIncrementalSolver
+ solveIncrementally(formula: String, popAfterCheck: Boolean): Option[Map[String, Int]]
+ pop: Unit
+ clear: Unit

Figure 3.6: Class diagram of interfaces/traits for accessing SMT solvers

3.3.5 Reachability

The *Reachability* package consists of classes which use the solver at a higher level. In Figure 3.1 the actions *Refinement* and *Reachability* represent these steps. These are the building blocks for the algorithms defined in the *API* object in the *Execute* package. The elements are listed below and in Figure 3.7.

- **ActionSystemModel:** this data structure represents the output of a solver. Usually this is a number of assignments to variables. In fact there are three kinds of variables: original (pre)state variables, the post state variables and the trace variables. An *ActionSystemModel* object separates these three types cleanly.
- **MutatedActionFinder:** this is a deprecated way for determining which action has been mutated in comparison with a specification. It uses the solver heavily by translating every action of the mutant one by one.
- **SyntacticEqualityChecker:** a more efficient way to determine the mutated action is the syntactic equality checker. By comparing the ASTs of the specification and the mutant syntactically this is achieved in the *SyntacticEqualityChecker*.
- **PostStateFinder:** this class implements the logic to get all direct post states from one initial state. It uses an instance of an *SMTIncrementalSolver*. At creation time it pushes the specification on the assertion stack.
- **ReachabilityChecker:** the *ReachabilityChecker* uses a *PostStateFinder* instance to explore the next possible states. With an extra *SMTIncrementalSolver* instance it checks if one of the found post states is an unsafe state. If not it continues with one of the found post states as an initial state. The used search algorithm in this state tree is breadth first. This terminates either when an unsafe state is found or the maximum search level is reached.
- **ReachabilityAnalyzer & ReachableStatesExplorer:** the *ReachabilityAnalyzer* and the *ReachableStatesExplorer* split this task into two separate ones. The set of reachable states is generated by the *ReachableStatesExplorer* which serves the *ReachabilityAnalyzer* as an input to find an unsafe state.

3.3.6 Util

The *Util* package contains two objects for creating an output in a file representation:

- **StopWatch:** the *StopWatch* is used mainly in the *API* object (*Execute* package) to determine timing behaviour.
- **CsvFileWriter:** the results of the as2smt application are written into a comma-separated values (CSV) file.

3.4 Usage

The *Simple Build Tool* (SBT) acquires all external dependencies needed to build and run the *as2smt* application. The only requirement needed is that a JVM version ≥ 1.6 is provided.

3.4.1 SBT: Configuration & Usage

The folder *as2smt/code* contains a script *sbt* to launch the *SimpleBuildTool* [49]. This tool offers an interactive shell to build, test, generate documentation for and deploy pure Scala, Java or mixed projects. It is configurable via a *build.sbt* file, located in the same directory where it has been started on the command line. The SBT configuration of this project is shown in Listing 3.7. There the name, the version of the project and the used Scala version is set. Additionally, a Scala compiler option is enabled (*-deprecation* triggers compiler warnings if deprecated classes are referenced) and a library dependency is added. In general, every parameter-value pair has to be separated by an empty line. For a more complex configuration, a Scala file can be used instead (cmp. [49], *Documentation* section).

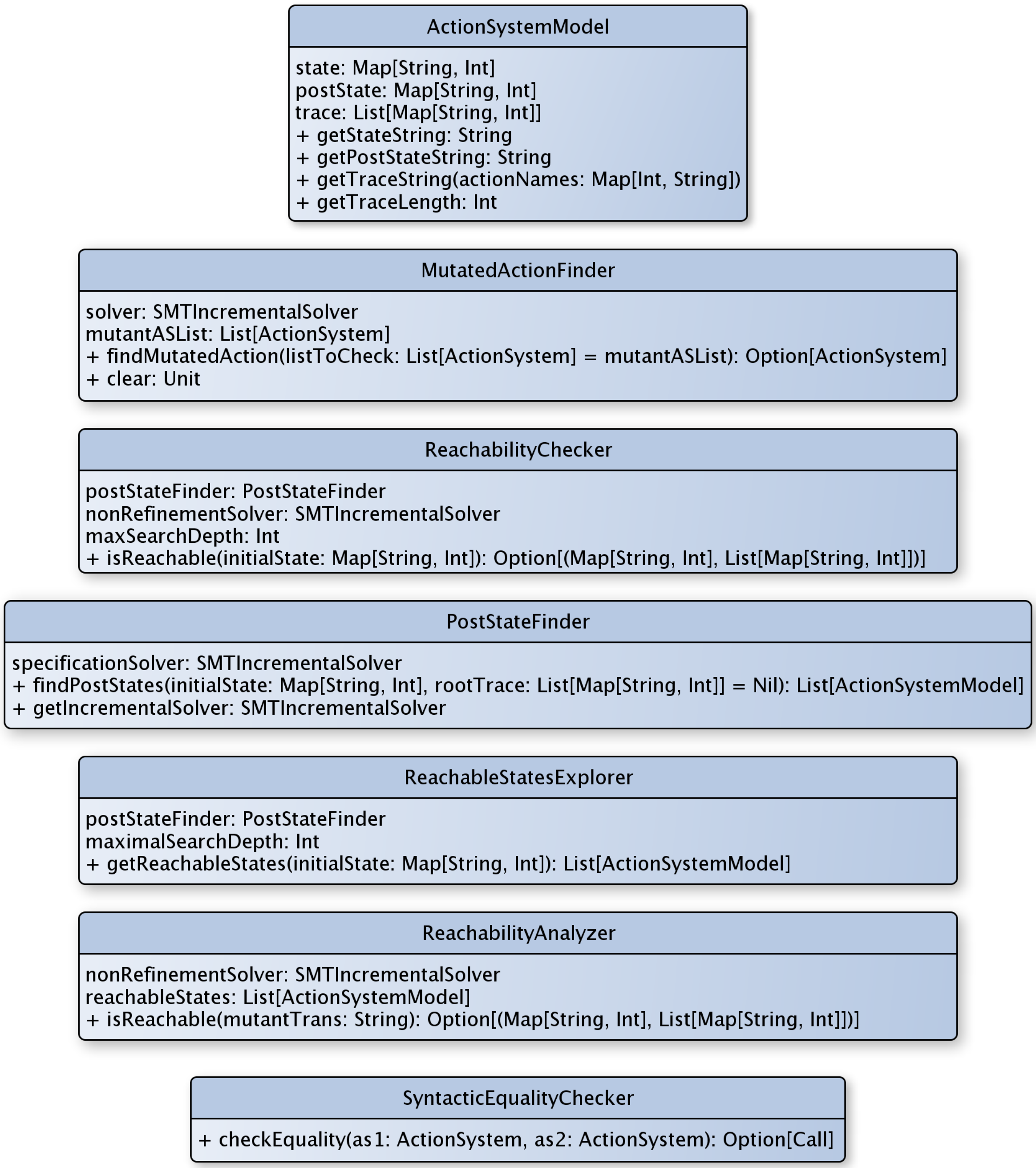

ActionSystemModel
state: Map[String, Int]
postState: Map[String, Int]
trace: List[Map[String, Int]]
+ getStateString: String
+ getPostStateString: String
+ getTraceString(actionNames: Map[Int, String])
+ getTraceLength: Int
MutatedActionFinder
solver: SMTIncrementalSolver
mutantASList: List[ActionSystem]
+ findMutatedAction(listToCheck: List[ActionSystem] = mutantASList): Option[ActionSystem]
+ clear: Unit
ReachabilityChecker
postStateFinder: PostStateFinder
nonRefinementSolver: SMTIncrementalSolver
maxSearchDepth: Int
+ isReachable(initialState: Map[String, Int]): Option[(Map[String, Int], List[Map[String, Int]])]
PostStateFinder
specificationSolver: SMTIncrementalSolver
+ findPostStates(initialState: Map[String, Int], rootTrace: List[Map[String, Int]] = Nil): List[ActionSystemModel]
+ getIncrementalSolver: SMTIncrementalSolver
ReachableStatesExplorer
postStateFinder: PostStateFinder
maximalSearchDepth: Int
+ getReachableStates(initialState: Map[String, Int]): List[ActionSystemModel]
ReachabilityAnalyzer
nonRefinementSolver: SMTIncrementalSolver
reachableStates: List[ActionSystemModel]
+ isReachable(mutantTrans: String): Option[(Map[String, Int], List[Map[String, Int]])]
SyntacticEqualityChecker
+ checkEquality(as1: ActionSystem, as2: ActionSystem): Option[Call]

Figure 3.7: Class diagram of package *Reachability*

```
name := "as2smt-sbt"

version := "1.0"

scalaVersion := "2.9.1"

scalacOptions += "-deprecation"

libraryDependencies += "org.scalatest" %% "scalatest" % "1.8" % "test"
```

Listing 3.7: Configuration of SBT (build.sbt)

When starting the *sbt* script, all dependencies of the project will be downloaded automatically from an external repository. This includes the set Scala version and the external library dependencies. Afterwards, the interactive shell will appear (cmp. Figure 3.8)

```
[info] Loading project definition from /home/matthias/private/thesis/
    as2smt/code/project
[info] Set current project to as2smt-sbt (in build file:/home/matthias/
    private/thesis/as2smt/code/)
>
```

Listing 3.8: SBT shell

The SBT shell is capable of command completion just like in the bash on Unix-based systems by using the *Tab* button. The following commands have been used during development process:

- **compile:** This command will compile the application to Java Byte code.
- **doc:** The *doc* command triggers the generation of the *ScalaDoc*.
- **test & test-only:** To run all tests suites *test* is used. Single test suites can be executed by *test-only*.
- **update-sbt-classifiers:** This will update the SBT installation.
- **eclipse:** In SBT additional plugins can be added. This is set in a separate file *plugins.sbt* located in the *project* folder (cmp. Listing 3.9). To open the project in Eclipse, a *.project* can be generated via this command.
- **run:** The *run* command starts the *main* function in the application.

```
resolvers += Classpaths.typesafeResolver

addSbtPlugin("com.typesafe.sbteclipse" % "sbteclipse-plugin" % "2.1.1")
```

Listing 3.9: SBT plugins: adding plugins to SBT

3.4.2 Running the Application

In general, the application requires three parameters:

- SMT solver
- API method
- Action System

The implemented *main* function (*execute.Run.main*) is executed in SBT by entering *run*. The needed parameters can be set

- directly with the run command, e.g. *run 1 1 1*.
- interactively, setting it one by one.

The values for the parameters are numerical. The mappings of the numbers to their meaning are shown in the interactive mode. After typing *run*, the output as presented in Listing 3.10 appears. The possible values are:

1. MathSAT

2. SMTInterpol
3. Z3

By entering one of the three numbers, the selected solver will be used as a back-end of the *as2smt* tool for this run. If an input is chosen which is not listed, a default value is taken instead. Listing 3.10 illustrates this behaviour.

```
> run
[info] Running execute.Run
please select a solver:
1 − MathSAT
2 − SMTInterpol
3 − Z3
> 5
no solver selected. choosing Z3 by default.
```

Listing 3.10: Output of run command: choosing the solver

The next parameter to be set is the API method with the following possible values:

1. Syntactic check, incremental solving, reachability computed per mutant
2. Syntactic check, incremental solving, precomputation of reachable states
3. Syntactic check, incremental solving, precomputation and single assertion of reachable states

The default value for this parameter is set to API method #2. Listing 3.11 represents the output for this parameter selection.

```
please select the API method to be used:
1 − Syntactic check, incremental solving, reachability per mutant
2 − Syntactic check, incremental solving, precomputation of reachable
    states
3 − Syntactic check, incremental solving, precomputation and single
    assertion of reachable states
> 0
no method selected. choosing 2 by default.
```

Listing 3.11: Output of run command: choosing the solving method

The last parameter is the Action System configuration to be tested. This contains a specification file path and a directory with all mutants to be tested. The *as2smt* application comes with eight different configurations based on four action systems (cmp. Section 6.2):

1. Original car alarm system (CAS)
2. Original CAS with parameter values multiplied by 10
3. Original CAS with parameter values multiplied by 100
4. Original CAS with parameter values multiplied by 1000
5. Car alarm system with Boolean PIN
6. Car alarm system with numerical PIN
7. Triangle
8. Triangle with parameter values multiplied by 10
9. Custom Action System (selection of name and paths)

The output is shown in Listing 3.12. For this parameter the Original CAS is selected by default.

```
please select the Action System to be tested:
1 - Original CAS
2 - Original CAS (10)
3 - Original CAS (100)
4 - Original CAS (1000)
5 - Boolean-PIN CAS
6 - PIN CAS
7 - Triangle
8 - Triangle (10)
9 - [INPUT]
> 9
name: test
specification file path: ../../examples/car_alarm_system/simplified_non
    -det_2/AlarmSystem_5Custom_noskip_sound_off_twice.pl
mutant folder path: ../../examples/car_alarm_system/simplified_non-
    det_2/mutants
```

Listing 3.12: Output of run command: choosing the Action System

In total, there are 72 existing run configurations. By choosing #9, a custom Action System can be set. If this should be permanently part of the application, this can be embedded easily in the source code by adding two lines of code:

- The selection function *getActionSystem* (takes an integer as a parameter) has to be adapted by one additional case (cmp. Listing 3.13).
- The output function *getActionSystem* (takes three Strings as parameters) has to be extended by the new option. This way, the interactive mode works also properly.

```
case 10 => ("../example/path/to/spec_file.pl", "../example/path/to/
    mutantFolder/", "exampleName")
```

Listing 3.13: Adding an Action System configuration (part 1)

```
10 - exampleName
```

Listing 3.14: Adding an Action System configuration (part 2)

The results in CSV (comma separated values) format of every run will be saved in the *results* directory.

3.5 Limitations of the Implemented Tool

In this section, the limitations of the implemented tool are listed.

- **First order mutants:** the *as2smt* application is able to deal with a first order mutations only. Multiple mutations are not detectable by the tool. If there would be a mutation in more than one action, just the first action found would be analysed. In Section 1.5, the coupling effect has been discussed. In regards to it, first order mutants are sufficient to also uncover higher order mutations.

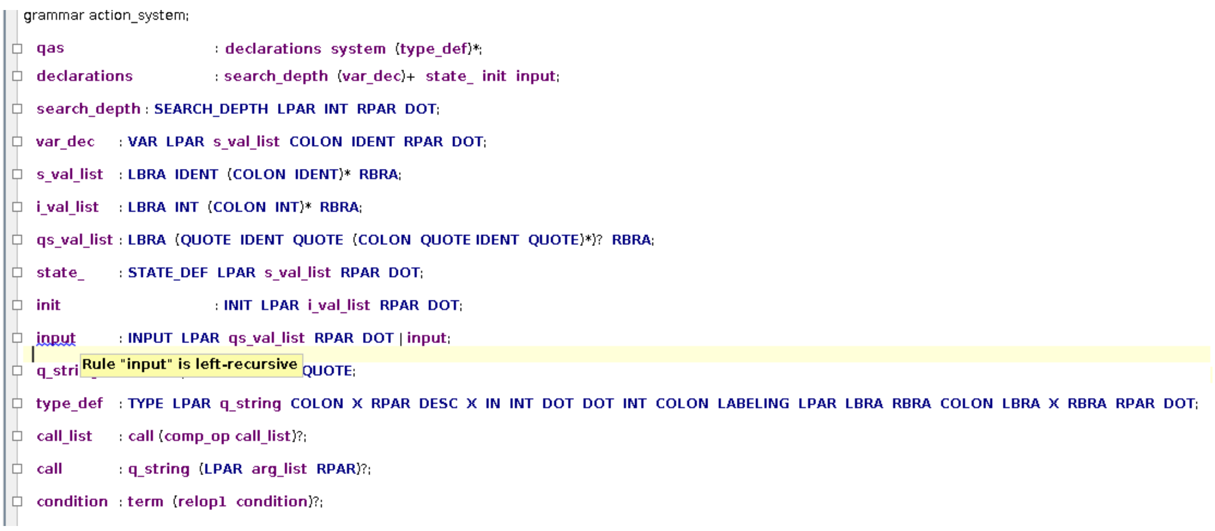
- **Mutants with the same variable definitions as the original:** the mutant is translated but without any variable definitions. This is important to note because this marks another limitation of the tool. There must not exist any mutations concerning the state variables. Neither changes of their number, nor their names are possible to detect with the presented tool.
- **Mutants with fixed action signature:** the tool is not able to handle additional parameters for actions in mutants as this would imply additional variable declarations. All parameters are declared as read only variables in the beginning at the time the original is translated.
- **Mutants with same number of calls in the do-od block:** at the current state, *as2smt* is not able to handle different numbers of calls to actions in mutants. It will terminate with an error.

4 Translating Action Systems to SMT-LIB

This chapter presents how the translation of Action Systems to SMT-LIB input language has been achieved. For brevity, the language will just denoted by *SMT-LIB* or *SMT-LIB code*. It involves the steps *Parsing* and *Translation* in Figure 3.1. At first, the source language is introduced. The subsequent section deals with the methods of lexical and syntactic analysis of the source language. Furthermore, the translation of each element of the Action System language definition to SMT-LIB code is discussed. Finally, the last section introduces a different approach of the translation which has been shown to be more efficient (cmp. Chapter 6).

4.1 Input Language: Action Systems in Prolog

In general, *Action System* is a formalism (cmp. 2.1.1), not a concrete programming language. A tool called Ulysses (cmp. [3, 4, 7]) developed at the Institute for Software Technology (IST) at Graz University of Technology has been using Action Systems (AS for short, cmp. Section 2.1.1). Aichernig et al. show the Action System syntax for the Prolog programming language by an example [8]. The syntax was formalised into an Extended Backus-Naur form (EBNF) representation in the (not publicly available) internal Ulysses documentation. Taking this existing EBNF as a source, a subset (with the extension for Qualtiative Action System [5]) is transformed into a left to right and leftmost derivation with one look ahead token grammar (LL(1) for short). The parsing technique used for the *as2smt* translation supports grammars in other forms than LL(k). There is even an implementation supporting left recursions [86]. But backtracking is omitted when transforming the grammar into this kind of context-free grammar. Hence, this represents the more optimal solution with respect to performance. This transformation was achieved by the aid of ANTLRWorks - an ANTLR IDE. Figure 4.1 shows a screenshot of the *ANTLRWorks* application [74]. Its graphical interface enables the user to spot even very opaque indirect recursions. In the shown example, the editor highlights the rule *input* to indicate that the rule is declared in a left-recursive fashion. By applying the suggested transformations of the EBNF, the grammar can be converted to an LL(k) grammar[1]. The resulting grammar can be found in Appendix A. On the basis of this grammar, a compiler has been implemented. It is presented in the further sections of this chapter.



```
grammar action_system;
qas            : declarations system (type_def)*;
declarations   : search_depth (var_dec)+ state_ init input;
search_depth : SEARCH_DEPTH LPAR INT RPAR DOT;
var_dec     : VAR LPAR s_val_list COLON IDENT RPAR DOT;
s_val_list  : LBRA IDENT (COLON IDENT)* RBRA;
i_val_list  : LBRA INT (COLON INT)* RBRA;
qs_val_list : LBRA (QUOTE IDENT QUOTE (COLON QUOTE IDENT QUOTE)*)? RBRA;
state_      : STATE_DEF LPAR s_val_list RPAR DOT;
init           : INIT LPAR i_val_list RPAR DOT;
input       : INPUT LPAR qs_val_list RPAR DOT | input;
q_stri Rule "input" is left-recursive QUOTE;
type_def    : TYPE LPAR q_string COLON X RPAR DESC X IN INT DOT DOT INT COLON LABELING LPAR LBRA RBRA COLON LBRA X RBRA RPAR DOT;
call_list   : call (comp_op call_list)?;
call        : q_string (LPAR arg_list RPAR)?;
condition   : term (relop1 condition)?;
```

Figure 4.1: ANTLRWorks: editor for parser generator ANTLR

[1]For LL(1), the option $k = 1$ has to be set in ANTLRWorks.

4.2 Lexical and Syntactic Analysis

A compiler generation framework like ANTLR [72] could have been used for implementing this translation. But there exists a parsing technique which is inherent to functional programming (cmp. Section 3.1.2): combinatory parsing. The main benefits of the Scala implementation of parser combinators for this project have been

- its usage is clearly better integrated into the implementation language,
- the readability of the code. The lexer part of a parser in Scala resembles very closely to an EBNF grammar,
- the dependency on the pure Scala API without external references,
- the possible gain in performance (optimised code for this purpose), and
- the more intuitive way of writing programming code for the semantic analysis.

In the following subsection, we take a close look how combinatory parsing works in Scala.

4.2.1 Combinatory Parsing

The main idea behind this approach is the usage of higher-order functions which each parses a distinct part of the input. Back in 1989, Frost and Launchburry introduce this way of constructing *a natural language interpreter in a lazy functional language* [44]. In 1992, Hutton presents a library for parser combinators in Haskell [53]. The Scala API includes the used parser combinator framework [70].

Building block: Functions

In combinatory parsing, a parser is nothing else than a function. This function is applied on the whole input and validates that one lexeme exists at its beginning. It outputs a *successful* message and the rest of the text without the validated lexeme if this lexeme has been found. If not, it returns a *failure* message [53]. In the Scala library *scala.util.parsing.combinator*, such a parser is represented by an instance of the *Parser* class. It is simply a function which takes one parameter of type *Input* (which is usually a character stream) and returns a value of type *ParseResult* which is typed itself by another type *T*. In the *as2smt* tool, the extension *JavaTokenParsers* has been used. In difference to its base class, it implements an implicit type conversion for parser combinators for constant strings. Furthermore, the *Input* type is set to *Char* by default. Additionally, there are already defined combinators for parsing identifiers, integers, floats and string literals. To call this combinator function in an object-oriented way (in the multi-paradigm programming language Scala a function is a first-order object), the *apply* method is defined. Listing 4.1 shows the abstract *Parser* class of the Scala API (version 2.9.2). *ParseResult* contains the remaining, not yet parsed part of the input plus one of the messages mentioned above. The Scala API distinguishes between two cases of the failure message. So, the *ParseResult* can be one of the following values:

- **Success** which is equivalent to the mentioned successful message.
- **Failure** which denotes that the function was not able to match the lexeme, but the program may continue by backtracking and look for another possibility.
- **Error** which indicates that the function was not able to match the lexeme and no backtracking is done.

```scala
abstract class Parser[+T] extends (Input => ParseResult[T]) {
  def apply(in: Input): ParseResult[T]
}
```

Listing 4.1: Scala API Parser class

Implicit conversion

To fully understand the syntax, one has to know about the implicit conversion feature of the Scala compiler. It is able to determine the type of any variable including return values. Listing 4.2 shows an example of a usual Scala function with a String as return value. The return value type is given explicitly after the function name (separated by a colon, Line 1). However, the Scala compiler is able to check the return value on its own as the right-hand side (Line 2) of the function definition is of the type String (implicit return type). Therefore, Listing 4.3 also compiles without any warnings or errors.

```scala
1 def conStrFunction:String =
2   "constantString"
```

Listing 4.2: Example for a function returning a String in Scala

```scala
1 def conStrFunction =
2   "constantString"
```

Listing 4.3: Example for a function without explicit return type

In case of an explicit return type, the Scala compiler tries to match the explicit and the implicit types. If they do not match, the compiler looks up all declared implicit conversions. These are marked by the keyword *implicit* in the source code. The used *RegexParsers* trait (an extension of *Parsers*) has such an implicit conversion defined. Listing 4.4 illustrates this conversion function defined in the mentioned object. Because of this feature of the Scala compiler and the declared import, Listing 4.5 compiles without any warning or error: the implicit type is *String*, the explicit one is *Parser[String]*. The compiler finds an implicit type conversion from *String* to *Parser[String]*.

```scala
trait RegexParsers extends Parsers {
    /* [...] */

 implicit def literal(s: String): Parser[String] = new Parser[String] {
    // object creation
   }

  /* [...] */
}
```

Listing 4.4: Excerpt from the RegexParsers class with implicit conversion from String to Parser

```scala
import scala.util.parsing.combinator.JavaTokenParsers

object MyParser {
  def actionStringParser:Parser[String] = "ACTION"
}
```

Listing 4.5: Example for a parser parsing the String ”ACTION”

Combinations of Parsers

With the implicit conversion feature and the implemented parsers in the *JavaTokenParsers* object, the following lexemes can be parsed:

- Any constant String (cmp. Listing 4.5)
- An ASCII identifier (*ident* parser)
- A string literal encapsulated by double quotes (*stringLiteral* parser)
- An integer without a sign or with a negative sign (*wholeNumber* parser)
- An integer with a sign (*decimalNumber* parser)
- An integer with a sign, negative sign and *e* or *E* and a signed integer (*floatingPointNumber* parser)

For parsing more complex input, these parsers are combined with so-called *combinators*. A parser combinator is simply a function which takes two parsers and combines them in a certain way. The Scala API defines a great variety of these combinators. In the *as2smt* application, the following combinators have been used:

- $\mathbf{P_1 \sim P_2}$ is the sequence combinator. Two parsers connected with it will result in a new parser. This parser first applies P_1 followed by P_2.
- $\mathbf{P_1 | P_2}$ is the alternative compinator. The resulting parser tries to apply first P_1 on the input. If this fails, P_2 is applied to the input.
- **opt**($\mathbf{P_{optional}}$) is the optional combinator. $P_{optional}$ is applied to the input. If this fails the resulting parser does not fail in total.
- **rep**($\mathbf{P_{element}}$) is the repeat combinator. This parser will try to parse $P_{element}$ as many times as possible until it fails.
- **repsep**($\mathbf{P_{element}, P_{sep}}$) is the same as the repeat combinator but with a separation element. This parser will try to parse $P_{element}$ one time. When this does not fail, it will try to apply P_{sep} and then again $P_{element}$. This continues until P_{sep} or $P_{element}$ fails.
- $\mathbf{P_1 \sim> P_2}$ and $\mathbf{P_1 <\sim P_2}$ are equal to the sequence combinator but the left/right parser's output is ignored (cmp. Section 4.2.2).
- $\mathbf{P_{element}}$ * $\mathbf{P_{sep}}$ is a combination of an element parser and a separator parser (cmp. Section 4.2.3).

With these combinators, it is possible to write down any EBNF. The Scala code will resemble this grammar notation. For illustration, consider the action *repair* in Listing 4.6 (defined in Listing 2.2 in Line 35-41). For parsing such an action, let there be the EBNF defined in Listing 4.7[2].

```
'repair' :: (engine #= 0 #/\ integrity #= 1) => (
  % repair successful
  integrity := 0
  ;
  % repair fails
  integrity := 1
)
```

Listing 4.6: Action engange from the rocket steering software example

[2]Note that the illustrated EBNF only considers lexemes shown in the example in Listing 4.6. The limitation of one or two boolean relatation expressions is intended. The complete EBNF for actions in Action Systems can be found in Appendix A.

```
1 action    : ID ":: (" boolExpr ") => (" stmtBlock  ")";
2
3 boolExpr : boolRel ("#/\\" boolRel)?  ;
4
5 boolRel   : ID "#=" NUMBER;
6
7 stmtBlock : stmt (";" stmt)*;
8
9 stmt      : ID ":=" NUMBER;
```

Listing 4.7: EBNF for parsing actions (some rules and alternatives for readability omitted)

The transition from EBNF (cmp. Listing 4.7) to Scala parser combinator code (cmp. Listing 4.8) is rather simple. Every rule becomes one function, returning a *Parser* instance. References to *ID* and *NUMBER* (Lines 1,5 and 9) are mapped to the *ident* and *wholeNumber* parser (from the *JavaTokenParsers* class; Lines 2, 8 and 14). By type implicit conversion, the strings become autmatically *Parser* instances, parsing the respective string. Additionally, the question mark operator (Line 3) gets replaced by the *opt* combinator (Line 5). Furthermore, the star operator (Line 7) becomes the *repsep* combinator (Line 11). By applying these conversions, the code resembles still to EBNF but is in fact compiling and working Scala programming code for parsing the given input (Listing 4.6).

```
 1 def action: Parser[Any] =
 2   ident ~ ":: (" ~ boolExpr ~ ") => (" ~ stmtBlock  ~ ")"
 3
 4 def boolExpr: Parser[Any] =
 5   boolRel ~ opt("#/\\" ~ boolRel)
 6
 7 def boolRel: Parser[Any] =
 8   ident ~ "#=" ~ wholeNumber
 9
10 def stmtBlock: Parser[Any] =
11   stmt ~ repsep(stmt, ";")
12
13 def stmt:Parser[Any] =
14   ident ~ ":=" ~ wholeNumber
```

Listing 4.8: Scala code for parsing the EBNF described in Listing 4.7

The return types of the parser functions above is always *Any*. This omits a concrete type declartion as the example illustrates only the lexical and syntactic analysis with Scala parser combinators. For the translation process, an abstract syntax tree has to be created by the parser. The following section deals with the topic of processing the return values to AST nodes.

4.2.2 Abstract Syntax Tree Node Generation

As mentioned before, the return values have to be processed to abstract syntax tree nodes. Listing 4.1 shows the Scala *Parser* class. The type of the parser defines exactly the type of the *ParseResult* object. So, the return type of a constant string parser like shown in Listing 4.5 is a string, in fact the constant string itself. The *repsep* combinator returns a list of the element's type. The optional combinator *opt* returns a parser of type *Option* with the inner type of the return type of the optional parser. With sequential composition of parsers, this is more complex. Apart from the $\sim$ combinator, there exists also a type with the same name which has to type parameters T and U [3]. A parser formed by the sequential composition

[3]A syntactic sugar feature of Scala enables that types with two type parameters can be written also like infix left-associative operator: `~[T,U]` becomes `T ~ U`.

of parser with types *A* and *B* has the return type ~`[A, B]`, `A` ~ `B` respectively. Listing 4.9 shows the correct return types for two parser methods.

```
def action: Parser[Any] =
  ident ~ ":: (" ~ boolExpr ~ ") => (" ~ stmtBlock  ~ ")"

def boolExpr: Parser[String ~ String ~ String ~ Option[~[String,String
    ~ String ~ String]]] =
  boolRel ~ opt("#/\\" ~ boolRel)

def boolRel: Parser[String ~ String ~ String] =
  ident ~ "#=" ~ wholeNumber

def stmtBlock: Parser[String ~ String ~ String ~ List[String ~ String ~
    String]] =
  stmt ~ repsep(stmt, ";")

def stmt: Parser[String ~ String ~ String] =
  ident ~ ":=" ~ wholeNumber
```

Listing 4.9: Corrected return types (cmp. Listing 4.8; ignored *action* return type for readability)

Additionally to the combinators mentioned in the previous section, there exist the two functions:

- **P ^^ f** applies at first the parser P on the input. If this is successful, the conversion function f will be applied to the output of P. The input of f is the parser P (also if P is a combination of parsers).
- **P ^^^ f** works the same as the first operation except of ignoring the output of P. This is necessary when parsing constant values. So, f will always return a constant in this case if the input is parseable by P.

When using these combinators, AST nodes can be constructed instead of obscure concatenation of types. In the *as2smt* application, the nodes are declared as classes of type *Translation* (cmp. Section 3.3.3). Still, the conversion function has to handle a concatenation of types. For being able to distinguish the structural part of the input, i.e. the individial parse results, a case expression is used. Besides for pattern matching (cmp. Section 3.3.3), sequences of case expressions (wrapped in curly brackets) can also be used as partial functions (cmp. [70, p.291]). Listing 4.10 shows the AST node generation through conversion functions which are such case sequences.

```
def action: Parser[Action] =
  ident ~ ":: (" ~ boolExpr ~ ") => (" ~ stmtBlock ~ ")" ^^ {
  case actionName ~ _ ~ guard ~ _ ~ stmtList ~ _ => Action(actionName,
      guard, stmtList)
}

def boolExpr: Parser[List[EEquals]] =
  boolRel ~ opt("#/\\" ~ boolRel) ^^ {
  case a ~ None => a :: Nil
  case a ~ Some(_ ~ b) => a :: b :: Nil
}

def boolRel: Parser[EEquals] = ident ~ "#=" ~ wholeNumber ^^ {
  case a ~ _  ~ b => EEquals(EVar(a), EConst(b.toInt))
}

def stmtBlock: Parser[List[Assignment]] = stmt ~ repsep(stmt, ";") ^^ {
  case a ~ b => a :: b
}

def stmt: Parser[Assignment] = ident ~ ":=" ~ wholeNumber ^^ {
  case a ~ _ ~ b => Assignment(a, AEInteger(b.toInt))
}
```

Listing 4.10: Scala code for AST generation

4.2.3 Precedence of Operators

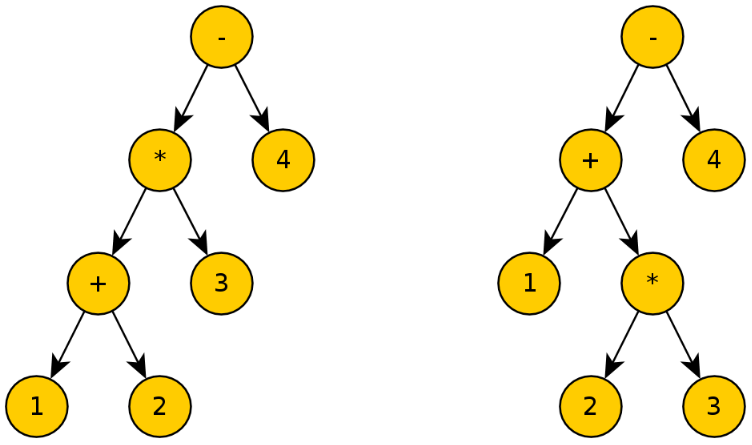


Figure 4.2: Incorrect (left) and correct (right) AST of the expression $1 + 2 * 3 - 4$

When parsing arithmetic or boolean expressions, it has to be dealt with operator precedences. For example, there is the expression $1 + 2 * 3 - 4$. The mathematical rule *multiplication and division before addition and subtraction* holds. Figure 4.2 presents an incorrect and the correct AST of this expression. Actually, this problem is already solved in the EBNF by using the *precedence climbing method* [77]. There, every precedence level is declared as an own parsing rule.

```
expression    : prec1-expression;

prec1-expression : prec2-expression (prec1-op prec1-expression)*

prec2-expression : element (prec2-op prec2-expression)*

element     : ID | NUMBER
```

Listing 4.11: Precedence climbing method in EBNF

For the parser generator *ANTLR*, there exists an alternative solution. For Scala, Mcbeath suggests in his blog a much cleaner solution [64]. It makes use of the star parser combinator and the fact that combinator rules are functions which can also take input parameters. The star combinator combines an element parser $P_{element}$ with a separator parser P_{sep}. From the input consumption point of view, this works exactly like the *repsep* parser combinator (cmp. 4.2.1). But the handling of the result objects is different. Instead of ignoring the return value of the separator parser and putting all results of the element parser into a list, the separator parser combines both sides to one result object. This is applied in a left associative way.

For example, the expression `1 + 2 + 3` can be parsed by the code in Listing 4.12. In the *as2smt* application, the *Translation* trait is also a base trait for all AST node types. When applying the example input to the *arithExpr* function, the following steps will proceeed:

1. the *arithFactor* parser consumes `1` and outputs `AEInteger(1)`,
2. the *arithOp* parser consumes `+`,
3. the *arithFactor* parser consumes `2` and outputs `AEInteger(2)`,
4. the *arithOp* parser consumes `+`,
5. the *arithFactor* parser consumes `3` and outputs `AEInteger(3)`,
6. the *arithOp* parser outputs Sum of output of step (1) and (3),
7. the *arithOp* parser outputs Sum of output of step (5) and (6).

```
def arithExpr : Parser[Translation] =
  arithFactor * arithOp | arithFactor

def arithOp: Parser[(Translation, Translation) => Translation] =
  "+" ^^^ { (a: Translation, b: Translation) => Sum(a,b) }

def arithFactor: Parser[Translation] =
  wholeNumber ^^ {case a => AEInteger(a)}
```

Listing 4.12: Example for star combinator usage

Listing 4.13 shows an extention of the example from Listing 4.12. The top level combinator is also the *arithExpr* function (Line 4). It distinguishes between a simple arithmetic factor (Line 10) and an *arith* expression (Line 6) where the precedence decision is encapsulated. Initially, the level is set to 1 (the minimum precedence level). Then, parser will behave as follows on the input $1 + 2 * 3 - 4$ (cmp Figure 4.2).

1. At first, the precedence level is 1. So, the *arithOp* parser checks the input for plus or minus operations. In case of the example, there is one plus and one minus operation. Due to the left-associativity, the root node of the expression is the minus operation, having the plus operation as a left child.

2. Then, the input is checked for operations of precedence level 2, so multiplication or division. In the example, there is one occurrence of a multiplication. It is the element on the right hand side of the plus operation. So, the multiplication becomes the right child of the plus AST node.
3. Afterwards, the level has increased to 3. The *arith* parser delegates to the *arithExpr* parser which consumes the numbers.

The advantages in using this method over the *precedence climbing method* directly in EBNF are extensibility and readability. It is easy to spot all precedences for one kind of expression. Adding another precedence level or adding new operators to a precedence level is simple.

```scala
val minPrec = 1
val maxPrec = 2

def arithExpr: Parser[Translation] = (arith(minPrec) | arithFactor)

def arith(level: Int): Parser[Translation] =
  if (level > maxPrec) arithFactor
  else arith(level + 1) * arithOp(level)

def arithFactor: Parser[Translation] =
  ident ^^ { case ident => AEVar(ident) } |
  wholeNumber ^^ { case wholeNumber => AEInt(wholeNumber.toInt) }

def arithOp(level: Int): Parser[((Translation, Translation) =>
   Translation)] = {
  level match {
    case 1 =>
      PLUS ^^^ { (a: Translation, b: Translation) => AEAdd(a, b) } |
      MINUS ^^^ { (a: Translation, b: Translation) => AESub(a, b) }
    case 2 =>
      TIMES ^^^ { (a: Translation, b: Translation) => AEMul(a, b) } |
      DIV ^^^ { (a: Translation, b: Translation) => AEDiv(a, b) }
    case _ => throw new RuntimeException("bad precedence level " +
       level)
  }
}
```

Listing 4.13: Precedence in Scala: arithmetic expressions

4.3 Translation from Action Systems to SMT-LIB

This section discusses the translation from Action Systems elements to the SMT-LIB input language.

4.3.1 Variable Declaration

Every Action System has state variables. These have to be declared in SMT-LIB, too. In the Action System notation the needed information is spread among several lines of the file. Listing 4.14 shows the definition of the state variable in Line 2. Line 10 indicates the bounds of the referenced *int* type. With this combined information, the state variable can be defined in SMT-LIB. Listing 4.15 shows the SMT-LIB code for the variable *out*. A declaration of a variable is achieved in SMT-LIB by declaring a function with no parameters and with one return value. The boundaries of the type of the variable are simply added by an additional assertion.

```
% definition of state space
var([out], int).

[...]

% action system
[...]

% types
type(int, X) :- X in 0..2, labeling([],[X]).
```

Listing 4.14: Example for a variable definition in Action System notation in Prolog

```
1 (declare-fun out () Int)
2 (assert
3   (and
4       (>= out 0)
5       (<= out 2)
6   )
7 )
```

Listing 4.15: Variable declaration in SMT-LIB

4.3.2 Assignment

In Definition 6 (Section 2.1.1), the predicative semantics of an assignment of a variable has been explained. It states that an assignment updates a local variable to a value. All other variables are left unchanged. In SMT-LIB, there is no notion of assignment, only of equality. Translation of boolean expressions can be mapped directly to SMT-LIB. But assignment has to be handled in a special way. An example for an assignment in the Action System description in Prolog is shown in Figure 4.16.

```
a := a + b
```

Listing 4.16: Example for an assignment in Action System description in Prolog

```
(declare-fun a () Int)
(declare-fun b () Int)
(declare-fun a_post () Int)
(declare-fun b_post () Int)

(assert
  (= a_post (+ a b))
  (= b_post b)
)
```

Listing 4.17: Example for an assignment in SMT-LIB

By introducing intermediate variables for reassignments of variables, this semantical gap can be closed. For every reassignment, there exists a new variable with the name of the original variable identifier suffixed with an underline character and a counting number. For the last assignment the variable name consists of the original variable identifier suffixed with the string *_post*. Listing 4.17 shows the translation to SMT-LIB[4]. Any intermediate variable would follow the described naming scheme and

[4]For this example, the type boundary assertions (Lines 2-7 in Listing 4.15) have been omitted for brevity. Moreover, the

would also be declared like a_post in the example. The *post* variable represents exactly the primed variable declared in Section 2.1.1).

4.3.3 Do-od Block

The Do-od block is a continuous loop of a set of statements (cmp. Section 2.1.1). A simple call in the Do-od block will be translated to an instance of the action. For example, an action with the name *plustwo* consists of the content of Listing 4.17. Then, there will be just this line translated to SMT-LIB in place of the Do-od block call. If there is a non-deterministic choice of calls in the Do-od block, these are handled the same way it would be the case for normal statements. Sequential composition in the Do-od block is not supported by the current version of the application but inside actions. The two operations will be discussed in the following two sections.

4.3.4 Sequential Composition

Intuitively, sequential composition of statements should be translatable by simple conjunction. But Aichernig and Jöbstl state in their paper [9] that the predicative semantics is defined differently. There exists an intermediate state between both blocks. v_0 marks the output of block one and the input for block two.

$$B_1(v, v'); B_2(v, v') \quad =_{df} \quad \exists (v_0 : B_1(v, v_0) \wedge B_2(v_0, v'))$$

The problem with the intuitive way is that it maps the semantics correctly in the positive case but not in the negative one. For example, let there be two assignments $out := 1$ and $out := 2$. The positive case:

$$out_0 = 1 \wedge out_1 = 2 \quad \neq \quad \exists out_0 : out_0 = 1 \wedge out_1 = 2$$

By negating the expressions above, the semantic difference is inherent (assuming the domain of the variable out is $\mathbb{N}$):

$$\neg(out_0 = 1 \wedge out_1 = 2) \quad \neq \quad \neg(\exists out_0 : out_0 = 1 \wedge out_1 = 2)$$

$$(out_0 \neq 1 \vee out_1 \neq 2) \quad \neq \quad (\forall out_0 : out_0 \neq 1 \vee out_1 \neq 2)$$

$$(out_0 \neq 1 \vee out_1 \neq 2) \quad \neq \quad (false \vee out_1 \neq 2))$$

$$(out_0 \neq 1 \vee out_1 \neq 2) \quad \neq \quad (out_1 \neq 2)$$

As out can be any decimal number, it also could be 1. Therefore, the first statement evaluates to false in the correct translation. This can be shortened by simplifying the last statement. The difference can now be easily seen, as the first (wrong) translation permits a second case which enables the expression to evaluate to *true*. Because of the mutant gets translated in a negated way, the correct translation can only be achieved with quantifiers. Aichernig and Jöbstl present this possible translation error among others in [9]. Therefore, the usage of an existential quantifier in the SMT-LIB translation is obligatory. Listing 4.18 illustrates an example of the translation of the sequential composition of two statements: $out := 1$ and $out := 2$.

action/do-od block construct has been ignored as it will be explained in the next subsection.

```
(exists
  ((out_1 Int))
  (and
    (= out_1 1)
    (= out_2 2)
  )
```

Listing 4.18: Example for an sequential composition in SMT-LIB

4.3.5 Non-Deterministic Choice

In case of the non-deterministic choice (NDC), the intuitive approach works correctly: both blocks are connected by disjunction. The disjunction is not an exclusive but inclusive one. Hence, the operation NDC enables one of two branches to be executed. If the choice would be an exclusive one, the semantics of this construct would be changed: if an expression e consisting of an NDC between two instances of `a := 1`, translation to an exclusive choice would lead constantly to an unsatisfiable expression.

In the rocket steering software example, there exists the action *land* (cmp. Listing 2.2, Lines 24-33). It contains the non-deterministic choice expression presented in Listing 4.19. Listing 4.20 shows its translation to SMT-LIB (omitting variable declartions).

```
% perfect landing
integrity := 0
;
% damage on landing
integrity := 1
```

Listing 4.19: Example for a non-deterministic choice in Action System notation in Prolog

```
(or
  (= integrity_1 0)
  (= integrity_1 1)
)
```

Listing 4.20: Translation to SMT-LIB of Listing 4.19

4.3.6 Guarded Command

If a certain (guard) property has to hold for executing a block of statements, a guarded command is used. It is called *guard*, because it controls the execution of its body. It resembles to an *if* expression in a procedural programming language. Hence, the guard consists of a Boolean expression. Only when the guard gives his confirmation (i.e. is satisfiable), the guarded block can be applied. Logically speaking, this results in a connection by a conjunction. As actions are in fact named guarded commands, the translation is performed in the same way. When an action is called, its guard and its body have to hold.

```
'repair' :: (engine #= 0 #/\ integrity #= 1) => (
  % repair successful
  integrity := 0
  ;
  % repair fails
  integrity := 1
)
```

Listing 4.21: Translation of an assignment with a guarded command

Listing 4.21 shows an example of a guarded command in form of the action *repair* from the rocket steering software example (cmp. Listing 2.2). Listing 4.22 shows its translation to SMT-LIB code consisting of

- the guard (Lines 2-6),
- the action body being a non-deterministic choice (Lines 7-11), and
- the trace (Lines 12-13; will be explained in the next subsection).

```
 1  (and
 2    ; guard
 3    (and
 4      (= engine 0)
 5      (= integrity 1)
 6    )
 7    ; action body
 8    (or
 9      (= integrity_post 0)
10      (= integrity_post 1)
11    )
12    ; trace
13    (= trace 4)
14  )
```

Listing 4.22: Translation of an action

4.3.7 Trace

In an Action System, multiple actions can be defined. The information, which action has been executed (has evaluated to true respectively), has to be recorded. There is a certain order in the definition of the actions. This order is used as an index for every action to identify it by a number. This index has to be used instead of the actual, textual identifier due to the fact that textual values are not possible in SMT-LIB. This index starts at zero. Each action body is wrapped by a conjunction with the assignment of this identifier to a so called trace variable. Listing 4.22 illustrates the translation of the *repair* action (cmp. Listing 2.2) to SMT-LIB code including the trace information (Line 13). The trace value 4 indicates that the action *land* (being the fourth declared action) has been called.

4.3.8 Parameters

Each action may have one or more parameters. In the do-od block these parameter can be set for every action call. The arity of the call and the action itself have to match up like in a standard function call. In case of the *as2smt* implementation just two kinds of parameters are allowed: numeric values and parameter variables. Numeric values are simply constants. Parameter variables can be interpreted by setting a range of possible values for a parameter. In the do-od section such a variable can be declared. This declaration includes a specific type. Types in an Action System are defined by a minimum and a maximum integer value. Listing 4.23 illustrates an example.

```
% [...]

% ACTIONS
actions (
  action1(A,B) =>  % [...]
),

% do-od
dood (
  [D:int]:action1(1,D)
)

% [...]

type(int, X) :- X in 0..2, labeling([],[X]).
```

Listing 4.23: Example: Action System with one action call with two parameters: one numerical (1) and one parameter variable (D)

Additionally to the information of the action identifier, parameter values have to be stored in separate variables. For each parameter, there is an additional trace variable. Hence, per Action System the highest arity defines the number of needed parameter trace variables. Listing 4.24 shows the translation of Listing 4.23 to SMT-LIB. The naming convention for such variables is *trace* plus *underline* plus *parameter index* (starting at zero).

```
; [...]

(declare-fun D () Int)
(assert (and
 (>= D 0)
 (<= D 2)
))

(assert
  (and
    ; [...]
    (= trace 0)
    (= trace_0 1)
    (= trace_1 D)
  )
)
```

Listing 4.24: Example: Assignment of parameter trace variables

4.4 Improving Performance

When applying the translation rules listed in the previous section, the Action System can be properly translated and solved by the used SMT solver. When discovering the time difference between Action Systems with and without sequential composition (cmp. Section 6.3.1), one can assume that the used existential quantifier in the sequential composition translation (cmp. Section 4.3.4) is the reason for this. In fact, at the time this thesis was written, just two SMT solvers supported quantifiers at all. Actually, when negating the mutant Action System, the existential quantifier becomes a universal quantifier. This quantification has been identified to be crucial with respect to performance.

This section presents an alternative way to the previously described translation from Action Systems to SMT-LIB code. This improved method makes use of a propositional logic rule called *one-point rule*. Its application in the context of this thesis represents actually symbolic execution.

4.4.1 Quantifier Elimination with the One-Point Rule

Propositional logic offers a conversion rule for expressions containing existential quantifiers. By applying the one-point rule (OPR), these quantifiers can be eliminated. It is defined as shown in Definition 11.

Definition 11. ***One-Point rule***

$$\exists x : x = e \wedge P(x) \quad \Longleftrightarrow \quad P(e)$$

The one-point rule states that if and only if a variable x is bound to a fixed value e, i.e. the expression e does not contain any bound variable, it is possible to substitute x by e and eliminate the existential quantification.

Aichernig and Jöbstl apply this quantifier elimination in [7]. Furthermore, they state that this logical rule application is only possible if and only if the value e is determinstic. If e contains a non-deterministic choice operation, the one-point rule can not be used as e would then be not fixed. Applying the one-point rule to the sequential composition of the assignments $x := e_1$ and $x := x + e_2$ (where e_1 and e_2 do not contain references to bound variables) results in the following:

$$\exists x_1 : (x_1 = e_1 \wedge x_2 = x_1 + e_2) \quad \overset{opr}{\Longleftrightarrow} \quad x_2 = e_1 + e_2 \wedge x_{post} = e_1 + e_2$$

In the second term, x_1 was substituted by its assigned value e_1. As the expression $e_1 + e_2$ does not contain any bound variables, assigning symbolic values is also possible. We relabel x_2 to x_{post} to show that this variable assignment is part of the post state of the Action System. By keeping track of the current value not in the formula but in a symbol table, we can reduce this to the following:

$$x_{post} = e_1 + e_2$$

This replacement of variables by their symbolic values represents the symbolic execution of the expression (in Prolog Action System notation) `x := e1, x := x + e2`.

By applying the quantifier elimination as shown in Definition 11, any variable occurrences in any expression e can be replaced by their symbolic values. So, e becomes a fixed value. For keeping track of such values, there has to be some kind of symbol table just like for compilers but for symbolic values. By the time an assignment is *executed*, the variables possibly bound in e can be replaced by their current symbolic values for being able to eliminate the existential quantifer. This kind of execution is called *Symbolic Execution*.

4.4.2 Introducing Symbolic Execution

Back in 1969, Balzer [20] performed a similar technique to the previously presented one in his *Extendable Debugging and Monitoring System* (EXDAMS). King formalised Symbolic Execution [56] in 1974. Further research has been performed by King [57], Boyer et al. [27] and Clarke [35]. King suggests that symbolic execution performs the same way as traditional execution in *trivial cases involving no symbols* [56]. The behaviour changes only when symbols are encountered in the following two cases:

- **Computation of Expressions:** the *value* that a symbol has to be assigned to, contains a symbol itself. Instead of assigning a concrete value to the symbol, the symbolic expression is saved in the symbol table. If an expression is dependent on an input variable (parameter), the evaluation is *delayed*. Conside the assignment $A := P + e$ where e does not contain any bound variables. P represents an input value (P for parameter), so it will not have a concrete value until it is executed *for real* (not symbolically). But the symbol A gets assigned the symbolic value $P + e$. The next time an expression containing A is encountered, it will be substituted by the symbolic value $P+e$.
- **Conditional Branching:** the execution path comes across a conditional branching operation. Considering a typical conditional program statement: `IF` B `THEN` S_1 `ELSE` S_2. The condition B determines if the left code block S_1 or the right code block S_2 is executed. It is not deterministic which branch will be executed. The execution path splits into two at this point. One path fulfills B, the other one fulfills $\neg B$. $B \vee \neg B$ represent the so called path conditions [56].

```
1 if (x > 0 && y > 0 && z > 0){         // pc -> x > 0, y > 0, z > 0
2   x = y * z;                          // x -> y * z
3   y = z - x;                          // y -> z - y * z
4   z = x + y;                          // z -> y * z + (z - y * z)
5 }
```

Listing 4.25: Example for symbolic execution

For illustrating symbolic execution, let there be the example shown in Listing 4.25. This Java code snippet consists of an *if* expression and three assignment statements. When executing this code symbolically, the following values get updated in the symbol table. At first, the path condition is set to $x > 0 \wedge y > 0 \wedge z > 0$ (Line 1). At this point, branching occurs. For this example, we consider only the *if* branch as the *else* branch does not update the variables. The first statement (Line 2) updates the variable x to the symbolic value $y \times z$. Then, variable y gets assigned to $z - x$. As, there is already a symbolic value set to x, its occurrence has to be replaced with it. Therefore, the symbolic value of y is $z - y \times z$. Finally, the last statement gives z the symbolic value $y \times z + (z - y \times z)$ by replacing x and y in the expression $x + y$ (Line 4).

Instead of executing the statements sequentially, symbolic execution outputs symbolic expressions per variable which depend only on the input paramters. In this way, a code snippet like shown in Listing 4.25 becomes a mapping of initial values to primed values.

4.4.3 Data Structures for Symbolic Execution

With respect to the translation of Action Systems, the *compiler* - becoming partly a symbolic execution environment - has to adjust its behaviour for the following kinds of operations:

- Assignment
- Guarded Command
- Sequential Composition
- Non-Deterministic Choice

The do-od block, trace variables and parameters will still be translated the same way as before. For being able to perform symbolic execution, two data structures are needed: the abstract syntax tree (AST) of the expression and a symbol table. ASTs of expressions have already been used for the translation shown in Section 4.3. The symbol table keeps track of the symbolic values assigned to the variables.

As mentioned, only some parts of the Action System translation are executed symbolically. In fact, only the bodies of the actions called in the do-od block are translated in this way. A single action body may contain one of the following nodes:

- **An assignment node (ASS)** assigns a value to a variable. This value is an arithmetic expression (cmp. Section 3.3.3). According to its definition, it might also contain references to bound variables.
- **A sequential composition (SEQ)** composes two of the AST node types mentioned in this list sequentially.
- **A non-deterministic choice (NDC)** states that one of two AST nodes (also type from this list) is executed. This is decided in a non-deterministic way.
- **A guarded command (GC)** consists of a *guard* (GD) and a *body* (BDY). The guard is a constraint in form of a Boolean expression (cmp. Section 3.3.3). The body is an AST node of any type. This AST node just gets executed if the constraint is fulfilled.

The symbol table consists of the following data structures (cmp. Section 3.3.3):

- **A symbol map** contains the name of a variable and its corresponding value. This value is a symbolic value in form of an abstract syntax tree of an expression.
- **A list of guard constraints** keeps track of all guard constraints found on an execution path. By the conjunction of the elements of this list, the so called *path condition* is formed.

4.4.4 Symbolic Execution Without Branching

At first, we consider only symbolic execution without branching. This subsection shows the symbolic execution of action bodies without non-deterministic choice. No branching means that there exists only one execution path. Figure 4.3 visualises that there is vector of state variables with their initial (symbolic) values v and the same vector with possibly different values v' after the symbolic execution of sequential compositions of assignments and guarded commands.

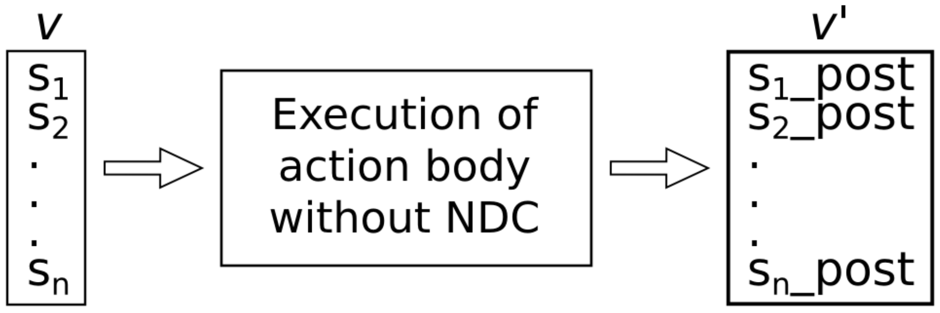


Figure 4.3: Vector v holds all symbolic values of the state variables s_1 to s_n. After executing an action body which does not contain any non-deterministic choice, there is one new vector v'.

For translation to SMT-LIB code, this means that all primed (`_post` suffixed) variables are equal to their symbolic values. When interpreting the code snippet from Listing 4.25 as an Action System (in a Java notation), there exists one action with its body consisting of a guarded commands and three sequentially composed assignments. Listing 4.26 shows the SMT-LIB code translation consisting of the path condition and the path assignments.

```
(and
  ; path condition
  (and
    (> x 0)
    (> y 0)
    (> z 0)
  )
  ; path assignments
  (and
    (= x_post
      (* x y)
    )
    (= y_post
      (- z (* y z))
    )
    (= z_post
      (+ (* y z) (- z (* y z)))
    )
  )
)
```

Listing 4.26: Translation of result of symbolic execution

Symbolic Execution of Assignments & Sequential Composition

The elementary object of an action body is the assignment. Algorithm 1 illustrates the symbolic execution of an assignment implemented in the *as2smt* application. At first, the current symbolic value for the symbol that should be assigned to a new value is retrieved (Line 2). Then, the one-point rule is applied to the new value (Line 3) and the symbol table is updated (Line 4). Sequential composition of assignments in an Action System is in fact the sequential symbolic execution. So, it is just the repeated call to Algorithm 1 with the current symbol table. Algorithm 2 illustrates this.

Algorithm 1 Symbolic execution of an assignment

```
Input: symbol table $st$, AST node of an assignment $ass$ with a left hand side (variable name, $lhs$) and
    a right hand side (symbolic value, $rhs$)
Output: updated version of symbol table $st$
1: function SYMBOLICEXECUTE(ass, st)
2:     $oldValue := st.getValueOfSymbol(ass.lhs)$
3:     $newValue := replaceAllVariables(ass.rhs, st)$
4:     $st.setValue(ast.variable, newValue)$
5:     return $st$
6: end function
```

Algorithm 2 Symbolic execution of a sequential composition

```
Input: symbol table $st$, AST node of a sequential composition $sc$ with a $first$ and a $second$ AST node
Output: updated version of symbol table $st$
1: function SYMBOLICEXECUTE(ass, st)
2:     $newSt := symbolicExecute(sc.first, st)$
3:     return $symbolicExecute(sc.second, newSt)$
4: end function
```

Symbolic Execution of Guarded Commands

There is also just one possible execution path if we take an action body which contains guarded commands but no non-deterministic choices. Algorithm 3 presents the steps that are taken to execute a guarded command symbolically: at first, the guard is freed from any variable occurrences (Line 2; except of parameters - cmp. Section 4.4.2). Then, the guard condition is added to the path condition (Line 3). Finally, the execution continues with the body of the guarded command (Line 4) where its resulting symbol table is returned.

Algorithm 3 Symbolic execution of a guarded command

```
Input: symbol table $st$, AST node of the guarded command $gc$ consisting of a $guard$ and a $body$
Output: updated symbol table $st$
1: function SYMBOLICEXECUTE(gc, st)
2:     $newGuard := replaceAllVariables(gc.guard, st)$
3:     $st.addToPathCondition(newGuard)$
4:     return $symbolicExecute(gc.body, st)$
5: end function
```

If there are multiple assignments and guarded commands connected by sequential composition, these are executed in the depth-first search order. Let there be the following action body as an example:

```
((a #>= 1) => (x := 1)) ,
(b #>= 1) => ((x := x + 2) ,
(x := x - a - b))
```

Additionally, let `a` have the initial value `1` and `b` be a parameter value. Figure 4.4 illustrates the AST of this expression, and enumerates the nodes by their execution order. By the end of the symbolic execution the data structures contain the following data:

- *Symbolic values:*

  `a` $\rightarrow$ `1`
  `b` $\rightarrow$ *same value as before execution (parameter value)*
  `x` $\rightarrow$ `1 + 2 - 1 - b`

- *Path condition:* `1 >= 1` $\wedge$ `b >= 1`

4.4.5 Symbolic Execution with Branching

As seen in the previous section, the one-point rule can be applied easily when there is no non-deterministic choice in an action body. When there exists one, the symbolic value in the symbol table is ambiguous. For example, the expression `(x := 1 ; x := 2), y := 1 + x` contains two execution paths. In the first one, the symbolic value 1 is assigned x and therefore y has the symbolic value $1 + 1$. In the second case, the symbolic values are 2 for x and $1 + 2$ for y. Each occurring non-deterministic choice results in a forking of the execution. So, if there exist n NDC nodes in one expression, there exist $n + 1$ execution paths. Every execution path results in one state vector v_i' (cmp. Figure 4.5). This motivated a normalisation of the abstract syntax tree before applying symbolic execution. This normalisation assures that the outermost operation is the non-deterministic choice between execution paths. Each path can then be symbolically executed as described in Section 4.4.4.

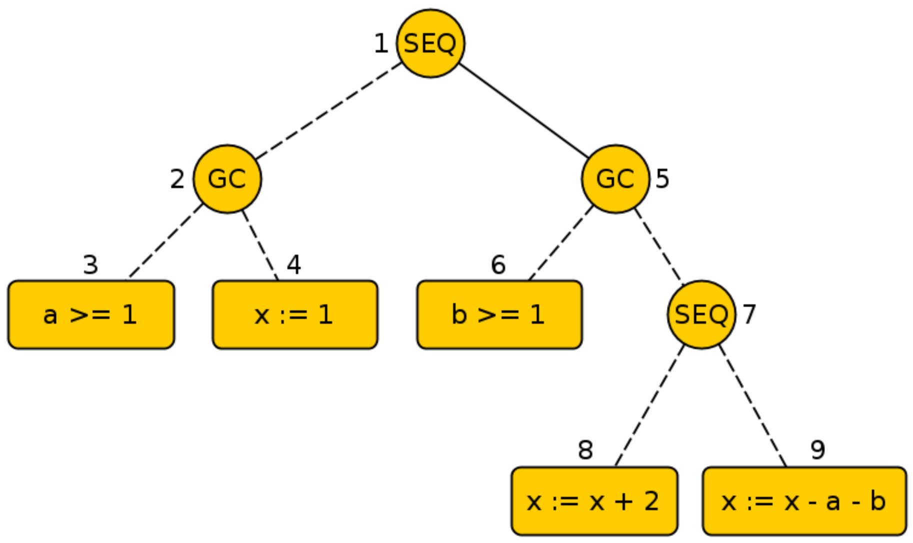


Figure 4.4: Symbolic execution: example for execution order of AST

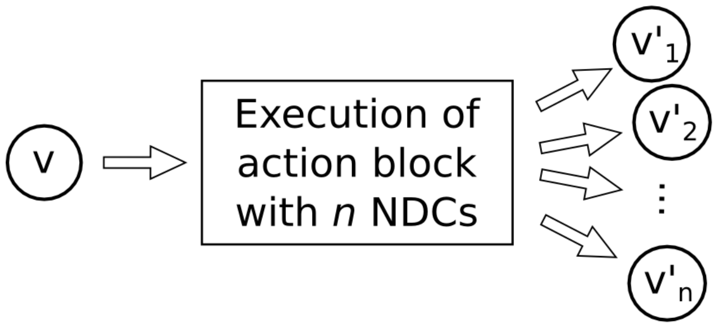


Figure 4.5: Vector v contains all state variable values (cmp. Figure 4.3). Executing an action body containing n non-deterministic choices results in n vectors v'_i.

Normalisation of the Abstract Syntax Tree

As mentioned in Section 4.4.1, the quantified expression has be deterministic to apply the one-point rule. Hence, the term has to be transformed to have non-deterministic choice as the outermost operation. In fact, this transformation represents the application of the distributive law in Boolean logic. So, the resulting normalised form equals a disjunctive normal form (DNF). Algorithm 4 describes this normalisation of the AST. As input, the mentioned AST of one action body is taken. The function *normaliseAst* returns a list of possible execution paths. It is called recursively. The returned list represents the different execution possibilities. The algorithm deals with four cases:

- **Assignment:** the AST node parameter's type ($currentAstNode$) is an assignment (Lines 2 & 3). It returns the node as a single element of a list.
- **Guarded Command:** the AST node parameter's type is a guarded command (Lines 4 - 10). The function is invoked recursively on the body of the guarded command. Consecutively, the returned list of execution paths is looped to prepend the original guard to each element. Finally, this list of modified elements is returned and the algorithm terminates.
- **Sequential composition:** the AST node parameter's type is a sequential composition (Lines 11 - 20). In this case, we have to call the function recursively twice: once on the left part of the sequential composition and once on the right part. These calls return two lists with execution

Algorithm 4 AST normalisation algorithm

Input: an AST node $currentAstNode$
Output: a normalised AST node

```
function NORMALISEAST(currentAstNode)
    if $typeof(currentAstNode) = Assignment(variable, value)$ then
        return $[currentAstNode]$
    else if $typeof(currentAstNode) = GuardedCommand(guard, body)$ then
        $astList := []$
        $bodyAstList := normaliseAst(currentAstNode.body)$
        for all $ast \in bodyAstList$ do
            $astList.enqueue(GuardedCommand(currentAstNode.guard, ast))$
        end for
        return $astList$
    else if $typeof(currentAstNode) = SequentialComposition(left, right)$ then
        $leftAstList = normaliseAst(currentAstNode.left)$
        $rightAstList = normaliseAst(currentAstNode.right)$
        $astList := []$
        for all $l \in leftAstList$ do
            for all $r \in rightAstList$ do
                $astList.enqueue(SequentialComposition(l, r))$
            end for
        end for
        return $astList$
    else if $typeof(currentAstNode) = NonDeterministicChoice(choice1, choice2)$ then
        return $[normaliseAst(choice1), normaliseAst(choice2)]$
    end if
end function
```

possibilities. Every element of the first (left) list and every element of the second (right) list are composed sequentially. Finally the function returns this list of sequential compositions.

- **Non-deterministic choice:** the AST node parameter's type is a non-deterministic choice (Lines 21 - 22). First, the function is called recursively with the left choice AST as a parameter. Then, the right choice is handled in the same way. The return values of both calls are composed to a list which is returned as a final result of the algorithm.

A normalised AST represents a set of possible execution paths per action. There is no more branching on a path. One execution path still consists of sequentially composed assignments and guarded commands. Figure 4.6 shows two abstract syntax trees of one action body. The left tree is an example for a not normalised abstract syntax tree, because it contains non-deterministic choice nodes (NDC) not positioned at the root. The right tree shows the normalised representation of the left AST. All of the three paths (separated by dashed lines) just contain guarded commands (GC - splits into body (BDY) and guard (GD)), sequential composition (SEQ) and assignments (ASS).

Translation of an Execution Path

As each execution path has no data dependency on any other execution path, multiple paths are simply combined by a disjunction of their translations. As mentioned in the beginning of this section, the do-od block, the parameters and the trace variables are still handled the same way as in Section 4.3.

With the presented translation in this section, the performance of the execution has been improved drastically (cmp. Section 6.3.1). So, the translation of the defined subset (cmp. Appendix A) of the Ac-

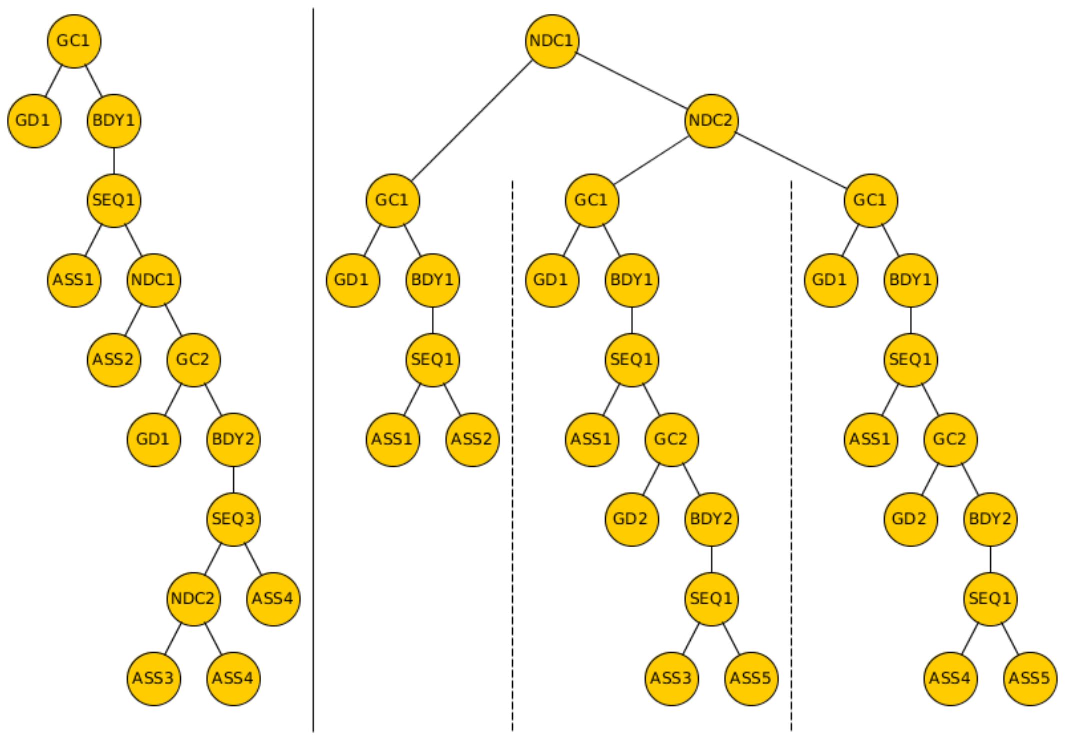


Figure 4.6: Original (left) and normalised (right) AST

Algorithm 5 Translation of execution paths to SMT-LIB (String representation)

Input: list of execution paths $executionPathList$

Output: translation of execution paths

```
1:  function TRANSLATEPATHS(executionPathList)
2:      $resultList := []$
3:      for all $path \in executionPathList$ do
4:          $st := symbolicExecute(path, ())$
5:          $pathAssignments := []$
6:          for all $(variableName) \in st.symbolMap$ do
7:              $variableValue := st.getValueOfSymbol(variableName)$
8:              $assignInSmtLib := translateToSmtLib(variableName, variableValue)$
9:              $pathAssignments.enqueue(assignInSmtLib)$
10:         end for
11:         $pathCondition := translateToSmtLib(st.getGuardList())$
12:         $pathTrans := translateToSmtLibConjunction(pathAssignments)$
13:         $resultTrans := translateToSmtLibConjunction(pathTrans, pathCondition))$
14:         $resultList.enqueue(resultTrans)$
15:     end for
16:     return $translateToSmtLibDisjunction(resultList)$
17: end function
```

tion System Prolog notation language to SMT-LIB is complete. Further work may involve the translation of features of the input language to SMT-LIB.

5 Refinement and Reachability

The previous chapter described the translation of Action Systems (notated in Prolog) to the SMT-LIB input language. This chapter discusses how the *as2smt* application uses this translation to generate *counterexamples* from which test cases can be created. It presents the steps *Refinement Check* and *Reachability Check* of the as2smt application (cmp. Figure 3.1). In Section 2.1.2, non-refinement of Action Systems has been discussed. It states that an Action System AS^m does not refine an Action System AS^o if and only if the following properties hold:

- there exists an unsafe state in the mutant AS^m which is not allowed by the specification AS^o.
- this unsafe state is reachable from the initial state defined in the specification AS^o.

These two properties for non-refinement of Action Systems are computed consecutively. Aichernig and Jöbstl perform the same approach in their *as2csp* implementation [8]. In fact, the problem their tool confronts is the same as this thesis: finding unsafe states by checking non-refinement of two Action Systems.

This chapter is divided into two sections: the first section describes how refinement of two Action Systems is computed. It ignores reachability completely as this will be discussed separately in the second section. In this consecutive section, different strategies for computing reachability check are presented.

5.1 Refinement Check

In this section, the refinement check of two Action Systems - a specification and a mutant - is discussed. At first, the notion of the *non-refinement formula* is introduced. Furthermore, its resolution by an SMT solver is explained. Then, an efficient way of checking refinement of two Action Systems is discussed.

5.1.1 Non-Refinement Formula

```
; [... variable definitions ...]

; negated specification
(assert
  (not
     ; [... specification action calls ...]
  )
)

; mutant
(assert
  ; [... mutant action calls ...]
)
```

Listing 5.1: Non-refinement formula: negated specification and positive mutant

The *non-refinement formula* combines the translations of the specification and the mutated Action System. In Section 2.1.2, Theorem 1 defined non-refinement of Action Systems. Ignoring the *reachable* term (as this will be discussed in the next section), non-refinement of Action Systems is given when at least one action A_i^m of the mutant AS^m and the negation of all actions[1] of the specification AS^o have

[1]As every action is distinct, it is sufficient that the non-refinement formula cosists of the mutated action and its specification counterpart - not all actions. Nevertheless, this translation has been chosen for this thesis. An important feature of the implemented tool relies on it - incremental solving (cmp. Section 5.3.2).

to hold for a state v. In SMT-LIB, it is rendered as shown in Listing 5.1. There are two consecutive assertions: one for the negated specification and one for the mutant. If these two assertions are both satisfiable[2], the SMT solver has found at least one valuation for all state variables and the trace variables. In this thesis, the combination of initial (unprimed) and post (primed) values of the state variables and the trace information (cmp. Section 4.3.7) is called a *model*[3]. Such a model is represented in the *as2smt* application by the class *ActionSystemModel* (cmp. Section 3.3.5). Figure 5.1 presents a graphical notation for the terms *state*, *trace* and *model*. A trace can either show a single transition like presented in the Figure (trace length is one). Or, it may cover multiple transitions (trace length greater than one). This notation will be used in further illustrations.

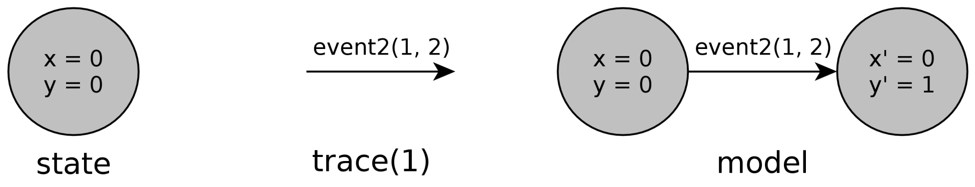


Figure 5.1: Graphical notation of state (left), trace (middle, trace length in parenthesis) and model (right)

As an example for non-refinement, Listing 5.2 shows an Action System with two state variables x and y. Assume their possible values are just zero and one. By changing the *less* operator < in Line 3 to the *less or equal* operator <=, a mutated Action System is defined. By translating those two Action Systems and combining them (as shown in Listing 5.1), a non-refinement SMT-LIB formula is created. With the two possible values - zero and one - for x and y, there are just four possible combinations which can serve as initial values. When choosing $x = 1$ and $y = 1$ the specification part of the non-refinement formula becomes unsatisfiable, i.e. it's negation is satisfiable.

```
 1  actions (
 2    event1::(y < 1) => (x := 1)
 3    event2::(x < 1) => (y := 1) /* event2::(x <= 1) => (y := 1) */
 4  ),
 5
 6  dood (
 7    event1
 8    ;
 9    event2
10  )
```

Listing 5.2: Example for an Action System

In Figure 5.2, the models for the example are shown in the previously described graphical notation (cmp. Figure 5.1). The *refinement* group models hold for the specification and the mutant. The model shown in the *non-refinement* group holds for the mutant but not for the specification. This is exactly the constraint that the non-refinement formula has to fulfil.

As stated in Section 3.2, the *as2smt* tool takes one specification and multiple mutants as input. For each combination of a specification and a mutant, there is one non-refinement formula. The translation process described in the previous chapter also takes care of the difference between the two kinds of Action Systems. The specification has to be translated first and includes the variable definitions. Then, the mutant is translated but without any variable definitions. This is important to note because this marks a limitation of the tool. There must not exist any mutations concerning the state variables apart from their valuation. For state variable definitions, neither changes of their set, nor changes to their names are

[2]In an SMT-LIB script, every declared assertion has to be satisfiable to come to a satisfiable conclusion.

[3]This name has been chosen in accordance to SMT-LIB[24]. There, the option *:produce-models* has to be set for retrieving an valuation for all variables in case of SAT.

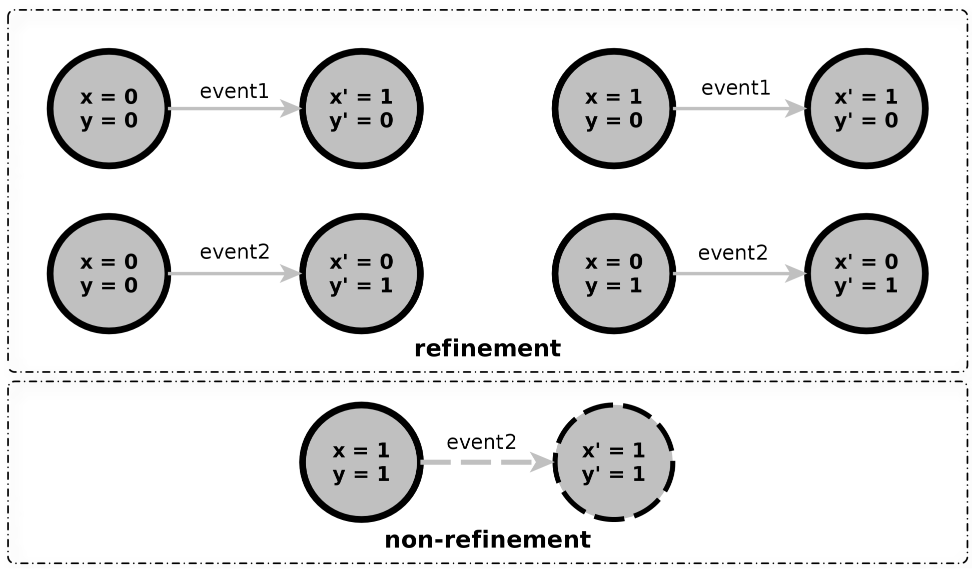


Figure 5.2: State variables: transitions from initial to post values, one counterexample to refinement at the bottom.

possible to detect with the presented tool. Further limitations have been listed in Section 3.5.

When creating the non-refinement formula, it is not necessary to translate the whole mutant, but just the mutated action. This has been shown by the disjunction in Theorem 1. Therefore, the mutated action has to be identified. The following two sections will present different approaches to achieve this identification.

5.1.2 Evaluating the Non-Refinement Formula

The first identification of the mutated action method works with the non-refinement formula. After parsing, the data structure *ActionSystem* (cmp. Figure 3.3) will contain - among others - a list of all action calls. Algorithm 6 shows the pseudo code of the implemented procedure to find the mutated action with the aid of the solver. It takes the original and the mutated Action Systems as inputs. At first, the specification is translated into SMT-LIB in a negated way (*TranslationType* is *NegatedSpecification*; Line 2). Then, the algorithm steps through the list of all calls in the do-od block (Line 3). These are joined by a non-deterministic choice (cmp. Section 2.1.1). Just the names have to be extracted, as they represent the references to the actions. Subsequently, a new Action System is constructed which equals the mutant (search depth, variable, type and action definitions) but with a different do-od block. It consists only of a single call to the action to be tested (parameter *call*; Line 4). The next step translates this Action System to SMT-LIB (Line 5). By the conjunction of this translation with the negated specification, the non-refinement formula is constructed (cmp. Figure 5.1.1). This serves as input to the SMT solver. The solver may then output three possible values:

- **SAT:** *Satisfiable* indicates that there is a semantic difference between the mutated action and the according original action. The algorithm returns the mutated action name (Line 9).
- **UNSAT:** *Unsatisfiable* provides the information that this action of the mutant refines the action of the specification. The next action call has to be checked (Line 11). If the SMT solver outputs this result for all calls, the whole mutant refines the specification (Line 16).
- **UNKNOWN:** *Unknown* denotes that the SMT solver is not capable of deciding between the two previous results. In this case, the used SMT solver does not suffice to decide on this refinement problem (Line 13).

Algorithm 6 Find mutated action

Input: two Action Systems, the original s and a mutant m
Output: the identifier of the mutated action

```
 1: function FINDMUTATEDACTION
 2:     $negSpecTrans := ActionSystemTranslator.translate(s, TType.NegatedSpecification)$
 3:     for all $call \in m.calls$ do
 4:         $singleCallAS := ActionSystem(m.sDepth, m.varDefs, m.actions, call, m.typeDefs)$
 5:         $singleCallTrans := ActionSystemTranslator.translate(singleCallAS, TType.Mutant)$
 6:         $nonRefinementFormula := negSpecTrans \wedge singleCallTrans$
 7:         $result := SMTSolver.solve(nonRefinementFormula)$
 8:         if $result = SAT$ then
 9:             return $call.actionName$
10:         else if $result = UNSAT$ then
11:             continue
12:         else
13:             throw SMTReturnedUnkownException
14:         end if
15:     end for
16:     return $null$
17: end function
```

If a mutant contained multiple mutations, this algorithm should not stop after finding the first mutated action. At the current stage of the tool, higher-order mutants are not supported (cmp. Section 3.5).

```
% actions
actions (
  engage :: (integrity = 0 ∧ engine = 1 ∧ x = 0 ∧ y = 0) => (
    state := 1,
    engine := 1,
    x := 1,
    y := 1
  ),
  % [... all other actions from specification...]
)
```

Listing 5.3: Rocket steering software mutant

Previously, an Action System was defined that models a rocket steering software. Listing 2.2 shows its action definitions (cmp. state graph in Figure 1.2). Let there be a mutant which injects a fault into the `engage` action as presented in Listing 5.3. The guard condition checks if engine is turned *on* instead of turned *off*. At first, the Action System is translated to SMT-LIB code as described in the previous chapter (Algorithm 6, Line 2). For the translation type is set to *NegatedSpecification*, the translation will be wrapped by a negated assertion (cmp. Listing 5.1, Lines 4 - 8). As the do-od block of the mutant is unchanged, all calls stay the same as in the specification. In fact, the action call to `engage` is the first one listed. So, it is also the first iteration of the for each loop (Lines 3 - 15). A new Action System is created as a clone of the clone but just consisting of the call to `engage` in the do-od block. Then, this Action System is translated to SMT-LIB in a positive way (cmp. Listing 5.1, Lines 11 - 13). In fact, the String concatenation of the negated specification translation with this translation is a conjunction of both terms. Technically speaking, both assertions are put on the assertion stack before asking the SMT solver to look for a possible valuation of the declared variables. Finally, the solver outputs SAT and returns a valuation for all state variables and trace variable. The latter evaluates to 1 identifying the mutated `engage` action (cmp. Section 4.3.7).

Algorithm 7 Syntactic equality check

```
Input: two Action Systems, the original $s$ and a mutant $m$
Output: the identifier of the mutated action or None
1: function CHECKSYNTACTICALEQUALITY
2:     if $s.actions.size = m.actions.size$ then
3:         $callListSpec := s.getCallList()$
4:         $callListMutant := m.getCallList()$
5:         $callCheckResult := checkCallListEquality(callListSpec, callListMutant)$
6:         if $callCheckResult = Some(x)$ then
7:             return $callCheckResult$
8:         else
9:             $actionCheckResult := checkActionListEquality(s.actions, m.actions)$
10:            if $actionCheckResult = Some(x)$ then
11:                return $callListMutant.findCallofAction(callListMutant)$
12:            else
13:                return $None$
14:            end if
15:        end if
16:    end if
17:    throw UnsupportedMutationException
18: end function
```

5.1.3 Comparing Action Systems Syntactically

Aichernig and Jöbstl [7] presented an improvement for their tool which has been implemented by the *as2smt* application as well: using a syntactic check for finding the mutated action. As we only consider first-order mutants, there exists just one mutated action. In the example before, the mutated action was the first called action. In this way, the algorithm loops just for one iteration. If a non-mutated action is the first element in this list of calls, looping continues. Literally speaking, a non-mutated action means no changes at all have been performed in comparison to the original action. So, translating a non-refinement formula consisting of the specification action A_i^o and the non-mutated action A_i^m to SMT-LIB, an SMT solver is asked to find a model for the expression $A_i^m \vee \neg A_i^o$. As A_i^m and A_i^o are equal, this contradiction can not be fulfilled by any valuation, the SMT solver returns *unsatisfiable* (UNSAT).

In addition to this, a real-mutated action has to differ syntactically to the specification. Nevertheless, it may also be an *equivalent mutant* which differs in syntax but not in semantics (cmp. Listing 1.2 and 1.3). Still, the information of the mutated action name can be retrieved by a syntactical analysis.

After parsing the specification and the mutant, two abstract syntax trees are available (cmp. Section 3.3.3). These two objects are compared syntactically. With Scala, this was relatively easy to implement. When comparing non-primitive data types, Java will check referential equality. This means that both variables have to point to the same value on the JVM stack. In contrast to that, Scala checks equality by comparing all fields of the objects. In Scala, there exist no primitive data types like in Java. So, every value is an object.

In the *as2smt* application, the *SyntacticEqualityChecker* is responsible to evaluate which action or call differs between two Action Systems (cmp. Section 3.3.3). Algorithm 7 illustrates how this problem has been solved. One Scala feature is used here which has not been discussed yet. In Java, the value *null* indicates a missing pointer - a null pointer respectively. If a variable is allowed to be null, it has to be checked every time before its usage. This violates the principle of pure object orientation (cmp. Section 3.1.2). For this purpose, the *Option* type exists (cmp. [70], page 288ff). It may have one of two different values:

- **None** is used instead of a *null* value.

- **Some** contains a value of a variable which could have been nullable.

The function described in Algorithm 7 has the return value *Option[Call]*. So, there might be some *Call* value returned, there might be none (this is what an Option object in Scala represents). At first, it is checked, whether the two Action Systems contain the same amount of actions (Line 2). If this is not the case, an unsupported mutation is detected (Line 17, cmp. Section 3.5). Furthermore, the call lists are extracted and checked for containing the same elements (Lines 3-5). With this check, mutations of the parameters in action calls can be detected (Lines 6 & 7). If no mutation was found yet, the algorithm proceeds by checking each action (Line 9). Again, if a syntactic difference is found, the identifier of the called action is returned (Line 11). Otherwise, the algorithm concludes that the present Action Systems are syntactically equal which implies their semantic equality (and so their conformance).

A mutant and its original may differ syntactically but may have the same semantic meaning. This is the limitation of the syntactic analysis. Therefore, the found, *possible* non-refinement formula has to be checked once by the SMT solver. If the solver determines that the formula is unsatisfiable, the Action Systems are semantically equal.

An alternative to both presented algorithms would be to simply input the mutated action name as a parameter. At the time of mutant creation, this is a known information. If its forwarding is ensured (for example by some marking or similar), the effort of finding the mutated action can be omitted.

5.2 Reachability Analysis

In the previous section, the algorithms of determining the non-refinement formula (including all actions of the specification and the mutated action) has been presented. This section deals with the reachability analysis of possible unsafe states. It presents four different ways of its computation.

5.2.1 Initial Strategy

The initial strategy lists all models which fulfill the non-refinement formula. Algorithm 8 describes how to collect these models. The function takes two parameters: the non-refinement formula and the list of already found models. Initially, the list of found models is set to the empty list. The first step in the function is to execute the SMT solver with the non-refinement formula (Line 2). This will return a model if the formula has been satisfiable (Line 3). Otherwise, all models that fulfill the formula have been found (Line 9). Furthermore, the new model is appended to the found models list (Line 4). Additionally, it is translated to SMT-LIB in a negated way (Line 5). Then, this translation is conjuncted with the current formula (Line 6). Similarly to the *back jump clause* of the DPLL algorithm (cmp. Section 2.2), this term assures that the solver does not output the same state once again.

Listing 5.4 gives an example for *back jump clause* in SMT-LIB. The model notation of this example is exactly the one shown in Figure 5.1. In addition to the two assertions from the non-refinement formula (cmp. Listing 5.1.1), a third assertion is put on the assertion stack. As stated before, this resembles to a logical conjunction. It contains the negation of a previously found model which fulfills the non-refinement formula.

Algorithm 8 Find all possible models, initial strategy

Input: the non refinement formula and a list of models (initially the empty list)

Output: the list of all models fulfilling the non-refinement formula

```
function FINDMODELS
    $result := SMTSolver.solve(nonRefinementFormula)$
    if $result.satisfiability = SAT$ then
        $models.append(result.model)$
        $negModelTrans := translateNegatedModel(result.model)$
        $newNonRefFormula := nonRefinementFormula \land negModelTrans$
        return $findModels(newNonRefFormula, models)$
    else
        return $models$
    end if
end function
```

```
(assert
  (not
    (and
      ; pre-state
      (= x 0)
      (= y 0)
      ; post-state
      (= x_post 0)
      (= y_post 1)
      ; trace
      (= trace 2) ; index of event2 is 2
      (= trace_0 1) ; first parameter
      (= trace_1 2) ; second parameter
    )
  )
)
```

Listing 5.4: Example for a back jump clause in SMT-LIB

When having collected these models, the actual reachability check is performed. Every Action System defines initial values for all state variables. From this point, a reachability tree is constructed by using the specification. Every run of the SMT-solver in this context represents one transition in this tree (cmp. Section 5.1.1).

Figure 5.3 presents a graphical representation of this approach. First, the non-refinement formula is input to SMT solver without setting any pre- or post-state. In this way, the solver returns all possible combinations of pre-states v_i and post-states v_i' with trace length one (cmp. graphical model notation, Figure 5.1). Second, the complete state space starting at the initial state is computed (will be described more closely later in this chapter). For determining an unsafe state, the state space has to contain the pre-state v_i of a found model.

However, this initial algorithm does not scale. If the state space of the state variables are larger, the number of models to check increases rapidly. The explicit enumeration of all transitions (models) encoded by the non-refinement formula has proven to be not efficient. Nevertheless, it gave insights for developing the strategy described in Section 5.2.4.

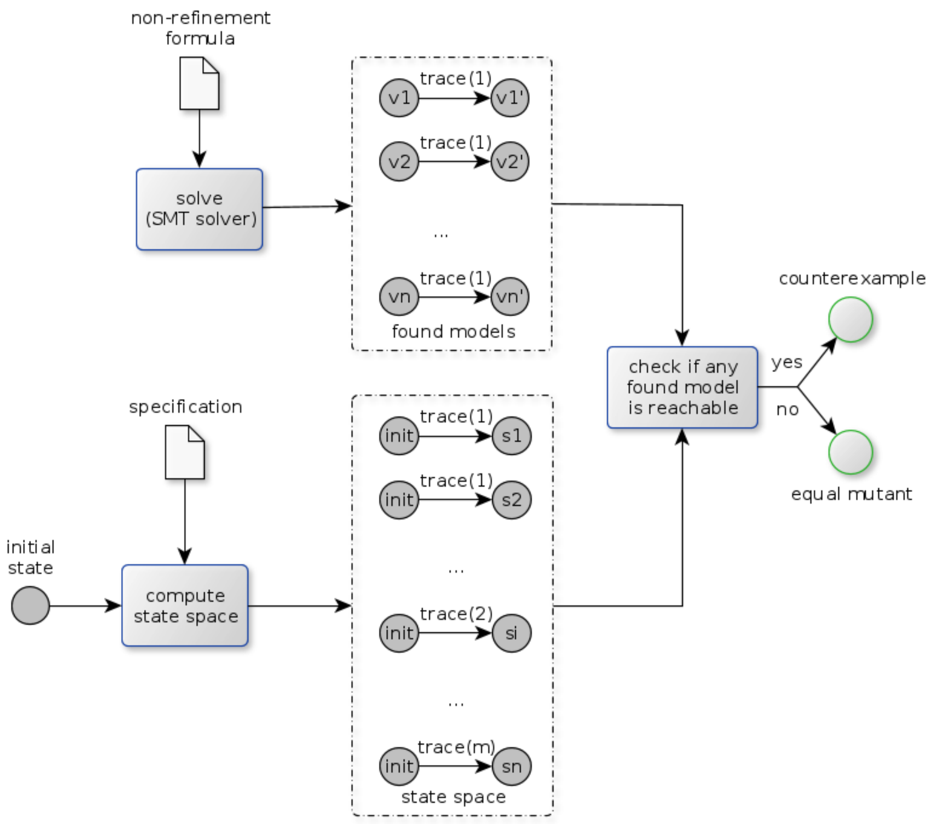


Figure 5.3: Initial strategy: compute all models, check if any pre-state is reachable

5.2.2 Reachability analysis by breadth-first search

As the first approach has shown to be impractical, an alternative had to be found. The non-refinement formula is satisfiable for every transition which can be performed in the mutated Action System but not in the original one. When setting a specific pre-state in the non-refinement formula, it will be only satisfiable if the given state is an unsafe state. Instead of enumerating the possible unsafe states and checking whether they are reachable or not, the reverse is done. The reachable target states from one source state are computed and checked if they are unsafe. Aichernig and Jöbstl also performed the reachability analysis in their tool by using this method [7]. Algorithm 9 illustrates these steps. At first, the post state of the current model (which is the initial state of the Action System in the first recursion step) is translated to SMT-LIB (Line 1). Then, this translation is appended to the non-refinement formula (Line 2). Furthermore, this string serves as an input to the solver (Line 3). If the result is satisfiable, an unsafe state is identified (Line 4). The tuple containing the unsafe state and its trace is then returned (Line 6). If the result is not satisfiable, the search for a reachable unsafe state continues. The algorithm aborts, when the maximum search depth has been reached. The length of the trace indicates the depth of the search. Hence, the check if the length of the trace of the current model is equal or greater to the maximum search depth is the valid abortion condition (Lines 8 & 9). If the search has to continue, all transitions from the post-state of the current model are calculated.

At this point, the current model can be added to the list of tested states as its safety has been shown (Line 12). If the call to *findPostStates* returns a non-empty list, the new states to explore are the ones which have not been covered yet (Lines 13 - 16). Consecutively, the list of states to explore is extended by the newly found post states (Line 17). In case there are no more states to explore, the search is finished (Line 24). This means that all transitions have been checked for non-refinement leading to a negative result. So, the mutant refines the specification. If there is at least one state to explore (Line 18), the function will be applied for one more recursion (Lines 19 - 21): the parameters $specTrans$,

Algorithm 9 Check reachable states for *unsafety*

Input: the translation of the specification, translation of the non-refinement formula, the maximum search depth, the currently checked model, the list of explored states and the list of tested states

Output: trace to unsafe state if found

```
1: function isReachableRec(specTrans, nonRefFormula, curModel, exploreStates, testedStates, maxSearchDepth)
2:     $initStateTrans := translate(currentModel.postState)$
3:     $formula := nonRefFormula \wedge stateTrans$
4:     $result := SMTSolver.solve(formula)$
5:     if $result.satisfiability = SAT$ then
6:         $unsafeModel := newActionSystemModel(result.model)$
7:         return $Some((unsafeModel.prestate, curModel.trace))$
8:     else
9:         if $curModel.trace.length \geq maxSearchDepth$ then
10:            return $None$
11:        else
12:            $testedStates.prepend(curModel)$
13:            $newStates := findPostStates(specTrans, curModel.postState, curModel.trace)$
14:            if $newStates \neq Nil$ then
15:                $newStates := newStates \setminus (testedStates \cup exploreStates)$
16:            end if
17:            $exploreStates.append(newStates)$
18:            if $exploreStates.size \geq 1$ then
19:                $nextModel := exploreStates.head$
20:                $newExploreStates := exploreStates.tail$
21:                $result := isReachableRec(specTrans, nonRefFormula, nextModel,$
                       $newExploreStates, testedStates, maxSearchDepth)$
22:                return $result$
23:            else
24:                return $None$
25:            end if
26:        end if
27:    end if
28: end function
```

nonRefFormula and *maxSearchDepth* stay the same in every recursion call. The next model, which will be checked, is the head element of the list of states to be explored. Removing this head element, this list is taken as a new value for the list of states to explore. Furthermore, the list of tested states for the next call to the present function is updated by appending the current model to it. Then, the function is applied recursively.

Figure 5.4 shows an example for Algorithm 9. On the left hand side, the situation after the first recursion loop is illustrated: From the initial state, there exist three post-states (1-3). These states represent the list of states to be checked (marked with dashed lines). In the list of found states, only the initial state is kept (marked with permanent lines). On the right side, the situation after the second recursion loop is presented: state 1 is no unsafe state, so the loop has to continue. Furthermore, the state has been added to the list of found states. As the dashed lines indicate, the list of states to be explored ranges now from state 2 to 5. The next state to be checked, will be state 2.

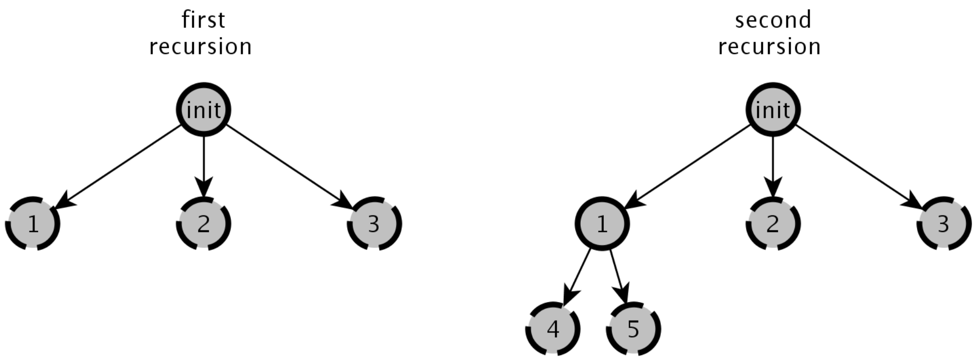


Figure 5.4: Example for the recursive execution of Algorithm 9

Find Possible Transitions From A State

For computing the reachable states from one specific state, the *as2smt* application uses the class *PostStateFinder* (package *reachability*, cmp. Section 3.3.5). It contains an algorithm to retrieve the post-states using the SMT solver. Algorithm 10 takes as parameters:

- **specWithInitStateTrans**: an SMT-LIB formula containing the (positive) specification and a reachable pre-state.
- **rootTrace**: the trace which leads from the initial state to the pre-state - called root trace.
- **curModel**: the model identified at the last recursion step pointing from the initial state of the Action System to a found post-state.
- **foundModels**: the models found until this point of execution (initially set to the empty list).

The first action of this function is to append the current model to the found models list (Line 2). This model is actually the last found transition. Then, the post state of the current model is negated and translated into SMT-LIB (Line 3). This assures that the solving process does not return the same *ActionSystemModel* again. Therefore, the translation is performed in a negated way (cmp. example in Figure 5.4). By adding this translation to the existing one (specification and initial state), a new SMT formula is created and checked by the solver (Line 3 & 4). If the result is not satisfiable, the algorithm has reached its end and returns the list of found models (Line 9 & 10). Otherwise, the solver will indicate that the formula is satisfiable. Hence, it has found another transition from the tested state. This new model is then the input for the next recursion of this function (Lines 7 & 8).

As illustrated in Figure 5.4 and described in the previous paragraph, each call to the *findPostStatesRec* function results in the list of possible transitions from this state. This may include also states, which have been checked for safety before. Therefore, Algorithm 9 subtracts these previously found states from the states to be checked (Line 15).

5.2.3 Precomputation of State Space

In Section 5.2.2, the next reachable states from a certain state are computed if this state is proven to be a safe state. In general, this seems to be the most efficient way: The states are checked in breadth-first search order. In this way, no unnecessary state exploration occurs. When looking at real use cases of this functionality, the situation differs. Usually, not just one mutant will be checked but a set of mutants. The state space does not change when checking a list of mutants. This repeated state space exploration - until the unsafe state is identified - for each mutant represents an unnecessary redundancy. This section introduces the strategy to compute the whole state space in advance to eliminate this redundancy. In fact, with this approach, the previously explained algorithm (cmp. Section 5.2.2) is split into two phases: the state space exploration and the unsafety check for the reachable states.

Algorithm 10 Find post states

Input: the translation of the Action System $asTranslation$, the root trace, the last found model $curModel$, the list of found post-states $foundModels$

Output: a list of all post-states reachable from the pre-state set in $asTranslation$

```
1: function FINDPOSTSTATESREC
2:     $foundModels.append(curModel)$
3:     $negStateTrans := translateNegState(curModel.postState)$
4:     $asTranslation := asTranslation \wedge negStateTrans$
5:     $result := SMTSolver.solve(asTranslation)$
6:     if $result.satisfiability = SAT$ then
7:         $nextModel := newActionSystemModel(result.model, rootTrace)$
8:         return $findPostStatesRec(asTranslation, rootTrace, nextModel, foundModels)$
9:     else
10:        return $foundModels$
11:    end if
12: end function
```

Phase #1 has to be executed just once for a list of mutants but for the whole state space. In the previous approach, this has to be done for each mutant separately. Phase #2 is left unchanged in both strategies. A disadvantage of this procedure is that its computation depends heavily on the size of the whole state space. This is not the case for non-equivalent mutants for the procedure presented in Section 5.2.2: There, each found state is checked instantly. So, the breadth-first search terminates when an unsafe state is identified. Nevertheless, the search is carried out on the whole state space for equivalent mutants (cmp. Figures 1.2, 1.3). Hence, if there is any equivalent mutant in a list of mutants to check, the previous approach needs more state exploration steps than the one presented in this section.

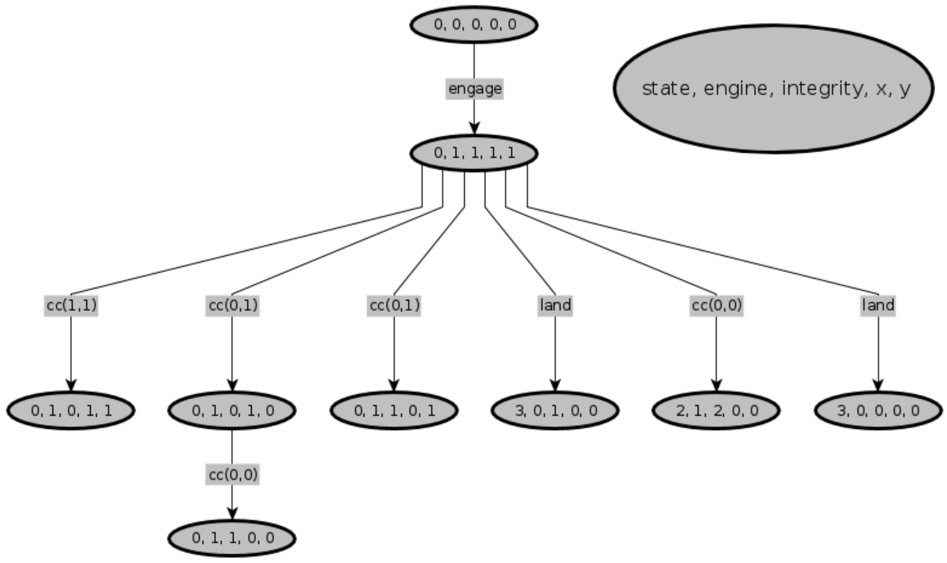


Figure 5.5: State space of the rocket steering software Action System

For example, let there be the Action System $rocket^o$ which consists of the state space shown in Figure 5.5. It consists in total of nine states. Exactly this number of states is computed when checking ten mutants. Assume that nine mutants are non-equivalent and one is equivalent. In Section 2.1.2, Definition 10 states that an Action System AS^m refines second Action System AS^m if every reachable state in specification AS^o that holds in mutant AS^m has to hold also in AS^o. For determining this fact, all reachable states have to be computed. So, the exploration of the whole state space has to be carried out for equivalent mutants. For the present example, this means that any computation of states in addition to the exploration carried out for the equivalent mutant check is redundant.

5.2.4 Reachability In One Step

This section introduces a new approach for reachability analysis. Instead of checking each reachable state one by one, it can be performed at once. After pushing the negated specification to the assertion stack, the disjunction of all states is passed on to the solver. By this action, the solver already knows all possible solutions. An advantage of this method is that the number of times the solver has to be accessed externally is linear. For a list of mutants to check, its number consists of:

1. the number of reachable states: same as in the strategy presented in Section 5.2.3
2. the number of mutants to check: one check per mutant

As the solver will not check the states by a certain order, the found state is not necessarily the one with the shortest path. Nevertheless, this is a valid solution.

```
(assert
  (or
    (and
      (= state 0)
      (= engine 0)
      (= integrity 0)
      (= x 0)
      (= y 0)
    )
    (and
      (= state 1)
      (= engine 1)
      (= integrity 0)
      (= x 1)
      (= y 1)
    )
    ; [... all other states ...]
  )
)
```

Listing 5.5: SMT-LIB state space assertion

Again, let's consider the rocket steering software example. As shown in Figure 5.5, the state space of this Action System consists of nine states. For every non-equivalent mutant, the list of reachable states has to be looped until an unsafe state is identified. For the approach presented in the previous section, this requires one to nine individual checks. In contrast to this strategy, the concatenation of all reachable states encoded in SMT-LIB by a disjunction guarantees to be just one single check per mutant. Listing 5.5 shows the translation of the disjunction of all reachable states for the rocket steering software example.

5.3 Further optimisation

Optimising the *as2smt* application in performance has been a declared goal of this thesis. In Section 4.4, the translation from Action System Prolog notation to SMT-LIB has been changed to achieve lower run times than with quantification. Section 5.1.3 has presented an approach to find the mutated action by a syntactical comparison instead of an SMT solver call. The previous section presented different strategies to determine unsafe states differ with respect to performance depending on the configuration.

This section gives two more optimisation measures and an analysis of the presented reachability computation methods. At first, the way of accessing the solver has been changed in the course of the thesis. Instead of a command line call communicating over standard input/output, the SMT solver API is used. With this access method, the second measure has been enabled: incremental solving. Finally, the reachability method analysis is performed.

5.3.1 Solver Access

As shown in Figure 3.1, the *as2smt* application uses an external SMT solver as a backend. In general, all tested solvers support the input of a file, containing the SMT-LIB assertions to be solved. Initially, this way was used to interact with the solver in the following steps:

1. The Action System is translated to the SMT-LIB language (cmp. Section 4).
2. The translation is written to a text file (with the suffix *.smt2*).
3. The solver is started by a command line call, handing over the generated file name via standard input.
4. The output of the solver is redirected from the standard output to another file.
5. The file is read and parsed.

This method led to a high number of system IO operations which is known to be quite computation time consuming. Therefore, the access of the solver was changed to JVM bindings of the various SMT solvers (cmp. Section 3.3.4).

- **MathSAT:** *MathSAT* offers a Java API.
- **SMTInterpol:** at the SMTcomp 2012, *SMTInterpol* by a developer team at the University of Freiburg, Germany has received some attention as a new participant. It is completely written in Java, and therefore can be accessed natively by the *as2smt* tool.
- **Z3:** in spring 2012, there were no JVM language bindings available from Microsoft Research directly. By this time, a solution has been introduced by Angelo Gargantini. His team used *Yices* - another SMT solver - with Java by the aid of JNA. The JNA code itself was not programmed on his own but generated by a tool called *JNAerator* [31]. It takes as input a C header file and generates JNA code for binding the C API to Java. We performed this approach by Gargantini on the Z3 C API headers. This solution is of a general nature, as it can be used with other solvers as well. The project $Scala^{Z3}$ by Phillip Suter would have also fitted the requirements. By the time of the implementation of the binding, the $Scala^{Z3}$ library did not support the current version of the SMT solver. Therefore, it has been decided to use the more general approach. Since version 4.3 of *Z3*, official Java bindings have been added which appear to be identical to the ones generated by *JNAerator*.

The mentioned SMT solver *Yices* could not be used, as it implements just SMT-LIB in Version 1. With this solver API access, the steps are reduced to the following:

1. The Action System is translated to the SMT-LIB language (cmp. Section 4).
2. The translation is sent directly as input to the solver via the API.
3. The solver returns its satisfiability and in case of SAT also a model.

In general, interactions between the JVM and the operating system can be very costly in matters of performance. File reading and writing and command line calls are two kinds of such interactions. The API approach minimises the operating system IO to a minimal level [4].

5.3.2 Incremental Solving

In a standard use case of an SMT solver, one may take an SMT-LIB formula, give it to the solver and get back the information whether the input is satisfiable or not. Additionally, a model might be obtained if it has been satisfiable. On the next call, the solver forgets about the previous set of assertions. Let there be the $rocket$ Action System as an example. Its SMT-LIB representation is listed in Appendix C. When inputting this formula to the SMT solver and asking for satisfiability, it outputs SAT and returns a reachable state of the Action System as a model. When intending to check this formula appended by the assertion `(assert (= state 2))`, the concatenation of both SMT-LIB expressions has to be input to the solver. In fact, the internal state of the solver after the $rocket$ specification assertion (and before evaluating the additional assertion) is the same. When using the API access method, this state can be saved.

Therefore, the commands *push* and *pop* are defined in *SMT-LIB* in version 2: In general, a formula may consist not of only one but many assertions. These assertions are put on a so-called *assertion stack*. The *push* command creates a backtracking point on this stack. Then, new assertions can be added and checked. To go back to the state of the *push* call, just *pop* has to be invoked.

Figure 5.6 shows how push and pop work on this stack by an example: initially, two assertions are put on the stack (State (a)). Then, a backtracking point $bt\#1$ is set by calling *push* once. Afterwards, another assertion is put on the stack and checked for satisfiability (State (b)). By applying *pop* (State (c)), the initial situation (State (a)) is restored. Finally, a new assertion (assertion #4) can be added to the current knowledge base (assertions #1 and #2; State (d)).

This technique fits perfectly the requirements of the *as2smt* tool. In general, the following assertion types have been used:

- domain assertions for the variables (Listing 4.15)
- specification assertion (Listing 4.24, used for reachable states computation, cmp. Section 5.2.2)
- negated specification assertion (Listing 5.1)
- mutant assertions (Listing 5.1)
- state assertions (Listing 5.5)
- negated state assertions (Listing 5.4)

Some assertions stay always the same in the different solver calls and can be left on the assertion stack. In comparison to the previously described solver API access, incremental solving is a real performance booster. To illustrate this, two numbers have been picked (cmp. Table 5.1). The first key figure denotes the number of assertions necessary for non-refinement checking $\mathcal{M}$ mutants (ignoring the reachability analysis). The second key figure marks the number of assertions necessary for checking $\mathcal{R}$ reachable states fulfilling a non-refinement formula.

When checking a list of mutants, the negated specification assertion (cmp. Listing 5.1) will never change among the non-refinement formulas of the checked mutants. So initially, the negated specification is pushed on the stack. Then, the first mutant is pushed, evaluated and popped to restore the assertion stack with just the negated specification on it. This continues for all mutants. With non-incremental solving, the specification assertion has to be pushed the same amount of times as the number of mutant assertions, leading to a total number of $\mathcal{M} \times 2$ assertion evaluations. Incremental solving achieves the same result by just evaluating $\mathcal{M} + 1$ assertions (one time the negated specification, $\mathcal{M}$ mutants).

[4] The file creation of the result file remains (cmp. Figure 3.1).

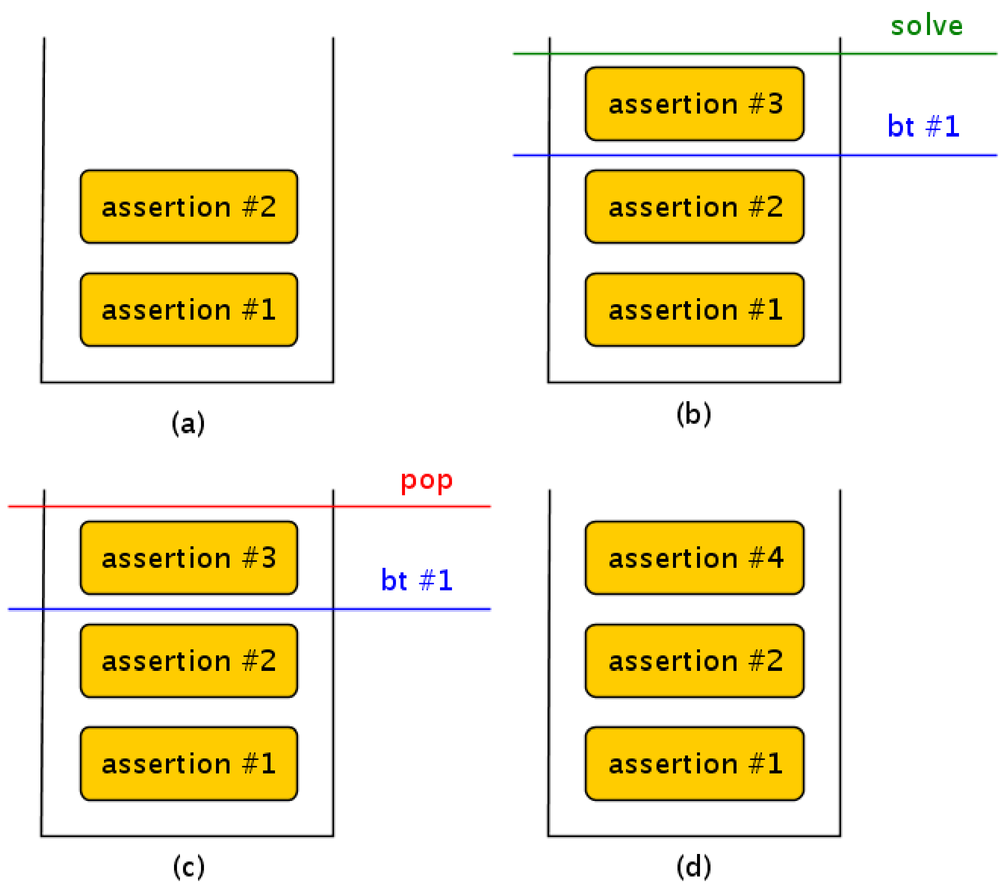


Figure 5.6: Assertion stack: usage of push and pop

number of assertions input to the solver	standard solving	incremental solving
refinement check of $\mathcal{M}$ mutants	$\mathcal{M} \times 2$	$\mathcal{M} + 1$
reachability check of $\mathcal{R}$ states (one mutant)	$\mathcal{R} \times 3$	$\mathcal{R} + 2$

Table 5.1: Standard solving vs. incremental solving: The number of assertions decreases drastically.

Also, the non-refinement formula itself does not change when checking various (reachable) states. Instead of pushing the non-refinement formula (which consists of two assertions) and the state assertion for each state, the negated specification assertion and the mutant assertion are kept on the stack and just the reachable state is pushed, evaluated and popped. In this way, the number of assertions decreases from $\mathcal{R} \times 3$ (the negated specification, mutant and state assertion) with standard solving to only $\mathcal{R} + 2$ (the negated specification and the mutant assertion stay the same) with incremental solving.

5.3.3 Reachability Computation Strategies

In Section 5.2, four different strategies have been presented to find the reachable states of an Action System. The first one was to check the reachability by possible looking up unsafe states in the list of reachable ones. This has been found very inefficient and therefore is not taken into account. In the following, we evaluate the other three strategies:

	# assertions
Strategy 1	$1 + \mathcal{M} \times (1 + \mathcal{R} + \mathcal{R}/2 \times \mathcal{T})$
Strategy 2	$2 + \mathcal{R} \times (1 + \mathcal{T} + \mathcal{M}/2)$
Strategy 3	$3 + \mathcal{M} + \mathcal{R} \times \mathcal{T} + \mathcal{R}$

Table 5.2: Reachability computation: number of average assertions per strategy

1. **Reachability analysis by breadth-first search:** the transition relation gets initialised with the initial state of the Action System. Each found state serves as initial state of the non-refinement formula. By checking its satisfiability, it is determined if it is an unsafe state (cmp Algorithm 9).
2. **Pre-computation of reachable states and check with non-refinement constraint:** this represents the same algorithm as the previous list item. However the reachable states computation is performed before non-refinement checking. For each mutant, the reachable states can be checked (cmp. Section 5.2.3).
3. **Pre-computation of reachable states and check whole state space at once:** the non-refinement check is performed by one SMT-LIB assertion instead by a number of checks (cmp. Listing 5.5).

Again, like in the Section 5.1, let there be an Action System with $\mathcal{R}$ reachable states and $\mathcal{M}$ mutants. Table 5.2 lists the three different strategies and their corresponding average number of assertions which have to be pushed to the assertion stack (cmp. Figure 5.6).

Strategy 1 needs two SMT solver instances: one for checking for unsafe states and one for computing reachable states. After pushing the negated specification and the first mutant, the initial state is pushed. Then, this can be either SAT and the algorithm finished or a new state has to be checked.

Assuming the unsafe states are *equally distributed*, it takes in average half of the state space ($\mathcal{R}/2$) per mutant to look through to find an unsafe state. For computing half of the state space, it takes the transition relation, the $\mathcal{R}/2$ pre-states and $\mathcal{R}/2$ times average transition number $\mathcal{T}$ post-states (cmp. Algorithm 10). In case, every pre-state has only one post-state, this number $\mathcal{T}$ equals 1. This represents the lower bound. For a state graph having transitions from every state to every state, $\mathcal{T}$ equals to $\mathcal{R}$ which is the maximum value for $\mathcal{T}$.

For recovering the reachable states for Strategy 1, the translation relation (e.g. the specification) and the initial state is set as a pre-state and are pushed to the second solver instance. So, the solver instance for the reachable states computation consumes in average $1 + \mathcal{M} \times (\mathcal{R}/2 + \mathcal{R}/2 \times \mathcal{T})$ assertions, and the one for unsafe state checking consumes $\mathcal{M} \times (1 + \mathcal{R}/2)$ assertions. Finally, the expression $1 + \mathcal{M} \times (\mathcal{R}/2 + \mathcal{R}/2 \times \mathcal{T}) + \mathcal{M} \times (1 + \mathcal{R}/2)$ can be transformed to $1 + \mathcal{M} \times (1 + \mathcal{R} + \mathcal{R}/2 \times \mathcal{T})$.

In Strategy 2, the reachable states are precomputed once. Its computation requires $1 + \mathcal{R} + \mathcal{R} \times \mathcal{T}$ assertions (translation relation plus $\mathcal{R}$ pre-states and $\mathcal{R} \times \mathcal{T}$ post-states). In contrast to Strategy 1, this is no longer dependent on $\mathcal{M}$. Exactly as previously described, $1+\mathcal{M}\times\mathcal{R}/2$ assertions have to be pushed to the SMT solver to find an unsafe state in case they are equally distributed. In total, $2+\mathcal{R}\times(1+\mathcal{T}+\mathcal{M}/2)$ assertions have to be input to the solver in this strategy.

The third strategy also precomputes the state space but translates it to one assertion on the assertion stack to remain there until every mutant has been checked. Again, the precomputation takes $1+\mathcal{R}+\mathcal{R}\times\mathcal{T}$ assertions. For the unsafe state check, only $1 + 1 + \mathcal{M}$ assertions are needed. In total, this sums up to $3 + \mathcal{M} + \mathcal{R} \times \mathcal{T} + \mathcal{R}$.

6 Case Studies

In the course of this thesis, the *as2smt* application has been developed. In Chapter 3, the implementation details have been discussed. In Chapter 4, the translation of an Action System to SMT-LIB has been presented. The previous chapter has given insights into the methods used to solve the model-based mutation testing problem. During development, algorithms have evolved and have been replaced by more efficient ones. This chapter documents these improvement stages where a difference in performance can be observed in our use cases.

At first, the test machine description is listed. Then, the various test inputs are presented. Furthermore, the development stages are discussed. Consecutively, the different configurations for the experiments are outlined. Finally, the results are documented.

6.1 Test Machine

The test machine consists of

- an Intel Core i7-2640M CPU running at 2.80GHz
- 8 GB DDR3 RAM
- an Intel Solid-State-Disk
- Linux operating system, kernel 3.16.6, OpenSuse 13.2, 64-bit
- Java Runtime Environment 1.8 (OpenJDK)
- Scala 2.10.3

For the solvers, the following versions have been tested:

- MathSAT [47] in Version 5.2.1
- SMTInterpol [32] in Version 2.1-3
- Z3 [76] in Version 4.3.2

6.2 Test Input

As described in Section 3.2, the implemented tool takes as input Action Systems represented in Prolog. The illustrated examples in the case studies have also been used by the *as2csp* [7] application. On its basis, Aichernig et al. compared the two tools with respect to performance [10].

6.2.1 Car Alarm System

The first case study contains an Action System modeling a simplified car alarm system. It origins from Ford's automotive demonstrator within the MOGENTES project [4]. There, the following requirements are listed to be fulfilled:

1. **Arming:** when the all doors are closed, the system arms itself after 20 seconds in this state.
2. **Alarm:** on any attempt to open a door in the armed state without authorisation (for instance a key), the alarm sound will be turned on for 30 seconds. Additionally, the hazard flasher will be enabled for five minutes.
3. **Deactivation:** deactivation of the anti-theft alarm system occurs when the car is unlocked from outside. This can be enforced at any time after the alarm has been set.

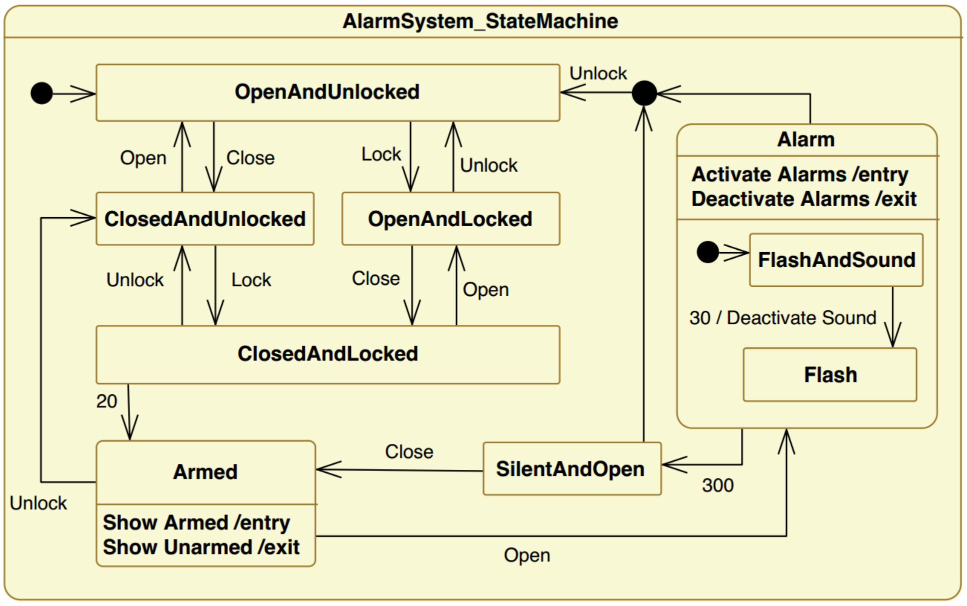


Figure 6.1: Car alarm system model (Ford's automotive demonstrator, MOGENTES project [4])

Aichernig et al. present a model in form of a UML state machine which implements these requirements [4]. This state machine is illustrated in Figure 6.1. There exist two transitions from the initial state *OpenAndUnlocked* to *ClosedAndLocked* via *ClosedAndUnlocked* and *OpenAndLocked* depending on the sequence of the actions *close* and *lock*. After 20 seconds in the *ClosedAndLocked* state, the alarm is turned on (state *Armed*). This implements the first requirement. By opening a door, the acoustic and optical alarm is raised in the *FlashAndSound* state. This has been modeled as a sub-state of the *Alarm* state. After 30 seconds the alarm sound is turned off but the hazard lights continue (sub-state *Flash*) to fulfil the second requirement. It is not specified what happens after the sound lights are disabled. If an unauthorised access happens (state *SilentAndOpen*) afterwards and the door gets closed, Aichernig et al. implemented a transition to the *Armed* state. The last requirement is simply adding the transition from the *Armed* state to *OpenAndUnlocked* when unlocking the vehicle.

Furthermore, Aichernig and Jöbstl modeled this state machine as an Action System [8]. Additionally, they manually constructed mutants by applying three different mutation operators:

- **Guards:** all guards in the entire Action System are changed to be true all the time resulting in 34 mutants.
- **Equality swapping:** each equality and each inequality sign gets swapped to its opposite. This leads to 56 mutants.
- **Incrementation:** every reference to a number is substituted by the number plus one. For Mutations which would have resulted in a domain violation this way, the lowest number in the domain has been used. Note that also the states are modeled as numbers. 116 mutants resulted from this mutation operation.

Furthermore, the equivalent mutant - actually the same Action System as the original - is also included in the set of mutants. So in total, this sums up to 207 mutants. This model is used in the case studies in four different versions: *CAS_1, CAS_10, CAS_100, CAS_1000*. *CAS* is short for *Car Alarm System*. The number i in *CAS_i* indicates the factor applied to the parameter values. For example, the action *after* (modeling a time delay) reacts on three different values: 20, 30 and 270. In the CAS_10, these values are set to 200, 300 and 2700. The reason for checking this is to test the behaviour of the solvers on domain extensions.

6.2.2 Car Alarm System with PIN Input

Furthermore, two new Action Systems are defined by parametrising the actions *Lock* and *Unlock* from the original CAS. If the parameter (PIN) is set to a predefined value, the Action System behaves as the original one. Otherwise, the result is the same as opening the car in the *Armed* state. The flash lights will turn on and the alarm sound will start to go off. The two versions differ in the domains of the parameters of the mentioned two actions:

- In the *CAS_BOOLPIN* model, the parameter is a simple Boolean. It may indicate the presence of a key-like token.
- The *CAS_PIN* Action System takes for the *Lock* and *Unlock* actions an integer in the range between 0 and 999 like in a traditional combination lock.

The application of the previously described three mutation operations results in 246 mutants, also including the equivalent mutant.

6.2.3 Triangle

Finally, an Action System that models a simple function, serves as a case study. This function checks if a triangle with given side lengths is equilateral, isosceles, scalene or actually no triangle. The Action System consists of three actions: *Input*, *Calculate* and *Output*.

- **Input** takes the three integers between zero and three and sets the state variables a, b and c to these values. These are the side lengths of the triangle.
- **Calculate** determines which case applies to the side lengths (state variables).
- **Output** takes the triangle kind as an input and sets the respective state variable.

In contrast to the previously introduced Action Systems, the state space size is directly dependent on the size of the parameter domain. In the different CAS versions, the domain of a parameter has been changed too, but the state variables have been fixed. Thethe range zero and three Action System limits this directly by the domain of the parameters. Therefore, another version of it extends the parameter domain from the range zero and three to an integer between zero and 30. As the range increases by a factor of ten, this version of the triangle test input is called *TRIANGLE_10* The state space is enlarged this way by a factor of 458. For test purposes, two mutants have been created manually.

Table 6.1 lists all the Action Systems used in the case studies. It gives an overview of the major differences between the test inputs. Apart from the parameter ranges, all standard CAS versions have the same properties. In contrast to the two *TRIANGLE* versions, parameter range increase does not affect the number of reachable states. Also, the extended CAS test inputs (including the *PIN*) have also 21 reachable states. But the number of mutants increases due to the higher number of parametrised actions.

6.3 Stages of Development

This section gives an overview of the different development stages and improvements applied to the *as2smt* application.

6.3.1 Quantifier Elimination

After our first tests using SMT formulas containing quantifiers, performance issues became obvious. Quantifiers are needed to translate sequential composition correctly (cmp. Section 4.3.4) when using

action system	#actions (parametrised)	#state variables	#mutants	#reachable states
CAS_1	11 (1)	6	207	21
CAS_10	11 (1)	6	207	21
CAS_100	11 (1)	6	207	21
CAS_1000	11 (1)	6	207	21
CAS_BOOLPIN	11 (3)	6	246	21
CAS_PIN	11 (3)	6	246	21
TRIANGLE	3 (2)	9	2	65
TRIANGLE_10	3 (2)	9	2	29792

Table 6.1: Action Systems used as test input for the *as2smt* application

	Original CAS (*CAS_1*)				Extended CAS (*CAS_BOOLPIN*)			
	io	translate	solve	total	io	translate	solve	total
direct translation	2.67	1.48	1670.52	**1674.75**	5.01	2.61	2852.74	**2860.36**
no quantifiers (incorrect)	2.69	1.45	11.90	**16.04**	3.58	1.99	19.74	**25.31**
no quantifiers (correct)	3.01	0.47	0.89	**4.37**	4.02	0.63	1.14	**5.79**

Table 6.2: Performance of non-refinement formula check of all mutants per translation type (in seconds)

negation. These test results did only cover the non-refinement check and not the reachability analysis as this had not been implemented by the time these experiments had been run. As a comparison, the solver had been run with the same output formula but without the needed quantifiers. In this way, the computation is incorrect. But it has given an indication for the performance gap when using quantifiers. Therefore, (existential) quantifiers have been eliminated by applying the one-point rule. This has been described in Section 4.4.

For comparability, the module containing the translation with quantification has been restored and placed as a copy into the package `oldapproach`. In Table 6.2, the results for this former translation method are presented. The *as2smt* application accesses the back-end $Z3$ via API (cmp. Section 6.3.2). The time consumption of the *as2smt* tool for three translation approaches are listed: the first one contains existential quantifiers as it has been restored from the old source code. Then, the second one skips the generation of the quantifiers to show the performance issue. Finally, the third approach applies the one-point rule to remove the quantification. Section 4.4.1 describes the quantifier elimination in detail. All three translation types have been applied to the original CAS (version *CAS_1*) and the extended CAS (version *CAS_BOOLPIN* test inputs. Table 6.2 shows four different times. The *total* run time is split into three different parts:

- *io:* time needed to read (input) files, parse and write (output) files
- *translate:* time needed to translate the Action System from an internal representation to SMT-LIB
- *solve:* solving time of the SMT solver

Apart from the enormous gap in the $solve$ time between the first and the two last approaches, the $translate$ time also differs between the two correct approaches. The reason lies in the difference of complexity of the translation. When using the *direct* translation including quantifiers, more variables and constraints have to be generated as when applying the one-point rule and *executing* the code symbolically. For instance, every time a variable is re-assigned in an action, a new variable (including also its bounds) has to be created in SMT-LIB (cmp. Section 4.3.2). For the chosen test inputs, using quantifier elimination results in the decrease of the run time by approximately 99.8%.

	Original CAS (*CAS_1*)				Extended CAS (*CAS_BOOLPIN*)			
	io	translate	solve	total	io	translate	solve	total
command line call	4.24	0.48	2.74	**7.46**	5.58	0.66	3.37	**9.61**
API access	3.01	0.47	0.89	**4.37**	4.02	0.63	1.14	**5.79**

Table 6.3: Timing behaviour of non-refinement formula check of all mutants per access type (in seconds)

	Original CAS (*CAS_1*)				Extended CAS (*CAS_BOOLPIN*)			
	io	translate	solve	total	io	translate	solve	total
non-incremental	3.01	0.47	0.89	**4.37**	4.02	0.63	1.14	**5.79**
incremental	1.66	0.30	0.62	**2.58**	2.20	0.40	0.80	**3.40**

Table 6.4: Timing behaviour of the non-refinement formula check of all mutants per solving type (in seconds)

6.3.2 Solver Access

In Section 5.3.1, the two different types to access the SMT solver have been discussed: by command line calls or by API access. As the algorithms for reachability analysis have been implemented after the change of access type to the API access, the experiments in this section cover only the non-refinement check without any reachability analysis. Table 6.3 shows the run times of the *as2smt* application by the two access types for two examples (*CAS_1* and *CAS_BOOLPIN*) using Z3 as a back-end. The results indicate that the way of accessing the solver improves the performance significantly.

For the listed examples, the diminished file I/O activity represents a crucial factor. Writing input from and reading results to files becomes unnecessary (cmp. Section 5.3.1). Even more significantly, the solver time is decreased drastically as the solver has not to read input and write output files but is being fed with SMT-LIB strings directly. In total, the performance boost can be quantified by a 40% descreased run time.

6.3.3 Incremental solving

In Section 5.3.2, incremental solving has been described. This SMT-LIB feature specifies backtracking points which can be set by `push` and recovered by `pop` [24]. With command line calls, incremental solving can be used but assertion stack modifications are only possible if they happen in the scope of one input file. With API access, incremental solving becomes more useful.

Similar to Table 6.3, Table 6.4 shows the run times of the *as2smt* application for two examples (*CAS_1* and *CAS_BOOLPIN*) using Z3 as a back-end. But instead of comparing access types, solving types are compared: non-incremental and incremental solving. All previous mentioned improvements have been also applied. Here, the results for non-incremental solving are exactly the ones listed in Table 6.3 for the API access type.

Like in the previous section, the numbers indicate a performance improvement. Again, file I/O times have been reduced (this time by about 45%). Also, translation time has been minimised by 35 %. Additionally, the solver computation time is lowered by 30 %. For the test inputs, incremental solving shows a reduction of the run time by around 40 % in total. For reachability, this can not be confirmed by empirical results, as the reachability analysis algorithms require incremental solving. Nevertheless, Section 5.3.2 gives strong arguments that non-incremental solving performs much worse for the reachability analysis.

6.4 Configuration of Experiments

The improvements listed in the previous section are undoubtedly speeding up the performance. So, quantifier elimination, solver access via API and incremental solving are used in all experiments. Their performance benefit has been shown before. The influence of the choice of the reachability computation method has been discussed only in an analytical way (cmp. Section 5.3.3), not in an experimental one. Furthermore, the selection of the test input and the solver may also be significant. Hence, the *as2smt* application defines three options to be selected (cmp. Section 3.4.2):

- **SMT solver:** *MathSAT*, *SMTInterpol* or *Z3* can be chosen.
- **Solving method:** the unsafe state is determined by
  - finding and checking each reachable state (Strategy 1).
  - precomputing all reachable states and then checking them for each mutant. The check occurs by their breadth-first search ordering (Strategy 2).
  - precomputing all reachable states, translate them to one assertion and check each mutant. With this method, it is not assured to get the unsafe state with the shortest path (Strategy 3).
- **Action System:** the example input can be selected from eight different Action Systems with already declared mutations. In general, any Action System in Prolog notation (cmp. Appendix A, cmp.) can be input. But only the Action Systems listed under Section 6.2 have been selected as case studies.

Three solvers, three reachability solving methods and eight different example Action Systems (specifications and mutants) result in 72 test configurations.

6.5 Results

This section presents the performance results of the *as2smt* application with the test configurations. The different reachability solving methods (cmp. Section 5.3.3) are applied to the chosen test input (cmp. Section 6.2). Each combination of test input and reachability solving method can be executed with one of three SMT solvers (cmp. Section 6.4).

6.5.1 Strategy 1: Reachability Analysis by Breadth-First Search

At first, the three solvers are compared by their performance on the four different variable domain ranges for the original CAS. Table 6.5 shows the results for the different CAS versions (CAS_1, CAS_10, CAS_100 and CAS_1000). Per run, the time needed for solver interactions and the total run time have been tracked. Furthermore, the average execution time per mutant and the longest run for one mutant are listed.

In general, it can be observed that the execution time of the *as2smt* application depends heavily on the time consumption of the solver. In numbers, this variance can range up to 90 % of the total run time. In fact, the period of time the tool takes ignoring the interaction times with the SMT solver varies between 1.57 and 2.69 seconds.

Focusing on solver performance, *Z3* and *SMTInterpol* outperform *MathSAT* by far (approximately by a factor of five). In fact, that *SMTInterpol* implemented in Java performs so much better than *MathSAT* implemented in C seems to be remarkable. Taken into account the different implementation languages, performance gap between *SMTInterpol* and *Z3* seems to be rather small.

Another noticeable fact is that the computation times vary from run to run. When collecting the results, each configuration has been executed three times and the median value has been recorded to

in seconds		MathSAT		SMTInterpol		Z3	
		solver	total	solver	total	solver	total
CAS_1	$\sum$	23.84	**26.04**	5.66	**8.46**	4.76	**6.36**
	$\varnothing$	0.12	0.13	0.03	0.04	0.02	0.03
	max.	0.40	0.43	0.13	0.14	0.09	0.10
CAS_10	$\sum$	23.80	**26.16**	4.90	**7.25**	5.31	**7.06**
	$\varnothing$	0.11	0.13	0.02	0.04	0.03	0.03
	max.	0.39	0.40	0.12	0.14	0.09	0.11
CAS_100	$\sum$	24.12	**26.24**	5.12	**7.60**	4.71	**6.28**
	$\varnothing$	0.12	0.13	0.02	0.04	0.02	0.03
	max.	0.48	0.50	0.13	0.15	0.11	0.12
CAS_1000	$\sum$	22.95	**25.16**	5.00	**7.69**	4.74	**6.35**
	$\varnothing$	0.11	0.12	0.02	0.04	0.02	0.03
	max.	0.34	0.37	0.13	0.16	0.07	0.08

Table 6.5: Strategy 1: performance by SMT solver on CAS versions

in seconds		MathSAT		SMTInterpol		Z3	
		solver	total	solver	total	solver	total
CAS_BOOLPIN	$\sum$	32.17	**35.31**	6.00	**9.12**	6.15	**8.27**
	$\varnothing$	0.13	0.14	0.02	0.04	0.02	0.03
	max.	0.48	0.50	0.17	0.18	0.09	0.10
CAS_PIN	$\sum$	32.42	**35.70**	5.99	**9.17**	6.09	**8.21**
	$\varnothing$	0.13	0.15	0.02	0.04	0.02	0.03
	max.	0.46	0.49	0.17	0.19	0.08	0.09

Table 6.6: Strategy 1: performance by SMT solver on CAS version with PIN extension

minimise the variance. For *Z3* and *SMTInterpol*, the time differences between the runs last up to half a second, for *MathSAT* up to two seconds. When taking this variance into account, the computation times for the different CAS versions do not differ. With a much wider domain - respectively more variables - this may change.

For *CAS_BOOLPIN* and *CAS_PIN*, the average run time for refinement checking a mutant with Strategy 1 is similar to the numbers for the CAS versions (cmp. Table 6.5). In comparison to each other, the test inputs differ the same way as the different versions of the original CAS: the domain of one state variable is changed. By observing the same effect again, it can be determined that this kind of alteration has no major effect on the performance of the SMT solver and so on the *as2smt* application run time.

The *triangle* test input is distinct from the others by the size of its state space. The simple *triangle* example performs quite well. Especially with Z3 as a back-end, the *as2smt* application is very fast in recovering the correct result. With the *TRIANGLE_10* configuration, it can be observed that widened state space affects run time in a severe manner. Most interesting here is the performance of the *SMTInterpol* solver. It beats *Z3* and *MathSAT* by approximately a factor of seven or ten respectively.

6.5.2 Strategy 2: Precomputation of Reachable States and Check Each on Mutant

Strategy 2 performs the state space exploration before checking any mutant instead for each mutant until an unsafe state is reached. This action decreases the times, the solver has to be accessed. In Table 5.2, the *assertion estimation* formulas are listed for each strategy. These estimations are based on the assumption that when checking the mutants, the found unsafe states are equally spread in the state space. Table

in seconds		MathSAT		SMTInterpol		Z3	
		solver	total	solver	total	solver	total
TRIANGLE	$\sum$	1.57	**1.83**	0.88	**1.15**	0.31	**0.48**
	$\varnothing$	0.79	0.92	0.44	0.57	0.15	0.24
	max.	0.87	1.01	0.51	0.64	0.22	0.31
TRIANGLE_10	$\sum$	722.86	**728.19**	71.04	**80.47**	523.35	**526.36**
	$\varnothing$	361.43	364.09	35.52	40.24	261.67	263.18
	max.	387.58	390.47	36.45	40.45	263.56	265.07

Table 6.7: Strategy 1: performance by SMT solver on the triangle model

	Strategy 1		Strategy 2		Strategy 3	
	estimated	actual	estimated	actual	estimated	actual
CAS	7808	7391	2219	2077	264	262
CAS with PIN	9982	8554	2645	2571	309	307
TRIANGLE	261	126	262	243	200	197
TRIANGLE_10	119169	59749	119168	89375	89381	89375

Table 6.8: Estimated and actual numbers of asserts needed per strategy

		MathSAT		SMTInterpol		Z3	
		solver	total	solver	total	solver	total
CAS_1	$\sum$	2.90	**5.34**	1.80	**4.62**	1.23	**3.18**
	$\varnothing$	0.01	0.03	0.01	0.02	0.01	0.02
	max.	0.03	0.07	0.03	0.05	0.01	0.03
CAS_10	$\sum$	2.90	**5.08**	1.79	**4.63**	1.32	**3.26**
	$\varnothing$	0.01	0.02	0.01	0.02	0.01	0.02
	max.	0.04	0.07	0.03	0.04	0.01	0.03
CAS_100	$\sum$	2.99	**5.24**	1.91	**4.53**	1.74	**4.12**
	$\varnothing$	0.01	0.03	0.01	0.02	0.01	0.02
	max.	0.06	0.08	0.06	0.08	0.04	0.05
CAS_1000	$\sum$	3.12	**5.48**	1.93	**4.63**	1.29	**3.19**
	$\varnothing$	0.01	0.03	0.01	0.02	0.01	0.02
	max.	0.03	0.05	0.03	0.04	0.01	0.02
CAS_BOOLPIN	$\sum$	3.90	**6.73**	2.18	**5.75**	1.59	**4.16**
	$\varnothing$	0.01	0.03	0.01	0.02	0.01	0.02
	max.	0.03	0.06	0.03	0.06	0.01	0.03
CAS_PIN	$\sum$	3.94	**7.08**	2.19	**5.96**	1.54	**4.05**
	$\varnothing$	0.01	0.03	0.01	0.02	0.01	0.02
	max.	0.03	0.05	0.05	0.06	0.01	0.03
TRIANGLE	$\sum$	1.40	**1.67**	0.61	**0.82**	0.45	**0.61**
	$\varnothing$	0.36	0.83	0.16	0.41	0.12	0.31
	max.	0.67	0.84	0.28	0.41	0.21	0.31
TRIANGLE_10	$\sum$	659.46	**699.79**	70.02	**105.10**	318.38	**353.13**
	$\varnothing$	164.96	349.90	17.57	52.55	79.77	176.56
	max.	329.54	349.98	34.88	52.61	158.85	176.59

Table 6.9: Strategy 2: performance by test input and SMT solver (in seconds)

in seconds	Strategy 1		Strategy 2		Strategy 3	
	2 mutants	3 mutants	2 mutants	3 mutants	2 mutants	3 mutants
solver	71.04	110.50	70.02	64.29	74.76	63.68
total	80.47	126.69	105.10	102.03	111.97	100.37

Table 6.10: Comparison of performance of *SMTInterpol* on *TRIANGLE_10* test input

6.8 shows the estimated and the actual numbers of assertions per strategy. For the the CAS examples (original and extended), the numbers match up very closely. Nearly half of the state space had to be uncovered per mutant.

Table 6.9 shows the results for all the test input. Interestingly, *MathSAT* reaches a much better performance than with Strategy 1. In fact, its timing behaviour gets very close to the one of *SMTInterpol*. With *Z3* as a back-end, the *as2smt* tool computes the unsafe states much faster than with the other two solvers.

In respect to the previous strategy, a general performance increase can be observed. *MatSAT* even boosts its average time consumption per mutant for the CAS test inputs from 0.12 seconds down to 0.01 seconds. Also by using *SMTInterpol* and *Z3*, a definite run time decrease by about 50% to 75% can be measured for the various CAS test inputs.

With the simple triangle use case, there seems to be no difference (as the state space size is nearly the same as in the CAS examples). When widening the state space, the complete computation of all reachable states may take more time than with the approach described in the previous strategy. This is the case if the total number of states is higher than the sum over states that have to be checked for each mutant. For the *TRIANGLE_10* test input, the three solvers behave differently. While *SMTInterpol* shows an increase in total run time, a run time decrease can be observed for *MathSAT* and *Z3*. In fact, the *solver* time of *SMTInterpol* is nearly the same for both strategies (around 70 seconds). But the computational overhead of the *as2smt* increases with Strategy 2. For just one more mutant, also *SMTInterpol* shows a decrease in total run time with Strategy 2 (cmp. Table 6.10).

6.5.3 Strategy 3: Precomputation of Reachable States, List and Check Each Mutant

As mentioned, this method does not calculate the unsafe state with the shortest trace but just the first one that the SMT solver finds. If a shortest trace is not regarded as a requirement of the application, this method is comparable to the others. In Table 6.8, the estimated and the actual number of assertions are listed for all strategies. There, Strategy 3 needs only 12 % to 13% of the assertions in comparison to Strategy 2 for the CAS input versions.

As seen previously, the triangle test input behaves differently as the preceding state computation creates an unnecessary overhead. Still, the application of Strategy 3 results in an improvement regarding the run time when comparing to the same test input applied to Strategy 2.

Table 6.11 presents the results for the tested solvers and for all test inputs. In general, it decreases the run time of all solvers by about 25% for the (original and *PIN/BOOLPIN*) CAS versions.

In total, the *as2smt* application shows the best performance for *Z3* and Strategy 3 on the CAS and the *TRIANGLE* test inputs. For *TRIANGLE_10*, *SMTInterpol* outperforms the other solvers by far. In comparison to Strategies 1 & 2, Strategy 3 is the slowest among all for this configuration. But again, this changes when adding one mutant (cmp. Table 6.10).

		MathSAT		SMTInterpol		Z3	
		solver	total	solver	total	solver	total
CAS_1	$\sum$	1.46	**3.90**	0.82	**3.79**	0.46	**2.44**
	$\varnothing$	0.01	0.02	< 0.01	0.02	< 0.01	0.01
	max.	0.01	0.04	0.01	0.04	< 0.01	0.02
CAS_10	$\sum$	1.43	**4.00**	0.84	**3.76**	0.49	**2.54**
	$\varnothing$	0.01	0.02	< 0.01	0.02	< 0.01	0.01
	max.	0.03	0.05	0.01	0.04	< 0.01	0.02
CAS_100	$\sum$	1.36	**3.79**	0.81	**3.70**	0.51	**2.58**
	$\varnothing$	< 0.01	0.02	< 0.01	0.02	< 0.01	0.01
	max.	0.01	0.04	0.01	0.03	< 0.01	0.02
CAS_1000	$\sum$	1.44	**3.85**	0.81	**3.46**	0.53	**2.58**
	$\varnothing$	0.01	0.02	< 0.01	0.02	< 0.01	0.01
	max.	0.01	0.04	0.01	0.03	< 0.01	0.02
CAS_BOOLPIN	$\sum$	1.67	**4.72**	1.03	**4.99**	0.58	**3.19**
	$\varnothing$	< 0.01	0.02	< 0.01	0.02	< 0.01	0.01
	max.	0.01	0.05	0.01	0.04	< 0.01	0.02
CAS_PIN	$\sum$	1.78	**4.83**	1.06	**4.83**	0.60	**3.24**
	$\varnothing$	0.01	0.02	< 0.01	0.02	< 0.01	0.01
	max.	0.01	0.04	0.02	0.03	< 0.01	0.02
TRIANGLE	$\sum$	1.25	**1.50**	0.81	**1.04**	0.39	**0.55**
	$\varnothing$	0.32	0.75	0.21	0.52	0.10	0.28
	max.	0.61	0.75	0.39	0.52	0.19	0.28
TRIANGLE_10	$\sum$	716.59	**758.40**	74.76	**111.97**	374.32	**412.181**
	$\varnothing$	179.18	379.20	18.87	55.98	93.61	206.09
	max.	358.24	379.21	37.02	56.00	187.10	206.13

Table 6.11: Strategy 3: performance by test input and SMT solver (in seconds)

6.6 Conclusion of Results

In this chapter, the presented improvements have been validated. Three different SMT solvers have been used as back ends and different strategies for finding unsafe states have been tested on various test inputs. The following findings can be listed:

- Quantification with SMT solvers has shown to be far slower than non-quantification and *manual application* (done by the *as2smt* tool instead) of the one-point rule (cmp. Section 6.3.1).
- API access of SMT solvers has to be favoured over command line calls when using a solver as a back-end (cmp. Section 6.3.2).
- Incremental solving decreases computation time as the solver has not to parse the same formulas multiple times (cmp. Section 6.3.3).
- SMT solver performances may show major differences. For the tested solvers in combination with the test input, *Z3* by *Microsoft Research* showed the best performances. Only for the last *TRIANGLE_10* test input, *SMTInterpol* is faster than *Z3*.
- Approaches with reduced interaction with the back-end SMT solver should be favoured over more intense interacting methods (cmp. Section 6.5.2 and 6.5.3).
- Computation of the complete state space may not trade-off if the number of checked mutants is too low (cmp. result for *TRIANGLE_10* test input with Strategies 1 & 2).

7 Related Work

7.1 Related Work on Action Systems

The used modeling language in this thesis has been *Action Systems*. Initially, Back and Kurkio-Suonio defined this formalism in 1983 [17]. Previously, Back has introduced his notation of *refinement* in terms of weakest precondition in [15] which he later applied to Action Systems [18].

Back and Sere extend Action Systems to include modularization features in [19]. They argue, that the monolithic, large systems are impractical for refinement. With this extension, a module can be defined which may export/import variables, set access restrictions to its own procedures and be able to call other module's procedures. Back and Sere present an adaptation of the Oberon programming language, called Action-Oberon, for expressing such modular Action Systems. In terms of semantics, reduction rules are defined for all modularization constructs to provide refinement also for Action-Oberon. Moreover, Back et al. encapsulate also data and actions in objects in the Action-Oberon language [16].

In [26], Bonsangue et al. extend Action Systems with object-orientation. The introduced OO-Action Systems can be mapped directly to ordinary Action Systems. In this way, the (weakest precondition) semantics is preserved. Additionally, they present refinement rules for expressing inheritance and reuse of code and proofs.

Krenn et al. present a mapping of UML state diagrams to object-oriented Action Systems [59]. This translation maps concurrency to non-deterministic choice, enables support of triggered transitions and sets event-processing to be in-order and loss-less. Furthermore, this mapping is used to translate UML further to labelled-transition systems (LTS).

7.2 Related Work on Model-Based Mutation Testing

In 1998, Ammann et al. use a model-checker for generating test cases. The generated mutants and the original model are input to the model-checker which validates *if the mutant and the original program produce different outputs*. If they produce non-equal outputs a test case is generated from a counter-example.

In 2006, Aichernig and Salas introduce their *mutation testing approach for automatic test case generation from model-based specifications* [12]. There, they apply mutation operators to the pre- and post-conditions of models described in Object Constraint Language (OCL). Tretmans' Input-Output conformance (ioco) of LTS [82] serves as conformance relation.

Krenn and Aichernig show how to generate test cases by using contract mutation in Spec# [58]. This extension for the programming language C# allows - similar to the OCL - the definition of pre- and post-conditions. An interesting side note here is, that Spec# relies on the SMT solver Z3 just like the implemented *as2smt* tool.

Aichernig et al. apply model-based mutation testing also to an extension of Action Systems called *Qualitative Action Systems* [2]. Their use enables the modeling of continuous behaviour in a discrete domain [5]. Again, LTS semantics and *ioco* as conformance relation have been used.

In 2010, Aichernig and Jöbstl present the model-based mutation testing approach as it has been used in this thesis: conformance checking of Action System models by refinement [7]. In difference to *as2smt*, their *as2csp* tool uses *SICStus Prolog*. On the one hand, Prolog represents the implementation language. On the other hand, the Prolog engine acts as constraint satisfaction problem (CSP) solver. In the course of this thesis, *as2smt* and *as2csp* have evolved. Aichernig et al. compare both tools by their performance [10]. In fact, both tools achieve on the chosen test input similar results.

In [11], Aichernig et al. present the application of model-based mutation testing to real-time systems.

As models, timed automated are used. Their conformance is checked by an extended version of *ioco* called timed ioco [60]. The developed tool is based on bounded model-checking techniques. Again, an SMT solver forms the back end of the tool.

Brillout et al. use a model checker to generate test cases from Simulink models and its mutants [28]. Their COVER tool checks for equivalence by bounded model checking. For being able to deal with a decidable problem, a finite range for input and output values is assumed. Equivalence checking is limited to a certain search depth k (just like the refinement check of the presented tool in this thesis), denoted as *k-equivalence*.

8 Concluding Remarks

8.1 Summary

In this thesis, we have shown how SMT solvers can be used for model-based mutation testing (MBMT). MBMT is a novel testing technique and enables the generation of test cases on a model level. Initially, a set of fault-injected versions of an original program model is generated by an external program. This program - a mutation generator - has not been in the scope of this thesis.

The main contribution of this thesis is the implementation of the *as2smt* tool. It applies MBMT on *Action System* models. This formalism enables the modeling of reactive systems and features also non-determinism. Originally, Action Systems are interpreted by weakest precondition semantics (by Back and Wright). In this thesis, we applied the predicative semantics defined by Aichernig et al.

An SMT solver represents the computational back end of the tool. SMT solvers have evolved and gained in popularity the last years. With the definition of the standardised SMT-LIB language, a universal input language for SMT solvers exists. Due to this interchangeability, any SMT solver supporting SMT-LIB can be used. Bindings for the solvers *MathSAT*, *SMTInterpol* and *Z3* have been integrated into the tool.

The *as2smt* application implements a compiler which translates from the modeling formalism Action Systems (in a special notation) to the standard SMT solver input language SMT-LIB. This has been only possible by the non-standard, predicative semantics given to Action Systems.

In MBMT, the original model and one mutant are *compared* with respect to their behaviour. Equality of programs is an undecidable problem. Even with bounded exploration depth it has still exponential runtime. Especially, hard is the possible non-determinism in Action Systems.

The state-based refinement relation has been chosen to effectively check the conformance of action systems with SMT solvers. Refinement had been already defined for Action Systems. It states that any reachable state change which is valid for the mutated model, has to hold also in the original model. For test case generation, non-refinement is checked.

A non-conforming behaviour of the mutant (in regards to the original) has to fulfill two relations: non-refinement relation and the reachability relation. The non-refinement relation has been entitled through out this thesis by non-refinement formula which refers to the generated SMT-LIB formula. This non-refinement formula consists of the translation of the negation of the original model conjuncted by the checked, mutated model. For checking reachability, there has to exist a transition from the initial state of the Action System to the pre-state of the checked transition. This requires a state space exploration of the model.

We evolved the tool's translation process from a direct translation of the predicative semantics to a symbolic execution of each execution branch of the model. For the reachability analysis, three different algorithms have been implemented and evaluated.

8.2 Contribution of this Thesis

The main contribution of this thesis is the implementation of the presented test case generation technique. The *as2smt* application enables test case generation of models (with given mutants) in form of Action Systems with an SMT-solver as computational back end.

Two different ways of translating Action System models to the standardised SMT solver input language SMT-LIB have been evaluated. The direct translation of the predicative semantics to SMT-LIB has been shown to be valid but inefficient. The translation of sequential composition in Action Systems requires quantification. In contrast to other SMT-LIB language constructs, quantifiers trigger an increase in solving time. Additionally, just a few solvers support quantification at all. The second translation approach applies the one-point rule to eliminate the need of quantification in SMT-LIB. In fact, the translation process has been adapted to symbolically execute assignments.

For selecting the mutated action of the Action System, two approaches have been considered. First, the non-refinement formula has been input to the SMT solver which states a difference in behaviour in the mutated action. Second, the Action Systems are compared syntactically. As a result, checking for the mutated action has been shown to be more efficient in matters of performance by a syntactic check.

For reachability analysis, various algorithms have been shown which perform differently on the selected test input. The tool's run time depends heavily on the number of state variables, the number of reachable states and the number of mutants that has to be checked. From the presented algorithms, this thesis contributed two of them (cmp. Section 5.2.3 and Section 5.2.4). It has been shown in the performed experiments (cmp. Section 6) and the algorithm analysis (cmp. Section 5.3.3) that on inputs fulfilling certain properties the contributed algorithms outperform the algorithm presented in [7].

Performance has been a major objective in the course of this thesis. This continuous improvement process has been majorly affected by the work of Elisabeth Jöbstl (co-adviser of this thesis) and her *as2csp* application. A comparison of the two tools has been documented by Aichernig et al. [10]. Since the release of that paper, performance has been improved further. In total, performance differences between the two tools are minor.

Scala has been selected as an implementation language for the *as2smt* tool. The advantages of the usage of Scala have been outlined throughout the thesis (cmp. Section 3.1.2, 4.2, 4.2.1, 4.2.3). This modern programming language receives growing interest. Especially, its membership in the JVM family seems to grow its popularity[1].

Similarly to SMT-COMP - the SMT solver competition, three different solvers have been compared in terms of performance: *MathSAT*, *SMTInterpol* and *Z3*. In almost all use cases *Z3* is the fastest, but *SMTInterpol* shows partly similar performance. Most interestingly, the Java-based *SMTInterpol* even outperforms *Z3* for two test configurations (cmp. Section 6.6).

Finally, it can be stated that model-based mutation testing allows test case generation in an efficient way for small models. Larger models have to be tested and algorithms may have to be adapted to scale. The largest model that was analyzed with this technique was the test model of an industrial measurement device [1, 55].

8.3 Future work

Originally, it was planned to support all kinds of arithmetic operators but just a subset has been implemented. Additionally, the support for data types like arrays has been on the list of features but has not been realised. The current tool may be extended to support this.

The integration of further solvers might also be a valuable extension to the existing tool as it would show also their performance. Among these is CVC4 (from New York University, United States of America) which has been released in the beginning of 2013. Noteworthy is the ability of parallel solving [22]. This could change the time behaviour severely on multi-core architecture.

Taking the idea of parallelism, the existing solving methods could be run each in its own thread. The quickest would then terminate the others and so the best fitting solving method (cmp. Section 6.5) would be *chosen* automatically.

[1]The author of *Groovy*, another functional programming language based on the JVM, states that he *would probably never created Groovy* if Scala had existed already in 2003 [80]. Furthermore, he sees Scala as the new replacement for Java.

A EBNF of Action System in Prolog Notation

The following Listing shows the LL(1)-version of grammar defining the Prolog Action System notation int ANTLR notation. It has been initially presented in a not publicly available documentation of the Ulysses tool (cmp. Section4.1). A subset of this definition has been transformed in an LL(1) grammar to avoid back-tracking of the parser. This tranformation has been achieved by the aid of the tool *ANTLRWorks* [74].

```
grammar action_system;

// PARSER

qas                : header declarations  system  (type_def)∗;

header             :  ’””(:−.∗)+””’;

declarations       : search_depth  (var_dec)+  state_  init  input;

search_depth       : SEARCH_DEPTH  LPAR  INT  RPAR  DOT;

var_dec            : VAR  LPAR  s_val_list  COLON  IDENT  RPAR  DOT;

s_val_list         : LBRA  IDENT  (COLON  IDENT)∗  RBRA;

i_val_list         : LBRA  INT  (COLON  INT)∗  RBRA;

qs_val_list        : LBRA  (QUOTE  IDENT  QUOTE  (COLON  QUOTE IDENT  QUOTE
    )∗)?  RBRA;

state_             : STATE_DEF  LPAR  s_val_list  RPAR  DOT;

init               : INIT  LPAR  i_val_list  RPAR  DOT;

input              : INPUT  LPAR  qs_val_list  RPAR  DOT;

q_string           : IDENT  | QUOTE  IDENT  QUOTE;

type_def           : TYPE  LPAR  q_string  COLON  X  RPAR  DESC  X  IN  INT
      DOT  DOT  INT  COLON  LABELING  LPAR  LBRA  RBRA  COLON  LBRA  X
    RBRA  RPAR  DOT;

call_list          : call (comp_op call_list)?;

call               : q_string  (LPAR  arg_list  RPAR)?;

condition          : term  (relop1  condition)?;

relop1             : IMPLIES_B | IMPLIES_R | IMPLIES_L;

relop2             : AND | OR | NEG;

relop3             : EQUALS | GREATER | SMALER| NEQUALS;

term               : factor (relop2 term)?;

factor             : boolean | rel_expr  | IDENT;
```

```

boolean           : TRUE | FALSE | FAIL;

rel_expr          : IDENT  relop3  INT;

stmt              : guard_com | IDENT  DEF arith_expr | SKIP;

guard_com         :  LPAR condition RPAR IMPL stmt_comp;

arith_expr        : IDENT (arith_op INT)∗ | INT;

arith_op          : PLUS | MINUS;

system            : AS  DESC  METHODS  (NONE| LPAR  NONE  RPAR)  COLON
    ACTIONS  LPAR  action_list  RPAR  COLON  DOOD  LPAR  call_list  RPAR
    COLON  S_QDES  (NONE | LPAR  NONE  RPAR)  DOT;

as                : ( as_ | LPAR  as_  RPAR);

as_               : q_string (LPAR  arg_list  RPAR)?  DDOT  DDOT  LPAR
    condition  RPAR  IMPL  LPAR  action_list  RPAR (COLON  as)?;

action_list       : (action | LPAR action RPAR) (COLON action_list)?;

action            : q_string DDOT DDOT guard_com ;

stmt_comp         : LPAR stmt_list RPAR (SEMICOLON stmt_comp)?;

stmt_list         : stmt (COLON stmt_list)?;

arg_list          : '';

comp_op           :  SEMICOLON | COLON | DSLASH;


// LEXER

MODULE            : 'module';
ACTIONS           : 'actions';
AS                : 'as';
METHODS           : 'methods';
DOOD              : 'dood';
NONE              : 'none';
SKIP              : 'skip';
S_QDES            : 'qdes';
SEARCH_DEPTH      : 'searchDepth';
VAR               : 'var';
STATE_DEF         : 'state_def';
INIT              : 'init';
INPUT             : 'input';
TYPE              : 'type';
IN                : 'in';
LABELING          : 'labeling';
X                 : 'X';
I                 : 'I';
TRUE              : 'true';
FALSE             : 'false';
FAIL              : 'fail';
```

```
NEG                 :  '#\\';
EQUALS              :  '#=';
NEQUALS             :  '#\\=';
AND                 :  '#/\\';
OR                  :  '#\\/';
GREATER             :  '#>';
SMALER              :  '#<';
IMPLIES_B           :  '#<=>';
IMPLIES_R           :  '#=>';
IMPLIES_L           :  '#<=';
DESC                :  ':-';
DEF                 :  ':=';
IMPL                :  '=>';
DSLASH              :  '//';
LPAR                :  '(';
RPAR                :  ')';
LBRA                :  '[';
RBRA                :  ']';
LCUR                :  '{';
RCUR                :  '}';
COLON               :  ',';
SEMICOLON           :  ';';
DOT                 :  '.';
DDOT                :  ':';
EQUAL               :  '=';
PLUS                :  '+';
MINUS               :  '-';
QUOTE               :  '\'';

IDENT               : ('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'0'..'9'|'_
       ')*
;

INT                 :  '0'..'9'+
;

WS                  :    ( ' '
    | '\t'
    | '\r'
    | '\n'
    ) {$channel=HIDDEN;}
;

COMMENT             :    '%' ~('\n'|'\r')* '\r'? '\n' {$channel=HIDDEN;skip()
       ;}
;
```

Listing A.1: EBNF of Action System in Prolog notation

B Action System Prolog Notation of the Rocket Steering Software Example

```
% crafted by Matthias Kegele

:- module(as, [var/2, input/1, searchDepth/1, qspace/2]).
:- use_module(library(clpfd)).
:- public(as/0).
:- dynamic(as/0).
:- dynamic(type/2).

% maximal search depth (change at will)
searchDepth(10).

% variable declarations
var([engine], bool).
var([integrity], int_0_2).
var([state], int_0_4).
var([x, y], n).

% state definition
state_def([engine, integrity, state, x, y]).

% initial state
init([0, 0, 0, 0, 0, 0]).

% controllable actions
input([]).

% action system
as :-
    methods (none),

    % actions
    actions (
      'engage' :: (integrity #= 0 #/\ engine #= 0 #/\ x #= 0 #/\ y #= 0)
          => (
  state := 1,
  engine := 1,
  x := 1,
  y := 1
      ),

      'course_correction'(x_next, y_next) :: (integrity #= 1 #/\ engine
          #= 1) => (
  % irregular course correction, destruction
  (((x - x_next #< 0) #\/ (y - y_next #< 0)) => (
    state := 2,
    integrity := 2)
  ;
  % regular course correction
  ((x - x_next #>= 0) #\/ (y - y_next #>= 0)) => (
    state := 1,
    x := x_next,
    y := y_next
)
    )),

    'land' :: (integrity #= 0 #/\ state #= 1) => (
state := 3,
engine := 0,
(
  % perfect landing
  integrity := 0
  ;
  % damage on landing
  integrity := 1)
    ),

    'repair' :: (engine #= 0 #/\ integrity #= 1) => (
% repair successful
integrity := 0
;
% repair fails
integrity := 1
    ),

    'integrity_check' :: (integrity #= 0 #/\ state #= 4) => (
% back to idle
state := 0
    )
  ),


dood (
  'engage'
  ; [A:n,B:n] : 'course_correction'(A, B)
  ; 'land'
  ; 'repair'
  ; 'integrity_check'
),

qdes (none).

%type_declarations
type(bool, X) :- X in 0..1, labeling([],[X]).
type(int_0_2, X) :- X in 0..2, labeling([],[X]).
type(int_0_4, X) :- X in 0..4, labeling([],[X]).
type(n, X) :- X in 0..10, labeling([],[X]).
```

Listing B.1: Rocket example Action System in Prolog

C SMT-LIB Translation of the Rocket Steering Software Example

```
; generated by the as2smt tool
; trace variable

(declare-fun trace () Int)
(assert (and (>= trace 0) (<= trace 5)))
(declare-fun trace_0 () Int)
(assert (and (>= trace_0 0) (<= trace_0 10)))
(declare-fun trace_1 () Int)
(assert (and (>= trace_1 0) (<= trace_1 10)))

; initial variable declarations

(declare-fun engine () Int)
(assert (and (>= engine 0) (<= engine 1)))
(declare-fun engine_post () Int)
(assert (and (>= engine_post 0) (<= engine_post 1)))
(declare-fun integrity () Int)
(assert (and (>= integrity 0) (<= integrity 2)))
(declare-fun integrity_post () Int)
(assert (and (>= integrity_post 0) (<= integrity_post 2)))
(declare-fun state () Int)
(assert (and (>= state 0) (<= state 4)))
(declare-fun state_post () Int)
(assert (and (>= state_post 0) (<= state_post 4)))
(declare-fun x () Int)
(assert (and (>= x 0) (<= x 10)))
(declare-fun x_post () Int)
(assert (and (>= x_post 0) (<= x_post 10)))
(declare-fun y () Int)
(assert (and (>= y 0) (<= y 10)))
(declare-fun y_post () Int)
(assert (and (>= y_post 0) (<= y_post 10)))

; parameter declarations

(declare-fun course_correction_x_next () Int)
(assert
  (and
    (>= course_correction_x_next 0)
    (<= course_correction_x_next 10)
   )
)
(declare-fun course_correction_y_next () Int)
(assert
  (and
    (>= course_correction_y_next 0)
    (<= course_correction_y_next 10)
  )
)

; translation of the action system

```

```
(assert
  (or
    (or
      (or
        (or
          (and
            (and
              (and
                (and
                  (= integrity 0)
                  (= engine 0)
                )
                (= x 0)
              )
              (= y 0)
            )
            (and
              (= trace 1)
              (and
                (= engine_post 1)
                (= integrity_post integrity)
                (= state_post 1)
                (= x_post 1)
                (= y_post 1)
              )
            )
          )
          (and
            (and
              (and
                (= integrity 1)
                (= engine 1)
              )
              (and
                (and
                  (= trace 2)
                  (= trace_0 course_correction_x_next)
                  (= trace_1 course_correction_y_next)
                )
                (or
                  (and
                    (or
                      (<
                        (- x course_correction_x_next)
                        0
                      )
                      (<
                        (- y course_correction_y_next)
                        0
          )
                    )
        (= engine_post engine)
        (= integrity_post 2)
        (= state_post 2)
        (= x_post x)
        (= y_post y)
                  )
                  (and
                    (or
                      (>=
                        (-x course_correction_x_next)
                        0
                      )
                      (>=
                        (- y course_correction_y_next)
                        0
                      )
                    )
                    (= engine_post engine)
                    (= integrity_post integrity)
                    (= state_post 1)
                    (= x_post course_correction_x_next)
                    (= y_post course_correction_y_next)
                  )
                )
              )
            )
            (and
              (>= course_correction_x_next 0)
              (<= course_correction_x_next 10)
            )
            (and
              (>= course_correction_y_next 0)
              (<= course_correction_y_next 10)
            )
          )
        )
        (and
    (and
      (= integrity 0)
      (= state 1)
    )
          (and
            (= trace 3)
            (or
              (and
                (= engine_post 0)
                (= integrity_post 0)
                (= state_post 3)
                (= x_post x)
                (= y_post y)
              )
              (and
                (= engine_post 0)
                (= integrity_post 1)
                (= state_post 3)
                (= x_post x)
                (= y_post y)
              )
            )
          )
        )
      )
      (and
        (and
          (= engine 0)
```

```
          (= integrity 1)
        )
        (and
          (= trace 4)
          (or
            (and
              (= engine_post engine)
              (= integrity_post 0)
              (= state_post state)
              (= x_post x)
              (= y_post y)
            )
            (and
              (= engine_post engine)
              (= integrity_post 1)
              (= state_post state)
              (= x_post x)
              (= y_post y)
            )
          )
        )
      )
    )
    (and
      (and
        (= integrity 0)
        (= state 4)
      )
      (and
        (= trace 5)
        (and
          (= engine_post engine)
          (= integrity_post integrity)
          (= state_post 0)
          (= x_post x)
          (= y_post y)
        )
      )
    )
  )
)
```

Listing C.1: Rocket example translation to SMT-LIB

Bibliography

[1] Bernhard K. Aichernig, Jakob Auer, Elisabeth Jöbstl, Robert Korosec, Willibald Krenn, Rupert Schlick, and Birgit Vera Schmidt. Model-based mutation testing of an industrial measurement device. In *Tests and Proofs - 8th International Conference, TAP 2014, Held as Part of STAF 2014, York, UK, July 24-25, 2014. Proceedings*, pages 1–19, 2014. (Cited on page 91.)

[2] Bernhard K. Aichernig, Harald Brandl, Elisabeth Jöbstl, and Willibald Krenn. Model-Based Mutation Testing of Hybrid Systems. In Frank S. de Boer, Marcello M. Bonsangue, Stefan Hallerstede, and Michael Leuschel, editors, *Formal Methods for Components and Objects - 8th International Symposium, FMCO 2009, Eindhoven, The Netherlands, November 4-6, 2009. Revised Selected Papers*, volume 6286 of *Lecture Notes in Computer Science*, pages 228–249. Springer, 2009. (Cited on pages 2, 8 and 88.)

[3] Bernhard K. Aichernig, Harald Brandl, Elisabeth Jöbstl, and Willibald Krenn. Efficient Mutation Killers in Action. In *IEEE Fourth International Conference on Software Testing, Verification and Validation, ICST 2011, Berlin, Germany, 21-25 March 2011*, pages 120–129. IEEE Computer Society, 2011. (Cited on pages 25 and 39.)

[4] Bernhard K. Aichernig, Harald Brandl, Elisabeth Jöbstl, and Willibald Krenn. UML in action: a two-layered interpretation for testing. *ACM SIGSOFT Software Engineering Notes*, 36(1):1–8, 2011. (Cited on pages ix, 25, 39, 77 and 78.)

[5] Bernhard K. Aichernig, Harald Brandl, and Willibald Krenn. Qualitative Action Systems. In Karin Breitman and Ana Cavalcanti, editors, *Formal Methods and Software Engineering, 11th International Conference on Formal Engineering Methods, ICFEM 2009, Rio de Janeiro, Brazil, December 9-12, 2009. Proceedings*, volume 5885 of *Lecture Notes in Computer Science*, pages 206–225. Springer, 2009. (Cited on pages 39 and 88.)

[6] Bernhard K. Aichernig and Jifeng He. Mutation testing in UTP. *Formal Aspects of Computing*, 21(1-2):33–64, 2009. (Cited on page 9.)

[7] Bernhard K. Aichernig and Elisabeth Jöbstl. Efficient Refinement Checking for Model-Based Mutation Testing. In Antony Tang and Henry Muccini, editors, *2012 12th International Conference on Quality Software, Xi'an, Shaanxi, China, August 27-29, 2012*, pages 21–30. IEEE, 2012. (Cited on pages 8, 25, 39, 53, 65, 68, 77, 88 and 91.)

[8] Bernhard K. Aichernig and Elisabeth Jöbstl. Towards Symbolic Model-Based Mutation Testing: Combining Reachability and Refinement Checking. In Alexander K. Petrenko and Holger Schlingloff, editors, *Proceedings 7th Workshop on Model-Based Testing, MBT 2012, Tallinn, Estonia, 25 March 2012.*, volume 80, pages 88–102, 2012. (Cited on pages 11, 13, 16, 39, 61 and 78.)

[9] Bernhard K. Aichernig and Elisabeth Jöbstl. Towards Symbolic Model-Based Mutation Testing: Pitfalls in Expressing Semantics as Constraints. In Giuliano Antoniol, Antonia Bertolino, and Yvan Labiche, editors, *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation, Montreal, QC, Canada, April 17-21, 2012*, pages 752–757. IEEE, 2012. (Cited on pages 8 and 49.)

[10] Bernhard K. Aichernig, Elisabeth Jöbstl, and Matthias Kegele. Incremental Refinement Checking for Test Case Generation. In Margus Veanes and Luca Viganò, editors, *Tests and Proofs - 7th International Conference, TAP 2013, Budapest, Hungary, June 16-20, 2013. Proceedings*, volume 7942 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2013. (Cited on pages ix, 9, 10, 77, 88 and 91.)

[11] Bernhard K. Aichernig, Florian Lorber, and Dejan Nickovic. Time for Mutants - Model-Based Mutation Testing with Timed Automata. In Margus Veanes and Luca Viganò, editors, *Tests and Proofs - 7th International Conference, TAP 2013, Budapest, Hungary, June 16-20, 2013. Proceedings*, volume 7942 of *Lecture Notes in Computer Science*, pages 20–38. Springer, 2013. (Cited on page 88.)

[12] Bernhard K. Aichernig and Percy Antonio Pari Salas. Test Case Generation by OCL Mutation and Constraint Solving. In *2005 NASA / DoD Conference on Evolvable Hardware (EH 2005), 29 June - 1 July 2005, Washington, DC, USA*, pages 64–71. IEEE Computer Society, 2005. (Cited on page 88.)

[13] Paul Ammann and Jeff Offutt. *Introduction to software testing.* Cambridge University Press, 2008. (Cited on page 2.)

[14] Domogaj Babic. Description of the DIMACS format. `http://www.domagoj-babic.com/uploads/ResearchProjects/Spear/dimacs-cnf.pdf`, November 2014. (Cited on page 20.)

[15] R.-J. Back. Correctness Preserving Program Refinements. Technical Report Mathematical Centre Tracts #131, Mathematisch Centrum Amsterdam, 1980. (Cited on pages 8 and 88.)

[16] Ralph Back, Martin Buchi, and Emil Sekerinski. Adding Type-Bound Actions to Action-Oberon. Technical report, Turku Centre for Computer Science, 1996. (Cited on page 88.)

[17] Ralph-Johan Back and Reino Kurki-Suonio. Decentralization of Process Nets with Centralized Control. In Robert L. Probert, Nancy A. Lynch, and Nicola Santoro, editors, *Proceedings of the Second Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, Montreal, Quebec, Canada, August 17-19, 1983*, pages 131–142. ACM, 1983. (Cited on pages 11 and 88.)

[18] Ralph-Johan Back and Reino Kurki-Suonio. Distributed Cooperation with Action Systems. *ACM Trans. Program. Lang. Syst.*, 10(4):513–554, 1988. (Cited on pages 11 and 88.)

[19] Ralph-Johan Back and Kaisa Sere. From Action Systems to Modular Systems. *Software - Concepts and Tools*, 17(1):26–39, 1996. (Cited on page 88.)

[20] R. M. Balzer. EXDAMS: extendable debugging and monitoring system. In *Proceedings of the May 14-16, 1969, spring joint computer conference*, AFIPS '69 (Spring), pages 567–580, New York, NY, USA, 1969. ACM. (Cited on page 53.)

[21] Clark Barrett et al. CVC3. `http://www.cs.nyu.edu/acsys/cvc3/`, November 2014. (Cited on page 30.)

[22] Clark Barrett et al. Parallel solving (CVC4). `http://cvc4.cs.nyu.edu/wiki/User_Manual#Parallel_solving`, November 2014. (Cited on page 91.)

[23] Clark Barrett, Aaron Stump, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2010. (Cited on pages 20, 21 and 30.)

[24] Clark Barrett, Aaron Stump, and Cesare Tinelli. The SMT-LIB Standard: Version 2.0. In A. Gupta and D. Kroening, editors, *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, UK)*, 2010. (Cited on pages 20, 22, 62 and 81.)

[25] Mordechai Ben-Ari. The bug that destroyed a rocket. *SIGCSE Bull.*, 33(2):58–59, June 2001. (Cited on page 1.)

[26] Marcello M. Bonsangue, Joost N. Kok, and Kaisa Sere. An Approach to Object-Orientation in Action Systems. In Johan Jeuring, editor, *Mathematics of Program Construction, MPC'98, Marstrand, Sweden, June 15-17, 1998, Proceedings*, volume 1422 of *Lecture Notes in Computer Science*, pages 68–95. Springer, 1998. (Cited on page 88.)

[27] Robert S. Boyer, Bernard Elspas, and Karl N. Levitt. SELECT - Formal System for Testing and Debugging Programs by Symbolic Execution. In *Proceedings of the International Conference on Reliable Software*, pages 234–245, New York, NY, USA, 1975. ACM. (Cited on page 53.)

[28] Angelo Brillout, Nannan He, Michele Mazzucchi, Daniel Kroening, Mitra Purandare, Philipp Rümmer, and Georg Weissenbacher. Mutation-Based Test Case Generation for Simulink Models. In Frank S. de Boer, Marcello M. Bonsangue, Stefan Hallerstede, and Michael Leuschel, editors, *Formal Methods for Components and Objects - 8th International Symposium, FMCO 2009, Eindhoven, The Netherlands, November 4-6, 2009. Revised Selected Papers*, volume 6286 of *Lecture Notes in Computer Science*, pages 208–227. Springer, 2009. (Cited on page 89.)

[29] Timothy A. Budd and Dana Angluin. Two Notions of Correctness and Their Relation to Testing. *Acta Inf.*, 18:31–45, 1982. (Cited on page 7.)

[30] Mats Carlsson et al. Sixtus Prolog documentation on enumeration predicates. https://sicstus.sics.se/sicstus/docs/4.2.3/html/sicstus.html/Enumeration-Predicates.html, November 2014. (Cited on page 11.)

[31] Oliver Chafik. JNAerator. http://code.google.com/p/jnaerator/, November 2014. (Cited on pages 23, 30 and 73.)

[32] Jürgen Christ. SMTInterpol. http://ultimate.informatik.uni-freiburg.de/smtinterpol/, November 2014. (Cited on pages 30 and 77.)

[33] Jürgen Christ, Jochen Hoenicke, and Alexander Nutz. SMTInterpol: An Interpolating SMT Solver. In Alastair F. Donaldson and David Parker, editors, *Model Checking Software - 19th International Workshop, SPIN 2012, Oxford, UK, July 23-24, 2012. Proceedings*, volume 7385 of *Lecture Notes in Computer Science*, pages 248–254. Springer, 2012. (Cited on page 19.)

[34] Alessandro Cimatti, Alberto Griggio, and Roberto Sebastiani. Computing Small Unsatisfiable Cores in Satisfiability Modulo Theories. *J. Artif. Intell. Res. (JAIR)*, 40:701–728, 2011. (Cited on page 19.)

[35] Lori A. Clarke. A System to Generate Test Data and Symbolically Execute Programs. *IEEE Trans. Software Eng.*, 2(3):215–222, 1976. (Cited on page 53.)

[36] David Cok. SMT-LIB turtorial. http://www.grammatech.com/resource/smt/SMTLIBTutorial.pdf, November 2014. (Cited on page 21.)

[37] Henry Coles. PIT mutation testing tool. http://http://pitest.org//, November 2014. (Cited on page 6.)

[38] Martin Davis, George Logemann, and Donald W. Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, 1962. (Cited on page 18.)

[39] Martin Davis and Hilary Putnam. A Computing Procedure for Quantification Theory. *J. ACM*, 7(3):201–215, 1960. (Cited on page 18.)

[40] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6,*

2008. Proceedings, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008. (Cited on pages 16 and 19.)

[41] Leonardo Mendonça de Moura, Nikolaj Bjørner, et al. Z3 tutorial. http://rise4fun.com/z3/tutorialcontent/guide#h23, November 2014. (Cited on page 22.)

[42] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 11(4):34–41, 1978. (Cited on pages 5 and 7.)

[43] Iulian Dragos et al. Scala IDE. http://scala-ide.org/, November 2014. (Cited on page 23.)

[44] R. Frost and John Launchbury. Constructing Natural Language Interpreters in a Lazy Functional Language. *Computer Journal*, 32(2):108–121, 1989. (Cited on page 40.)

[45] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995. (Cited on page 28.)

[46] Evguenii I. Goldberg and Yakov Novikov. BerkMin: A Fast and Robust Sat-Solver. In *2002 Design, Automation and Test in Europe Conference and Exposition (DATE 2002), 4-8 March 2002, Paris, France*, pages 142–149. IEEE Computer Society, 2002. (Cited on page 19.)

[47] Alberto Griggio et al. MathSAT. http://mathsat.fbk.eu/, November 2014. (Cited on pages 30 and 77.)

[48] R. G. Hamlet. Testing Programs with the Aid of a Compiler. *IEEE Transactions on Software Engineering*, SE-3(4):279–290, July 1977. (Cited on page 6.)

[49] Mark Harrah et al. Simple Build Tool (SBT). http://www.scala-sbt.org/, November 2014. (Cited on pages 23 and 33.)

[50] C.A.R. Hoare and J. He. *Unifying theories of programming*, volume 14. Prentice Hall, 1998. (Cited on page 9.)

[51] David Hoffman. I Had A Funny Feeling in My Gut. *Washington Post*, 1(10):A19, February 1999. (Cited on page 1.)

[52] Michael Huth and Mark Dermot Ryan. *Logic in computer science - modelling and reasoning about systems (2. ed.)*. Cambridge University Press, 2004. (Cited on pages 16 and 18.)

[53] Graham Hutton. Higher-Order Functions for Parsing. *J. Funct. Program.*, 2(3):323–343, 1992. (Cited on page 40.)

[54] Yue Jia and Mark Harman. An Analysis and Survey of the Development of Mutation Testing. *IEEE Trans. Software Eng.*, 37(5):649–678, 2011. (Cited on pages 6 and 7.)

[55] Elisabeth Jöbstl. *Model-Based Mutation Testing with Constraint and SMT Solvers*. PhD thesis, April 2014. (Cited on pages 10 and 91.)

[56] James C. King. A New Approach to Program Testing. In Clemens Hackl, editor, *Programming Methodology*, volume 23 of *Lecture Notes in Computer Science*, pages 278–290. Springer, 1974. (Cited on pages 53 and 54.)

[57] James C. King. On Generating Verification Conditions for Correctness Proofs. In Hans Jürgen Schneider and Manfred Nagl, editors, *Fachtagung über Programmiersprachen*, volume 1 of *Informatik-Fachberichte*, pages 253–267. Springer, 1976. (Cited on page 53.)

[58] Willibald Krenn and Bernhard K. Aichernig. Test Case Generation by Contract Mutation in Spec#. *Electr. Notes Theor. Comput. Sci.*, 253(2):71–86, 2009. (Cited on page 88.)

[59] Willibald Krenn, Rupert Schlick, and Bernhard K. Aichernig. Mapping UML to Labeled Transition Systems for Test-Case Generation - A Translation via Object-Oriented Action Systems. In Frank S. de Boer, Marcello M. Bonsangue, Stefan Hallerstede, and Michael Leuschel, editors, *Formal Methods for Components and Objects - 8th International Symposium, FMCO 2009, Eindhoven, The Netherlands, November 4-6, 2009. Revised Selected Papers*, volume 6286 of *Lecture Notes in Computer Science*, pages 186–207. Springer, 2009. (Cited on page 88.)

[60] Moez Krichen and Stavros Tripakis. Conformance testing for real-time systems. *Formal Methods in System Design*, 34(3):238–304, 2009. (Cited on page 89.)

[61] Daniel Kroening and Ofer Strichman. *Decision Procedures: An Algorithmic Point of View*. Springer Publishing Company, Incorporated, 1 edition, 2008. (Cited on page 18.)

[62] Matt Lake. The Big Cost of Software Bugs. http://www.computerworld.com/s/article/9183580/Epic_failures_11_infamous_software_bugs, September 2010. (Cited on pages x and 1.)

[63] J.P. Marques-Silva and K.A. Sakallah. GRASP – A New Search Algorithm for Satisfiability. In *Proceedings of the 1996 IEEE/ACM International Conference on Computer-aided Design*, pages 220–227, 1996. (Cited on page 19.)

[64] Jim McBeath. Scala Operator Precedence. http://jim-mcbeath.blogspot.co.at/2008/09/scala-parser-combinators.html, November 2014. (Cited on page 46.)

[65] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), April 1965. (Cited on page 1.)

[66] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th Design Automation Conference, DAC 2001, Las Vegas, NV, USA, June 18-22, 2001.*, pages 530–535. ACM, 2001. (Cited on page 19.)

[67] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Abstract DPLL and Abstract DPLL Modulo Theories. In Franz Baader and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning, 11th International Conference, LPAR 2004, Montevideo, Uruguay, March 14-18, 2005, Proceedings.*, volume 3452 of *Lecture Notes in Computer Science*, pages 36–50. Springer, 2004. (Cited on page 19.)

[68] Martin Odersky et al. Description of the DIMACS format. http://www.scala-lang.org/documentation/books.html, November 2014. (Cited on page 23.)

[69] Martin Odersky et al. Scala Programming Language. http://www.scala-lang.org/, November 2014. (Cited on page 23.)

[70] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala*. Artima, Mountain View, CA, 2nd edition edition, 2008. (Cited on pages 23, 27, 28, 40, 44 and 65.)

[71] A. Jefferson Offutt. Investigations of the Software Testing Coupling Effect. *ACM Trans. Softw. Eng. Methodol.*, 1(1):5–20, 1992. (Cited on page 7.)

[72] Terrence Parr et al. ANTLR parser generator. http://www.antlr.org, November 2014. (Cited on page 40.)

[73] Terrence Parr et al. ANTLR, tree pattern matching. https://theantlrguy.atlassian.net/wiki/display/ANTLR3/Tree+pattern+matching, November 2014. (Cited on page 27.)

[74] Terrence Parr et al. ANTLR Works. http://www.antlr3.org/works/, month = nov, title = ANTLR parser generator, year = 2013, 2014. (Cited on pages 23, 39 and 92.)

[75] Silvio Ranise and Cesare Tinelli. The SMT-LIB format: An initial proposal. In *Pragmatics of Decision Procedures in Automated Reasoning 2003, July 29, 2003, Miami, USA*, pages 94–111, 2003. (Cited on page 19.)

[76] Microsoft Research. Z3. http://z3.codeplex.com/, November 2014. (Cited on pages 30 and 77.)

[77] Martin Richards. BCPL: A tool for compiler writing and system programming. In *American Federation of Information Processing Societies: AFIPS Conference Proceedings: 1969 Spring Joint Computer Conference, Boston, MA, USA, May 14-16, 1969*, pages 557–566. ACM, 1969. (Cited on page 45.)

[78] Jordan Robertson. The Big Cost of Software Bugs. http://www.bloomberg.com/slideshow/2012-08-03/the-big-cost-of-software-bugs.html, August 2012. (Cited on pages x and 1.)

[79] SMT-COMP. Results of the SMT solver competition SMT-COMP. http://www.smtexec.org/exec/?jobs=856, July 2011. (Cited on page 30.)

[80] James Strachan et al. Scala as the long term replacement for java/javac? http://macstrac.blogspot.co.at/2009/04/scala-as-long-term-replacement-for.html, November 2014. (Cited on page 91.)

[81] Phillip Suter. $Scala^Z3$. https://github.com/psuter/ScalaZ3, November 2014. (Cited on page 30.)

[82] Jan Tretmans. Test generation with inputs, outputs, and quiescence. In Tiziana Margaria and Bernhard Steffen, editors, *Tools and Algorithms for Construction and Analysis of Systems, Second International Workshop, TACAS '96, Passau, Germany, March 27-29, 1996, Proceedings.*, volume 1055 of *Lecture Notes in Computer Science*, pages 127–146. Springer, 1996. (Cited on page 88.)

[83] Mark Utting and Bruno Legeard. *Practical model-based testing: a tools approach*. Morgan Kaufmann, 2006. (Cited on pages 4 and 5.)

[84] Mark Utting, Alexander Pretschner, and Bruno Legeard. A Taxonomy of Model-Based Testing Approaches. *Softw. Test. Verif. Reliab.*, 22(5):297–312, 2012. (Cited on pages 2, 3 and 4.)

[85] Bill Venners et al. ScalaTest Framework. http://www.scalatest.org/, November 2014. (Cited on page 23.)

[86] Alessandro Warth, James R. Douglass, and Todd D. Millstein. Packrat parsers can support left recursion. In Robert Glück and Oege de Moor, editors, *Proceedings of the 2008 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation, PEPM 2008, San Francisco, California, USA, January 7-8, 2008*, pages 103–110. ACM, 2008. (Cited on page 39.)

[87] Niklaus Wirth. A Plea for Lean Software. *Computer*, 28(2):64–68, 1995. (Cited on page 1.)