Dissertation

Automatic Object Type Test Input Data Generation for Java Programs based on Design by Contract Specification

Stefan J. Galler

Graz, 2011

Institute for Software Technology
Graz University of Technology

Supervisor/First reviewer: Univ.-Prof. Dipl.-Ing. Dr. techn. Franz Wotawa
Second reviewer: Prof. Ph.D. Rob Hierons

To my family and friends who
accompanied and supported me.

Thank you!

Gerald Steinbauer Sandra Lang Harald Brandl Jörg Weber Viktoria Galler Mar-
tin Weiglhofer Iris Uitz David Galler Georg Parsché Eva Galler
Rob Hierons Karl Voit Romana Hoffer Simona Nica Bernhard Aichernig Thomas
Hackl Oswin Aichholzer Chimisliu Valentin Andrea Mayr Bernhard Peischl
Birgit Vogtenhuber Gordon Fraser Elisabeth Jöbstl Romy Hoffer Monika Schu-
bert Stela Bulic Marlena Gaertner Pilz Alexander Willibald Krenn Christoph
Zehentner Dietmar Gaube Thomas Quaritsch Birgit Hofer Arabella
Gass Mihai Nica Romana Hoffer Nicola Gaertner Monika Mandl Josef Galler
Andreas Klammer Petra Pichler Bernhard Peischl Andrea Mayr Petra Pichler
Viktoria Galler Sandra Lang Romy Hoffer Marlena Gaertner Dietmar
Gaube Willibald Krenn Franz Wotawa Gordon Fraser Karl Voit Thomas
Hackl Bernhard Aichernig Georg Parsché Birgit Hofer Iris Uitz Josef Galler
Christoph Zehentner Romy Hoffer Eva Galler Stela Bulic Monika Schubert
Romana Hoffer David Galler Thomas Quaritsch Monika Mandl
Chimisliu Valentin Birgit Vogtenhuber Elisabeth Jöbstl Andrea Mayr Mihai Nica
Pilz Alexander Andreas Klammer Robert Mayr Gerald Steinbauer Oswin Aich-
holzer Nicola Gaertner Martin Weiglhofer Rob Hierons Arabella Gass
Jörg Weber Simona Nica Harald Brandl Stela Bulic Romy Hoffer Rob Hierons
Christoph Zehentner Romana Hoffer Andreas Klammer Bernhard Peis-
chl Birgit Hofer Monika Mandl Eva Galler Monika Schubert Nicola Gaertner
David Galler Martin Weiglhofer Bernhard Aichernig Sandra
Lang Simona Nica Franz Wotawa Willibald Krenn Andrea Mayr Robert Mayr
David Galler Pilz Alexander Elisabeth Jöbstl Iris Uitz Oswin Aichholzer Mi-
hai Nica Harald Brandl Birgit Vogtenhuber Marlena Gaertner Gerald
Steinbauer Petra Pichler Jörg Weber Viktoria Galler Dietmar
Gaube Gordon Fraser Georg Parsché Thomas Quaritsch Arabella Gass Thomas
Hackl Marlena Gaertner Chimisliu Valentin Karl Voit Thomas Hackl San-
dra Lang Josef Galler Andreas Klammer Bernhard Aichernig Karl
Voit Robert Mayr Franz Wotawa Oswin Aichholzer Viktoria Galler Martin
Weiglhofer Iris Uitz Rob Hierons Gerald Steinbauer Stela Bulic Bernhard
Peischl Georg Parsché Eva Galler Willibald Krenn Thomas Quar-
itsch Andrea Mayr Pilz Alexander Arabella Gass Birgit Vogtenhuber Elisa-
beth Jöbstl Petra Pichler Harald Brandl Josef Galler Gordon
Fraser Monika Schubert Monika Mandl Mihai Nica Nicola Gaertner Christoph
Zehentner Dietmar Gaube Simona Nica Jörg Weber Chimisliu Valentin
Birgit Hofer Monika Schubert Christoph Zehentner Robert Mayr Stela Bulic Mi-
hai Nica Franz Wotawa Willibald Krenn Georg Parsché Thomas Quaritsch Iris
Uitz Andreas Klammer Elisabeth Jöbstl Arabella Gass Petra Pichler
Josef Galler Bernhard Aichernig Eva Galler Gerald Steinbauer David Galler
Birgit Vogtenhuber Karl Voit Monika Mandl Thomas Hackl Viktoria Galler
Georg Parsché Oswin Aichholzer Gordon Fraser Harald Brandl Marlena
Gaertner Pilz Alexander Sandra Lang Franz Wotawa Rob Hi-
erons Simona Nica Nicola Gaertner Birgit Hofer Viktoria Galler Bernhard Peis-
chl Christoph Zehentner Chimisliu Valentin Dietmar Gaube
Jörg Weber Martin Weiglhofer Gordon Fraser Georg Parsché Birgit Hofer

Abstract (English)

Software is not only part of our everyday life but it becomes more and more integrated into everything we do. Two decades ago everybody could tell what a computer was. They were huge and expensive. Nowadays, computers are so small that we do not even know where they are embedded. We basically trust and have to trust computers and computer software that they do what they are supposed to do. Trust but verify!

One way to verify a software system is testing it. Therefore, leading software engineering companies and universities alike work on improving test technologies. Especially automating the process of testing gets a lot of attraction, since it has an immediate impact on the costs of software development.

Automation can take part in every step of the testing process. Automating the process of test execution is well-known and widely used in industry. Automatically generating tests and all required test input data, tough, it is still not fully accepted by industry. This work presents four approaches that improve automated test data generation of object types.

STACI automatically configures the static behavior of a mock object such that it satisfies the precondition of the method under test.

AIANA determines a method call sequence, which transforms all object type parameters of the method under test into a precondition satisfying state by means of AI planning.

SYNTHIA synthesizes a fake class that behaves according to its specification and replaces the original class and all its dependencies in a unit test.

INTISA uses bounded model-checking by means of SMT solving to calculate initial values for SYNTHIA that satisfy the precondition of the method under test. In addition, they are re-produceable through the public interface of the actual class.

Especially the combination of INTISA and SYNTHIA shows promising results on the two case studies used for evaluation: *StackCalc* and *StreamingFeeder*. The latter being a real-world application implemented and specified by our industry partner from the telecommunication industry. INTISA and SYNTHIA speed up the test generation process and provide a technology that allows to test most of the methods automatically.

Future work should not focus on one technology to solve all test data generation problems, but should try to standardize those technologies such that high-level test data generation framework are able to decide which technology fits best each time a new test input value has to be generated.

Kurzbeschreibung

Software ist nicht nur ein Teil unseres Lebens, mittlerweile sind wir ständig umgeben von Software. Vor zwei Jahrzehnten wusste jeder genau wo ein Computer eingebaut ist. Schlicht und einfach aus dem Grund, weil Compter damals ganze Räume füllten. Heute sind Computer so klein, dass wir gar nicht genau wissen, wo überall ein Computer eingebaut ist. Wir vertrauen einfach darauf, dass alle Computer ihre Berechtigung haben und wissen was sie tun. Aber Vertrauen ist gut, Kontrolle ist besser!

Testen ist eine Möglichkeit, die Fehlerfreiheit einer Software sicherzustellen. Führende Softwareentwicklungs Unternehmen wie auch Universitäten arbeiten daher an immer besser werdenden Test-Technologien. Speziell auf die Automatisierung des Testprozesses wird großer Wert gelegt, da dies merkbar Kosten reduziert.

Automatisierung kann in jedem Teilprozess stattfinden. Das Ausführen von händisch geschriebenen Tests findet auch in der Industrie bereits breiten Anklang. Das automatische Generieren von Tests und Test-Eingabedaten jedoch wenig. Diese Arbeit stellt vier Ansätze zum automatischen Erstellen von Test-Eingabedaten vor.

STACI ist ein Ansatz, der auf Basis von gegebener Spezifikation ein Mock-Objekt so konfiguriert, dass es die Vorbedingungen eines Tests erfüllt.

AIANA ermittelt mit Hilfe eines AI Planers eine Methoden-Sequenz, die alle Objekte, die die getestete Methode benötigt, in den gewünschen Zustand versetzt.

SYNTHIA erzeugt ein Objekt, welches sich an die gegebene Verhaltensbeschreibung hält, allerdings während der Testausführung die Original-Objekte ersetzt und dadurch Abhängigkeiten zu anderen Komponenten reduziert.

INTISA verwendet “bounded model-checking” und einen “SMT solver” um einen Anfangszustand für alle Test-Eingabedaten zu berechnen, welcher die Vorbedingungen des Tests erfüllen und tatsächlich über die öffentliche Schnittstelle der Klasse erzeugt werden kann. INTISA liefert die Ausgangswerte für SYNTHIA.

Speziell die Kombination von INTISA und SYNTHIA liefert vielversprechende Resultate. Die Ansätze wurden in zwei Fallstudien evaluiert: *StackCalc* und *StreamingFeeder*. Letztere ist eine Applikation, die von unserem Industriepartner entwickelt und spezifiziert wurde. Mit INTISA und SYNTHIA können mehr Methoden in weniger Zeit getestet werden.

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, ____________________   ____________________

Place, Date   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 entnommene Stellen als solche kenntlich gemacht habe.

Graz, am ____________________   ____________________

Ort, Datum   Unterschrift

Contents

List of Figures

List of Tables

Part I.

Introduction and Preliminaries

Chapter 1

Introduction

Software is in every part of our life. It is software that wakes us up in the morning, makes us coffee, tells us the early morning news. It is software that drives us to our working place that controls the traffic lights, that moves the elevator. It is software that flies planes and keeps nuclear reactors under control. These software components are getting more and more complex, have to be maintained and updated.

Hight quality software is therefore crucial. In academia and industry many people are working on new technologies to improve software quality and simultaneously reduce costs.

Even though "testing can only show the presence of errors, not their absence" [Dijkstra, 1972] testing is the one technology that is already widely used in industry. Mainly due to its efficiency and applicability.

Software testing is a very broad field of research. Therefore, the remaining part of the *Introduction* will layout on which area this thesis focuses.

1.1. Testing Notation

A very common test level classification is based on software process steps [Ammann and Offutt, 2008, p. 5].

Unit Tests test software with respect to their implementation.

Module Tests test software with respect to the detailed design.

Integration Tests test software with respect to interaction of modules.

System Tests test software with respect to architectural design.

Acceptance Tests test software with respect to user requirements.

Regression Tests test software with respect to its consistency to previous releases.

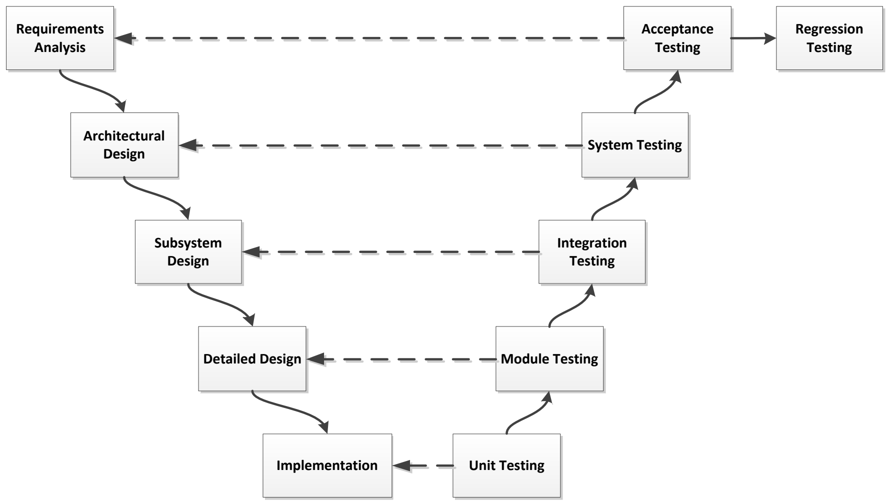


Figure 1.1.: Common test classification. This reprint of the well-known V-Modell in software engineering shows the categorization of test activities based on traditional steps in the software development process.

From this categorization one can see that testing requires always some part that defines what is correct and what not - the *test oracle* [Weyuker, 1982]. For regression tests it is the test result from the previous version. For acceptance tests it is the user (or a document describing the user requirements). For unit tests it might be some manually written assertions, or a formal specification nested inside the source code.

This thesis focuses on unit testing where the oracle is provided by means of Design by Contract™ specifications.

Another popular categorization focuses on the available information [Board, 2007]:

Black-box Tests test either functional or non-functional, without reference to the internal structure of the component or system.

White-box Tests test the system or component based on an analysis of the internal structure of it.

The explanations are taken from the Glossary published by the International Software Testing Qualifications Board.

Recently a third category was added: *gray-box tests*. This references to tests that have both source code and formal specification available.

This thesis focuses on gray-box tests.

Other test classification schemata focus on the domain of the test. For example, stress tests, usability tests, security tests, and performance tests. This classification schema can be seen orthogonal to the classification schema based on the software development process.

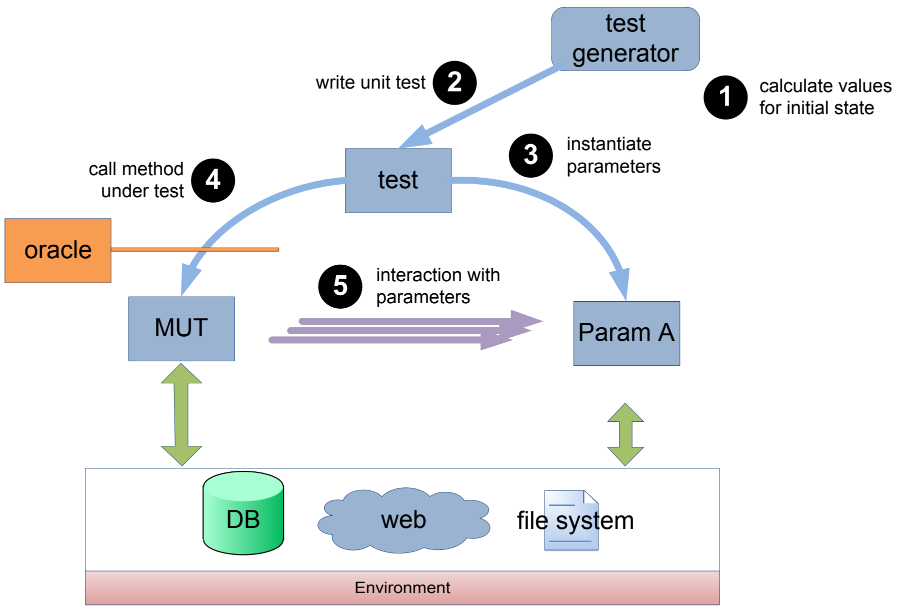


Figure 1.2.: Participants in Unit Testing.

1.2. Formal Specifications

Testing where only source code is available is not automatable. As already mentioned earlier, for testing an *oracle* is required to determine what is a correct behavior and what is an incorrect behavior. Such an *oracle* can be provided by a formal specification. Hierons et al. [Hierons et al., 2009] recently published a well-arranged overview of formal specifications for testing. He distinguishes between (*a*) model-based languages, (*b*) finite state-based languages, (*c*) process algebra state-based languages, and (*d*) hybrid languages.

Model-based languages provide a mechanism to specify the intended behavior by means of states and possible state changes of the system. Finite state-based languages are similar to the former, but only specifications over a finite set of states. Process algebra languages, on the other hand, describe the system as a number of communicating concurrent processes. Hybrid systems provide specifications that allow to state both discrete and continuous mathematics.

According to this, Design by Contract™ languages are part of model-based languages. Design by Contract™ provide mechanism to formally describe the behavior of a method, in other words the semantics of the methods.

This thesis focuses on testing where the oracle is provided by Design by Contract™ specifications.

1.3. Participants in Testing

Figure 1.2 shows an overview of the common elements that are part of unit testing: (*a*) test generator, (*b*) test, (*c*) system/class/method under test, (*d*) parameters, (*e*) the oracle, and (*f*) the environment.

The *test generator* can be either a human test engineer or, a software itself. The *test* is the piece of code which is executed to run the actual unit test. It is responsible for instantiating all *parameter* objects that are required by the *method under test*. In addition, the *test* is responsible to instantiate the system/class under test and transforms it in the required state. Furthermore, the *test* has to setup the correct environment consisting of databases, establishing connections to network resources and similar requirements. Finally, the *test* calls the *method under test* and passes all parameters. The *test oracle* evaluates the state of all involved objects after method execution and checks whether the result is as expected or not. The *test oracle* may be handwritten assertion statements at the end of the unit test, or any automatically inserted checks deducted from a formal specification.

This thesis focuses on test generators by means of software.

In automated software test generation and execution three different outputs are possible:

meaningless A test is considered to be meaningless if the passed parameters do not satisfy the *method under tests* requirements. Whether the requirement is expressed by a formal specification or only informal.

failing A test is considered to be failing if it is not meaningless and the result of executing the *method under test* is unexpected.

success A test is considered to be successful in any other case. That is if it is not meaningless and the result of executing the *method under test* is as expected.

This thesis focuses on generating test input data that leads to non-meaningless test cases.

1.4. Test Automation

Considering all participants in testing, automation may take place at different levels: (*a*) executing tests, (*b*) generating empty test classes and methods, (*c*) generating test cases, and (*d*) generating test data. Many tools exist that automate test execution, for example the *JUnit* framework. Most of the state-of-the-art integrated development environments (IDEs), such as Microsoft Visual Studio, and Eclipse, include tools that automatically generate empty test classes and methods. Current research efforts focus on automatically generating test cases and test data. The former automates the process of finding out which method sequence may reveal an error. The latter automates the generation of primitive values and especially non-primitive objects that can be used in test cases.

This thesis focuses on automating the process of test data generation for non-primitive objects.

1.5. Problem Statement

Testing object-oriented programs is very complex. A typical unit test for an object-oriented program has to (*a*) instantiate a receiver object, by calling a constructor of the receivers type, (*b*) optionally

	generated tests			
	succeding	failing	meaningles	tests/ MUT .
rand	21,8	0	617,6	0,032
rand + man	33,4	0	550,4	0,050
Z3	21,4	0	622,4	0,032
Z3 + man	36	0	541,2	0,052

Table 1.1.: Initial experiment results.

transform the receiver object into the required state by calling methods on it (*c*) call the method under test. Each of these steps may require concrete values to be passed as parameters to the constructors or methods executed. These parameters may be of primitive or object type. In case of object types the first two steps of the previously mentioned list have to be executed recursively.

Example 1.1. Consider *sumUp* from Listing 1.1 as the method under test. A unit test for it has to instantiate a Stack object, which consists of at least two elements as given by the Design by Contract™ specification.

```java
class SummationOperator {
  @Pre("stack.size()>=2")
  void sumUp(Stack stack) { ... }
}
```

Listing 1.1: Trivial precondition for a *Stack* instance.

The precondition given in Listing 1.1 is trivial. Nevertheless it is very unlikely for a random approach to generate a *Stack* object that satisfies this trivial precondition.

Consider the Java Stack implementation. It implements five methods and inherits another 55. Only seven methods of those add an element, nine methods remove elements, all other methods are so called state observing methods, or pure methods. The likelihood that two times in a row a method that adds an element to the Stack is chosen is therefore given by

$$\frac{7}{60} * \frac{7}{60} = \frac{49}{3600} = 0,0136 \approx 1,4\%.$$

Even if another random strategy only uses the methods declared in the *Stack* class only 4% of all generated *Stack* instances will satisfy the given precondition[*]. □

Initial experiments on two case studies, *StackCalc* and a RoboCup simulation league application, confirm the theory [Galler et al., 2008]. Table 1.1 shows the results of using a state-of-the-art random test generation approach for Java at that time, called JET [Cheon et al., 2005a], for generating unit tests for the *StackCalc* case study. The first line shows the results for the original *JET* implementation. The following lines are improved versions, where *man* means that in addition to random manually defined test input values were used and *Z3* indicates that the Z3 [de Moura and Bjø rner, 2008] SMT solver from Microsoft research was used for primitive values.

[*]Note, this is a simplified calculation. A real probabilistic approximation would have to include probabilities for choosing the decision how many methods are called, and when the process is stopped.

1.6. Thesis Statement

The stated problem can be solved by (*a*) separating the calculation of the required initial state of an object, and (*b*) replacing the original implementation of the object with a mock object.

By using AI planning techniques and SMT solver an initial state for the object can be calculated that is both reachable through out the public interface of the object and satisfies the precondition of the method under test. Synthesizing a class implementation for the original class that simulates the actual behavior given in terms of Design by Contract™ specifications removes any dependencies to the environment and finds in addition bugs that are hidden by the implementation due to implicit stronger assumptions as imposed by the actual Design by Contract™ specification.

1.7. Contributions

This thesis presents approaches to instantiate object types in a specific state, such that they satisfy the precondition of the method under test. This can be done by either, calculating a method sequence that transforms the object into the required state, or by replacing the actual implementation with a mock object. The thesis presents approaches for both categories.

AIANA [Galler et al., 2010c] and INTISA [Galler et al., 2011] calculate method sequences which transform the object in a precondition satisfying state. AIANA builds a planning problem based on the Design by Contract™ specification. INTISA uses ideas of bounded model-checking to find a sequence of method calls that reach the precondition satisfying state.

STACI [Galler et al., 2010a] introduces the idea of replacing all object type parameters with mock objects and provides an algorithm that calculates return values for all methods of the mock object such that it passes the precondition of the method under test. SYNTHIA [Galler et al., 2010b] is an improved mock object generator. It actually provides an API to generate SYNTHIA FAKE objects which use an SMT solver at runtime to calculate the return values for all methods according to the specification. INTISA perfectly integrates with SYNTHIA, thus INTISA can also be used to calculate the initial values for all SYNTHIA FAKE objects.

1.8. Outline

The thesis continues in Chapter 2 with the introduction of Design by Contract™, Satisfiability Modulo Theories, and existing test data generation techniques and tools. Furthermore, it tries to establish a common notation and outlines the case studies for evaluating the ideas presented in the thesis.

The second part of the thesis starts in Chapter 3 with the conceptional overview of all four presented test data generation approaches, continues with common definitions in Chapter 4 before in Chapter 5 STACI, in Chapter 6 AIANA, in Chapter 7 SYNTHIA, and in Chapter 8 INTISA are introduced.

The final part contains the conclusion (Chapter 9) and remarks on future work (Section 10) of the thesis.

Chapter 2

Preliminaries

2.1. Design by Contract™

The following sections explain the concept of Design by Contract™ by comparing it to a well known concept from our daily life: legal contracts. The history of Design by Contract™ is rewritten since it is not a complete new concept but builds on ideas reaching back to Hoare. Furthermore, the syntax and semantics of the general Design by Contract™ specification keywords are explained based on an example.

2.1.1. Concept

Design by Contract™ is a legal contract. It involves at least two parties - *the caller* and *the callee*. It states what the caller has to deliver - *the precondition* - and what it may expect in return from the callee - *the postcondition*. Furthermore, there is a general legal system with which all participants have to comply to - *the invariant*.

In terms of software engineering the term Design by Contract™ was coined by Bertrand Meyer when he introduced the programming language Eiffel, which allows to add semantic information to classes and methods by means of *invariants*, *preconditions*, and *postconditions* [Meyer, 1997].

The origins of Design by Contract™ lie in the Hoare triple [Hoare, 1969] $\{P\}S\{Q\}$, which is slightly modified for Design by Contract™ to:

$$\{P \land I\} \quad S \quad \{Q \land I\}$$

It states that if the precondition P holds before executing the system S, the postcondition Q holds afterwards. In the case of Design by Contract™ the invariant I is conjuncted to the precondition as well as to the postcondition, since the invariant states what holds at all time.

Many languages and tools build on this idea of pre- and postconditions, such as VDM [Jones, 1986], B [Johnson and Kernighan, 1973], and Z [Spivey, 1989]. Novel about Design by Contract™ is that the given specification is executable. Therefore, the practical use extends pure documentation. Runtime assertion checkers, which execute the given Design by Contract™ specifications (at runtime), throw an assertion error whenever a specification is violated. It immediately blames the part of software that does not comply with its specification. An error in the caller emphasizes a precondition violation, whereas an error in the callee is emphasized by a postcondition violation. Especially for automated testing, Design by Contract™ and runtime assertion checking are helpful. Design by Contract™ provides the test oracle for free. Any postcondition violation shows an implementation error with respect to the specification.

These advantages popularized Design by Contract™. By the year 2010 Eiffel is still the only programming language with built-in support of Design by Contract™, but many language extensions exist for basically all other state-of-the-art programming languages.

JML [Leavens et al., 1999] for Java is probably the best known Design by Contract™ extension. *Modern Jass* [Rieken, 2007], Jcontract [Parasoft, 2010a] and Contract4J [Wampler, 2006] are other extensions for Java. *Modern Jass* was developed as a master's thesis. Jcontract is the proprietary specification language of Parasoft, Inc.

In recent years Microsoft Research developed several tools and techniques that are related to Design by Contract™. Spec# [Barnett et al., 2005] and Code Contracts [Barnett, 2010] are two approaches to add Design by Contract™ specifications to the .NET languages. PEX [Tillmann and de Halleux, 2008] is an automated testing tool that uses Design by Contract™ specification to improve results.

Specification Semantics

The three main concepts of Design by Contract™ are [Meyer, 1992a]:

1. *precondition*: The precondition P is evaluated before the method is called. It specifies the expected value range of the method's parameters.
2. *postcondition*: The postcondition Q is evaluated directly after the method body is executed and before returning to the caller. It specifies the guaranteed behavior of the method. Whenever the client satisfies the precondition the method has to satisfy the postcondition, i.e., $P \implies Q$.
3. *class invariant*: A class invariant I specifies a property that applies for all instances of the class throughout their lifetime. All class invariants are checked before and after a method execution, i.e., $P \wedge I \implies Q \wedge I$. The invariant may be violated while executing a method's body.

The Eiffel keywords for preconditions and postconditions are *requires* and *ensures*, respectively.

Based on the example specification of an *Account* class given in Figure 2.1 the main specification concepts of Design by Contract™ are explained. The example is given by means of Java syntax with *Modern Jass* annotations, since all case studies through out the thesis were conducted on this setting.

Example 2.1. The *deposit()* method declaration in Figure 2.1 is enhanced with Design by Contract™ specification in terms of a precondition and a postcondition in lines 7 and 8, respectively. The precondition in Line 7 *requires* the caller to pass only values greater than zero and small enough to prevent *balance* from overflowing. If the method is called with precondition satisfying parameters, the postcondition in Line 8 has to *ensure* that the deposit amount has been added to *balance*. The invariant in Line 4 states that overdrawing the account is forbidden. □

```java
@Model(name=mBalance, type=Integer)
@Represents(name=mBalance, by=balance)
public class Account {
  @Invariant("balance>=0")
  protected integer balance;

  @Pre("amount>0 && amount<MAX_INT-balance")
  @Post("mBalance==@Old(mBalance) + amount")
  public void deposit(int amount) {
    //implementation
  }

  @Pre("amount>0 && amount<MAX_INT-balance")
  @Post("getBalance()==@Old(getBalance()) + amount")
  public void withdraw(int amount) {
    //implementation
  }

  @Pure
  @Post("@Return==mBalance")
  public int getBalance() {
    //implementation
  }

  @Also({
    @SpecCase(pre="getBalance()>0", post="@Return==true"),
    @SpecCase(pre="getBalance()==0", post="@Return==false")
  })
  public boolean isEmpty() {
    //implementation
  }
}
```

Figure 2.1.: Design by Contract™ Annotated Source Code Example. It emphasizes invariants, preconditions, and postconditions with pre-state access. The *Account* class is often used in the Design by Contract™ community, e.g. by Cheon et al. [Cheon and Rubio-Medrano, 2007] and Dhara et al. [Dhara and Leavens, 1996]
.

Two special keywords may occur in a postcondition (*a*) @*Old(*var0*)*, and (*b*) @*Return*.

The @*Old* keyword provides a way to access the value of a variable before method execution in the postcondition. In other words it provides a way to access the pre-state value of a variable.

Example 2.2. The postcondition of *deposit()* in Line 8 states that the value of *mBalance* after method execution (post-state) is equal to adding *amount* to the value of *mBalance* before method execution (pre-state). □

The @*Return* keyword provides a mechanism to access the return value of a method. The evaluation of a postcondition takes place right before returning the scope to the callee. The @*Return* keyword allows to access the return value at that particular time without being able to modify it.

Example 2.3. The postcondition in Line 20 states that *getBalance()* returns the current value of *mBalance*. □

Methods may have more than one behavior according to the current state. Therefore, most Design by Contract™ specification languages allow to express *multiple behavior* by means of special keywords, such as @*Also* in *Modern Jass* and *also* in *JML*. Multiple pre-/postcondition pairs are logically interpreted as $P_1 \implies Q_1 \wedge \cdots \wedge P_n \implies Q_n$, with P_1 to P_n being all preconditions and Q_1 to Q_n the corresponding postconditions. In other words, as long as the precondition is not satisfied, the postcondition is irrelevant. But if the precondition is satisfied the corresponding postcondition has to be satisfied as well.

Example 2.4. The *isEmpty()* method in Line 29 has two different behaviors: It returns *true* if there is money on the account, *false* otherwise. This *multiple behavior* is defined by two @*SpecCase* statements in Lines 26 and 27, which link a precondition to its postcondition. □

The example in Figure 2.1 shows two more features that are extensions to the core Design by Contract™ concept: (*a*) pure methods, and (*b*) model fields.

Pure methods are side effect free methods, with respect to the object state [Leavens and Cheon, 2003]. They may return a value but do not assign values to member variables. Only pure methods are allowed to be used in a Design by Contract™ specification.

Example 2.5. A typical example for a *pure* method is *getBalance()* in Line 19, since it is a getter method. Those methods let the callee observe a specific attribute of the object state, in that case the *balance*, but do not change it. Therefore, *getBalance()* can be used in the postcondition of *withdraw(...)* in Line 14. □

Cheon et al. [Cheon et al., 2005b] introduced the concept of model fields. They define a model field as a "specification-only field that describes the abstract state of some program fields." It is similar to common class fields, but can only be used for specification purposes. A model field abstracts type and name from the real implementation. Therefore, it allows one to write specifications independently of the implementation. It is declared through the @*Model* keyword. The @*Represents* keyword actually links the model field to an implementation. Depending on the Design by Contract™ specification language, any valid expression may be accepted - not only a single field reference.

Example 2.6. The only attribute of *Account* is *balance*. It is stored in a protected field, which should not be exposed to clients of *Account*. Therefore, *balance* is abstracted by means of a model variable *mBalance* defined as an integer variable in Line 1. Line 2 links the model field to the actual value, in this case the value of the private field *balance*. □

Specification Inheritance

One of the most important features of object-oriented programming languages is inheritance, also called structural subtyping. Inheritance allows one to pass subtypes of the requested object instead of the original type. Therefore, the behavior of a program might change at runtime depending on which kind of object is passed. In the presence of behavioral specification, such as Design by Contract™, one has to make sure that no contract is violated when passing a subtype. Liskov worked on the so called behavioral subtyping and defines the subtype relation by means of three rules [Liskov and Wing, 1994]:

- invariant rule: properties of a supertype are preserved by the subtype
- methods rule
  - precondition: subtype's method can be called at least in any state required by the supertype
  - postcondition: subtype's method postcondition can be stronger than the supertype method's postcondition
- constraint rule: supertype constraints have to be preserved

Those rules translate to the following logical interpretations for Design by Contract™, checked by the runtime assertion checker.

The invariant of a subtpye may be strengthened.

$$I_{super} \wedge I_{sub}$$

The precondition can be weakened.

$$P_{super} \vee P_{sub}$$

The postcondition can be strengthened.

$$Q_{super} \wedge Q_{sub}$$

Based on Liskov's work Dhara and Leavens introduced the notion of weak and strong behavioral subtypes. It basically extends subtyping to return values and exceptions thrown.

Some Design by Contract™ specification languages, such as *JML* [Leavens and Cheon, 2003] and *Modern Jass* [Rieken, 2007] extend the concept of visibility modifiers to the specification as well. The semantics of those modifiers is equivalent to applying those modifiers to any other supported concept of the corresponding programming language. In other words, a private specification entry is not inherited. Only protected and public entries are inherited.

	invariant	precondition	postcondition
method parameter	✗	✓	✓
member variable	✓	✓	✓
methods	✓	✓	✓
return value	✗	✗	✓

Table 2.1.: Specification Scope. The language elements that can be used in Design by Contract™ specifications.

Specification Syntax

This section deals with the syntactical elements of the Design by Contract™ specification that all mature Design by Contract™ specification languages have in common: (*a*) what logical expressions are allowed, and (*b*) what elements can be referenced inside a specification. Everything stated in this section applies to invariants, pre- and postconditions in similar ways.

Design by Contract™ specifications are executable boolean expressions. Thus, their syntax is equal to the used programming language. Each boolean expression can be build up of multiple clauses that are combined either by means of conjunction or disjunction. Furthermore, Design by Contract™ specifications support quantifiers as defined by first-order logic [Ammann and Offutt, 2008, p.104]. Each clause in turn can contain comparisons between hard-coded values, language features (variables, methods, ...) that are in the scope of the expression, or even method calls. Table 2.1 shows which language features can be used in invariants, preconditions and postconditions. The table does not include restrictions due to modifiers such as *public*, *protected*, *private*, and *static*. These modifiers influence the access of the specification.

2.1.2. Overview of Design by Contract™ Languages

The following overview of mature Design by Contract™ specification languages is based on a technical report of Andreas Maller [Maller et al., 2010], which was conducted under my supervision.

2.1.3. Eiffel

The Eiffel programming language is an object-oriented programming language developed by Bertrand Meyer. Eiffel focuses on building tools and techniques as part of the programming language that ensure software quality. Meyer defines software quality into the following factors [Meyer, 1997]:

- Reliability: The produced software should be robust and work correctly.
- Re-usability: The different components of a software are designed and implemented that they can be re-used in other projects.
- Portability: The software is platform independent, which means the software can work on different systems.
- Extendibility: The software can be extended if required.

```
class  ACCOUNT

balance :INTEGER

feature {ANY}
  deposit(amount :INTEGER) is
    require
      amount>0
      amount<MAX_INT-balance
    do
      -- implementation
    ensure
      balance = old balance + amount
  end

  getBalance() :INTEGER
    do
      -- implementation
    ensure
      Result = balance
  end

invariant
  balance >= 0
end
```

Figure 2.2.: Eiffel specification for *Account*. An excerpt from the well-known *Account* case study emphasizing pre- and postconditions, an invariant, pre-state access, and the return value access in a postcondition.

- Maintainability: The software is designed and implemented in a way that allows an effective and easy way of software maintenance.

The Design by Contract™ software paradigm [Meyer, 1992b, Meyer, 1997] is part of the Eiffel language. At compile time the specifications are translated to assertions that are stored as byte code with the class. Once compiled the assertion statements are evaluated at runtime. The Eiffel compiler can be configured to ignore different levels of Design by Contract™ specifications and assertions. The following six levels are available [Meyer, 1992b]:

1. off: any assertion checking is disabled.
2. precondition: evaluation of the keyword require.
3. postcondition: evaluation of the keyword ensure in addition to all level 2 checks.
4. invariant: evaluation of class invariants in addition to all level 3 checks.
5. loop invariant: evaluation of all loop specifications in addition to all level 4 checks
6. evaluation of the check instruction in addition to all level 5 checks

Figure 2.2 shows an Eiffel specification of the *Account* case study (see Section 2.1.1) used by many publications.

Pre- and postconditions in Eiffel start with the keyword *require* and *ensure*, respectively. Each method may only be annotated with a single pre- or postcondition. On the other hand each of those specifications may extend over multiple lines. Each line is combined by conjunction. The *old* keyword in Line 13 indicates a pre-state access to *balance*.

In case an Eiffel method has multiple behaviors, the Design by Contract™ specification has to be written accordingly. Eiffel does not provide a shortcut keyword such as *JML* or *Modern Jass* do. Furthermore, Eiffel does not support specifications for exceptional behavior.

Eiffel invariants are written at the end of a class body and start with the keyword *invariant*. Consistent to pre- and postconditions, there might only be one *invariant* keyword per class and multiple lines of invariant specifications are conjuncted. The invariant is checked after the constructor and before and after each method call. It is possible to expose invariants during the execution of a method body.

Eiffel does not extend the concept of element visibility to the specification. Therefore, any member variable or method may be called in a specification [Meyer, 1997].

Eiffel follows the general inheritance rules of Section 2.1.1.

In addition to basic pre- and postconditions, Eiffel supports loop invariants and variants. A loop invariant [Meyer, 1997] states the condition, which is satisfied in each iteration. The variant [Meyer, 1997] states the termination criteria. In other words, the variant statement has to decrease with every loop iteration.

2.1.4. Spec#

Spec# is an extension to the well-known C# programming language [Gunnerson, 2000]. Both are developed by Microsoft Corporation.

Spec# [Barnett et al., 2005] provides constructs for specifying non-null types, preconditions, postconditions, and invariants. The Spec# specifications are written in the C# source code files. At compile time the specifications are converted to CIL (Common Intermediate Language) which is Microsoft's .NET byte code format. The C# compiler then transforms the CIL file into an executable. Runtime violations of specifications cause exceptions to be thrown.

Pre- and postconditions are part of the C# method body and start with the keyword *requires* and *ensures*, respectively. Multiple pre- and postcondition sections are allowed in Spec#, which are interpreted as logical conjunctions. The keyword *otherwise* allows to specify a different exception that is thrown on violating the corresponding specification. For example, Line 9 in Figure 2.3 specifies that instead of the default exception indicating a precondition violation a *NegativeAmountNotSupportedException* is thrown on contract violation.

The postcondition in Line 11 features a pre-state access of *balance*, marked with the call to *old()*. Line 17 uses the *result* keyword to access the return value of *getBalance()*.

Spec# invariants [Barnett et al., 2005] are checked after the constructor, before and after each method execution, and after an *expose* block. The *expose* block allows a method to temporarily violate the invariants.

```
 1 public class Account
 2 {
 3   invariant balance >= 0;
 4   protected int balance = 0;
 5
 6   public virtual int deposit(int amount)
 7     modifies balance;
 8     requires amount > 0
 9       otherwise NegativeAmountNotSupportedException
10     requires amount < MAX_INT - balance
11     ensures balance == old(balance) + amount
12   {
13   }
14
15   [Pure]
16   public virtual int getBalance()
17     ensures result == balance;
18   {
19   }
20 }
```

Figure 2.3.: Spec# specification for *Account*. An excerpt from the well-known *Account* case study emphasizing pre- and postconditions, an invariant, pre-state access, and the return value access in a postcondition.

Spec# does neither provide a keyword for easily specifying multiple behaviors, nor for specifying exceptional behavior [Barnett et al., 2005].

In addition to the basic Design by Contract™ specifications, Spec# provides a mechanism to specify loop invariants and to deal with the framing problem.

The loop invariant in Spec# [Barnett et al., 2004] is part of the loop syntax. The following quantifiers are supported by the loop invariant:

forall all elements have to satisfy the given condition

exists at least one element has to satisfy the given condition

exists unique exactly one element has to satisfy the condition

The *modifies* and *[Pure]* keywords as seen in Line 7 and Line 15, respectively, can be used to specify what values do not change during method execution. Even though this type of specification is not evaluated at runtime due to performance reasons [Barnett et al., 2005]. The keyword *modifies* is part of the method body and takes a list of variables that do not change their value. Methods marked with *[Pure]* do not change any value that effect the internal state of the object.

In Spec# specification elements inherit the visibility of their parent elements, for example, the methods visibility level for pre- and postconditions. Accessing elements of a stricter visibility level is forbidden. Therefore, Spec# provides the *SpecPublic* keyword to increase the visibility level of a private class member.

Subclasses inherit the specification from their super classes and interfaces. Spec# forbids adding new preconditions. Additional postconditions can be added to overriding methods. This approach ensures that a subclass never violates the contract of one of its parents.

2.1.5. Code Contracts

Code Contracts [Cooperation, 2009, Barnett et al., 2009] is a spin-off project of Spec# at Microsoft Research. It is a library that provides Design by Contract™ functionality to all .NET languages. It is not only a research project but already seamlessly integrates with Microsoft Visual Studio 2008 or higher. The Code Contracts bundle contains a static contract and a runtime checker.

The static contract checker evaluates assertions and Design by Contract™ specifications of called methods statically. However it does not support the whole Design by Contract™ functionality of Code Contracts [Cooperation, 2009].

The runtime checker evaluates preconditions, postconditions and invariants dynamically. Pre- and postcondition specifications may appear anywhere in the method body. That feature may reduce readability for human beings, but has no influence on the runtime checking abilities of Code Contracts. The compiler reassembles all specifications at the CIL. The Microsoft Visual Studio compiler has different levels of runtime checking:

REQUIRE_ALWAYS Just the *RequireAlways* statements are checked during runtime.

CONTRACTS_PRECONDITIONS The preconditions and *RequireAlways* statements are checked during runtime.

CONTRACTS_FULL Every precondition, postcondition and invariant is checked during runtime.

Since Code Contracts is a library for all .NET languages, it provides a set of methods in the *Contract* namespace that allows Design by Contract™ specifications to be written [Cooperation, 2009]. The precondition is stated by passing an assertion to *Requires*. Multiple preconditions, as given in Lines 8 and 9 in Figure 2.4, are conjuncted.

An interesting feature of Code Contracts is the possibility to state preconditions with *if(...) throw* statements. This syntax support was added to support backward capability with existing source code, where typically the first lines of a method body included all argument checks. Figure 2.5 shows the logically equivalent precondition from Figure 2.4 but with the alternative precondition syntax. The *EndContractBlock* in Line 5 has to be present so that the compiler can distinguish between precondition if statements and actual if statements in the program.

The postcondition is passed to the *Ensures()* method in the *Contracts* namespace. A postcondition can reference the (*a*) methods return value, (*b*) the methods out parameter values, and (*c*) the pre-state value of any variable. *Contract.Result<T>()* refers to the return value of the method. *Contract.ValueAtReturn()* access the value of the requested out parameter. The `Contract.OldValue<T>()` statement accesses the value of the requested variable

Code Contracts provides two ways of specifying a method to check the invariants. In both cases the invariant clauses are stated in the method body by means of calling the *Invariant* method in the *Contract* namespace. The first one is to implement a method with the following signature: `protected`

```
public class Account
{
  invariant balance >= 0;
  protected int balance = 0;

  public virtual int deposit(int amount)
  {
    Contract.Requires(amount > 0);
    Contract.Requires(amount < MAX_INT - balance);
    Contract.Ensures(balance == Contract.OldValue(balance) + amount);
    // implementation goes anywhere
  }

  public virtual int getBalance()
    Contract.Ensures(Contract.Result<int>() == balance);
  {
  }

  protected void ObjectInvariant()
  {
    Contract.Invariant(balance>=0);
  }
}
```

Figure 2.4.: Code Contracts specification for *Account*. An excerpt from the well-known *Account* case study emphasizing pre- and postconditions, an invariant, pre-state access, and the return value access in a postcondition.

```
  public virtual int deposit(int amount)
  {
    if(amount > 0) throw new ArgumentException();
    if(amount < MAX_INT - balance) throw new ArgumentException();
    Contract.EndContractBlock();
    Contract.Ensures(balance == Contract.OldValue(balance) + amount);
    // implementation goes anywhere
  }
```

Figure 2.5.: Code Contracts support an *if() throw* syntax to specify preconditions due to backward compatibility issues with software written before Design by Contract™ support.

`void ObjectInvariant()`. The second is to annotate any given method with the *[ContractInvariantMethod]* attribute. The invariants are checked before and after the execution of a method. It is possible to dispose invariants of a class in a method body. Therefore, the corresponding class has to implement the *System.IDisposable.Dispose* interface [Cooperation, 2009].

Similar to Spec# Code Contracts does not provide any special keyword for specifying multiple behaviors. In contrast to Spec#Code Contracts provides a way to specify exceptional behavior by means of the *EnsuresOnThrow<T>()* method call.

Code Contracts provides the *[Pure]* annotation for methods that do not change the object state. Furthermore, all get properties are marked as *[Pure]* by default.

In addition to the mentioned Design by Contract™ features Code Contracts distinguishes between assertions and assumptions. *Contract.Assert()* statements are only checked statically. *Contract.Assume()* statements are only evaluated at runtime. Both statements can appear anywhere in a method body.

Code Contracts do not care about visibility attributes. Therefore, all class member can be accessed in a specification regardless of their visibility. Code Contracts uses the same inheritance rules as Spec#.

2.1.6. *JML*

JML [Leavens et al., 1998] is an interface behavioral specification language for Java introduced by Gary T. Leavens. *JML* allows the separation of source code and specification. Contracts may be written in a different file than the actual source code. *JML* provides its own Java compiler that generates the Java class file which includes all Design by Contract™ assertions.

JML is a very extensive specification language and therefore provides two sets of specifications:

- lightweight, and
- heavyweight [Leavens et al., 2002].

Lightweight specifications do not support multiple behaviors, exceptional behaviors, and ignore visibility levels of the specification. A heavyweight specification supports the full range of available *JML* modifiers.

The pre- and postcondition in *JML* start with the *requires* and *ensures* keyword, respectively. Multiple pre- or postconditions in the same *behavior* are conjuncted, as given in Lines 7 and 8 in Figure 2.6 [Leavens et al., 1998].

JML provides a number of special keywords that can be used in a postcondition clause. *\result* refers to the return value of the method. *\old* provides access to the pre-state of the passed variable.

The runtime assertion checker evaluates the *JML* invariants after a constructor call, and before and after any method call [Leavens et al., 2002]. This is similar to all other presented approaches. Furthermore, *JML* supports loop invariants through the *maintaining* keyword. It is checked at the beginning of each iteration of the loop. Similar to Eiffels *variant* declaration, *JML* supports *decreases* specification blocks. The control variable declared in the *decreases* block has to be of type integer or long. At the first iteration it has to be greater or equal to zero. With each iteration it has to decrease. Therefore, termination of the loop can be checked.

```
1  public class Account
2  {
3    /*@ invariant balance >= 0; @*/
4    protected int balance = 0;
5
6    /*@ assignable balance;
7      @ requires amount > 0;
8      @ requires amount < MAX_INT - balance;
9      @ ensures balance == \old(balance) + amount;
10     @*/
11   public virtual int deposit(int amount)
12   {
13   }
14
15   /*@ ensures \result == balance; @*/
16   public virtual int getBalance()
17   {
18   }
19 }
```

Figure 2.6.: *JML* specification for *Account*. An excerpt from the well-known *Account* case study emphasizing pre- and postconditions, an invariant, pre-state access, and the return value access in a postcondition.

JML uses the keyword `also` to implement multiple behavior. In other words, each method may have assigned more than one pre-/postcondition block. Each of which can contain multiple *requires* and *ensures* clauses. Furthermore, *JML* distinguishes between *normal_behavior* and *exceptional_behavior* specification blocks. The *signals* clause of the *exceptional_behavior* block defines the exceptions that may be thrown when executing the method.

JML also supports specification of framing conditions by means of, *pure* and *assignable*. All fields listed in the *assignable* clause may be modified during the method execution. Line 6 in Figure 2.6 specifies that only *balance* may be changed during method execution. This clause has different semantics if lightweight or heavyweight *JML* is used. In lightweight mode the default value is that everything may change. In heavyweight mode the default value is that nothing may change.

JML supports the concept of specification visibility. To be able to access private member variables, one has to define model or ghost fields [Cheon et al., 2005b, Leavens et al., 2002]. Those are specification only fields of the class. They may only be accessed in specifications. Their value at runtime may depend on the actual value of a field, or any other calculation expression.

JML follows the standard rules for inheritance as pointed out in Section 2.1.1.

JML provides much more specification concepts. One may even write abstract types, that are only used in specifications. Details can be found on the up-to-date *JML* homepage[*].

[*]http://www.eecs.ucf.edu/ leavens/JML/

```
@Model(name=''mBalance'', type=Integer.class)
@Represents(name=''mBalance'', by=''balance'')
public class Account
{
  @Invariant(‘‘mBalance>=0’’)
  protected int balance = 0;

  @Pre(‘‘amount>0 && amount<MAX_INT-mBalance’’)
  @Post(‘‘mBalance==@Old(mBalance)+amount’’)
  public virtual int deposit(int amount)
  {
  }

  @Post(‘‘@Result == balance’’)
  public virtual int getBalance()
  {
  }
}
```

Figure 2.7.: *Modern Jass* specification for *Account*. An excerpt from the well-known *Account* case study emphasizing pre- and postconditions, an invariant, pre-state access, and the return value access in a postcondition.

2.1.7. *Modern Jass*

Modern Jass [Rieken, 2007] is another approach to bring Design by Contract™ to Java. *Modern Jass* specifications are written by means of Java 5 annotations. There are three major differences to *JML*:

1. *Modern Jass* specifications are written by means of Java 5 annotations.
2. *Modern Jass* produces a JAR file that contains all contract information in addition to the Java byte code.
3. *Modern Jass* specifications focus on basic Design by Contract™ concepts.

The first two differences can be considered as advantages, since they improve the automation and ensure that the source code is not influenced by contracts if they are disabled. The third difference is a disadvantage, since the expressiveness of *JML* is much higher.

Figure 2.7 shows an example specification for the *Account* class. *Modern Jass* provides the annotations *@Pre()* and *@Post* to specify preconditions and postconditions, respectively. The *@Invariant* annotation specifies an invariant. Each annotation may only be used once per method. This limitation is imposed by the Java 5 annotation framework. Future releases of Java will remove this restriction.

The *@Old()* and *@Result* keyword in a postcondition reference to the pre-state value of the passed variable and the return value of the method, respectively.

Multiple behaviors can be expressed by using the *@Also* annotation, which can hold an array of *@SpecCase*s. A *@SpecCase* annotation in turn contains a single precondition and postcondition. The *@SpecCase* annotation can also be used to specify an exceptional behavior.

Modern Jass implements the concept of specification visibility. Specifications on public methods may not access non-public members. Instead one has to introduce a model field by means of a *@Model* annotation. Each model field has to be represented by an actual field, or calculation expression in each non-abstract class. Therefore, *Modern Jass* provides the *@Represents* annotation. Whenever the model field referenced by its name is accessed while evaluating a specification, the *by* clause of the corresponding *@Represents* annotation is used to calculate the actual value.

Modern Jass implements the standard inheritance rules for all specifications as explained in Section 2.1.1 and provides some more syntactic sugar keywords, such as *@Length*, *@Max*, or *@Min*.

2.2. Satisfiability Modulo Theories (SMT)

A satisfiability modulo theories (SMT) solver basically is a satisfiability solver which integrates additional theories to be able to directly reason over complex data types.

A satisfiability solver determines if any given satisfiability problem is solvable. It does this by searching a value assignment for all given predicates, such that the overall problem is satisfied. Lardeux defines a satisfiability problem as a pair (Ξ, Φ), where Ξ is a set of Boolean variables and Φ is a Boolean propositional formula [Lardeux et al., 2008]. The problem is satisfiable if there exists an assignment for each variable in Ξ with *true* or *false* such that Φ is satisfied.

The output of a SAT solver traditionally is only either *true* or *false*. In case the problem is satisfiable, most SAT solver can also present a variable assignment that satisfies the problem. This variable assignment is also called model [Bordeaux et al., 2006].

De Moura et al. defines a *theory* as a set of sentences [de Moura and Bjø rner, 2009]. A propositional formula Φ is satisfiable modulo a given theory T, if $T \cup \Phi$ is satisfiable. In other words, a theory contains symbols and manipulation rules on them for different concepts. State-of-the-art SMT solver include theories for linear arithmetic, difference arithmetic, non-linear arithmetic, free functions, bit vectors, arrays, pairs, tuples, lists, and strings. Therefore, they are able to solve not only problems given over the Boolean domain but problems given in any domain as long as a corresponding theory is given. The bit vector theory can be used to more generally model integer data types. Each bit in the vector represents a bit of the data type, therefore variables for 16-, 32-, or 64-bit integer types can be used. The theory can be reduced to Boolean satisfiability by simply introducing a Boolean variable for each bit. The string theory is related to the array and list theory. The array theory itself can be reduced to the free function theory. Having those theories by hand improves efficiency when doing any kind of automated software analysis, including testing.

Two different approaches for SMT solver exist [Nieuwenhuis et al., 2007]:

1. eager evaluation
2. lazy evaluation

The former approach transforms the SMT problem to a SAT problem and preserve satisfiability. This process is very sophisticated and suffers on practical problems such as running out of memory.

The latter approach combines state-of-the-art T-solvers and SAT solvers. Initially, each atom is treated as Boolean variable. In the case the formula is unsatisfiable the original problem is unsatisfiable as well. Does the SAT solver return a model then it is checked by the T-solvers. If no model exists that also satisfies the T-solver, a ground clause is added to the original problem and the process is started again. The added ground clause prohibits the SAT solver form returning the same model again, which was already shown to not be T-satisfiable.

It is well known that SAT solving is NP-complete and first-order logic is undecidable as pointed out by de Moura et al. [de Moura and Bjø rner, 2009]. For example, non-linear arithmetic is decidable for quantifier-free problems over real but not over integer variables. Despite these theoretical limits, SMT solver have improved in the last decades. SMT solver are often used in different applications and research areas, such as engineering, software and hardware verification, financial applications, traffic and logistics, which pays-off the effort in improving this technology.

Two mature SMT solver are yices [Dutertre and de Moura, 2006] and Z3 [de Moura and Bjø rner, 2008]. *Yices* supports a rich combination of first-order theories that occur frequently in software and hardware modeling: arithmetic, uninterpreted functions, bit vectors, arrays, recursive datatypes, and more. The Z3 SMT solver integrates a modern DPLL-based SAT, a core theory solver that handles equalities and uninterpreted functions, satellite solvers for arithmetic and arrays, and an E-matching abstract machine for quantifiers [de Moura and Bjø rner, 2008].

2.3. State-of-the-Art White- and Gray-Box Test Generation Tools

This section attempts to give an overview of currently available commercial and academic tools with respect to their test data generation capabilities. It is based on a survey submitted to the STTT journal [Galler and Aichernig, 2010]. The compiled list of all 19 tools under consideration are presented in Figure 2.8. The tools are filtered with respect to their level of *availability*, *maturity*, and *activity*. The remaining seven tools, i.e., AgitarOne, CodePro AnalytiX, AutoTest, C++test, Jtest, RANDOOP, and PEX, are challenged with in total 31 benchmark tests: 24 benchmark tests show the tools capabilities to generate primitive values; seven benchmark tests show how well they perform on non-primitive types and complex specifications.

2.3.1. Evaluation Procedure

The following paragraphs (*a*) show a classification of all candidate tools, (*b*) introduce the selection criteria *availability*, *maturity* and *activity*, (*c*) introduces the evaluation criteria, and (*d*) describes the evaluation procedure.

Candidate Tools

Figure 2.8 presents the map of all relevant tools on automatic test generation. The tools are categorized with respect to two dimensions:

1. source code required/present
2. specification usage

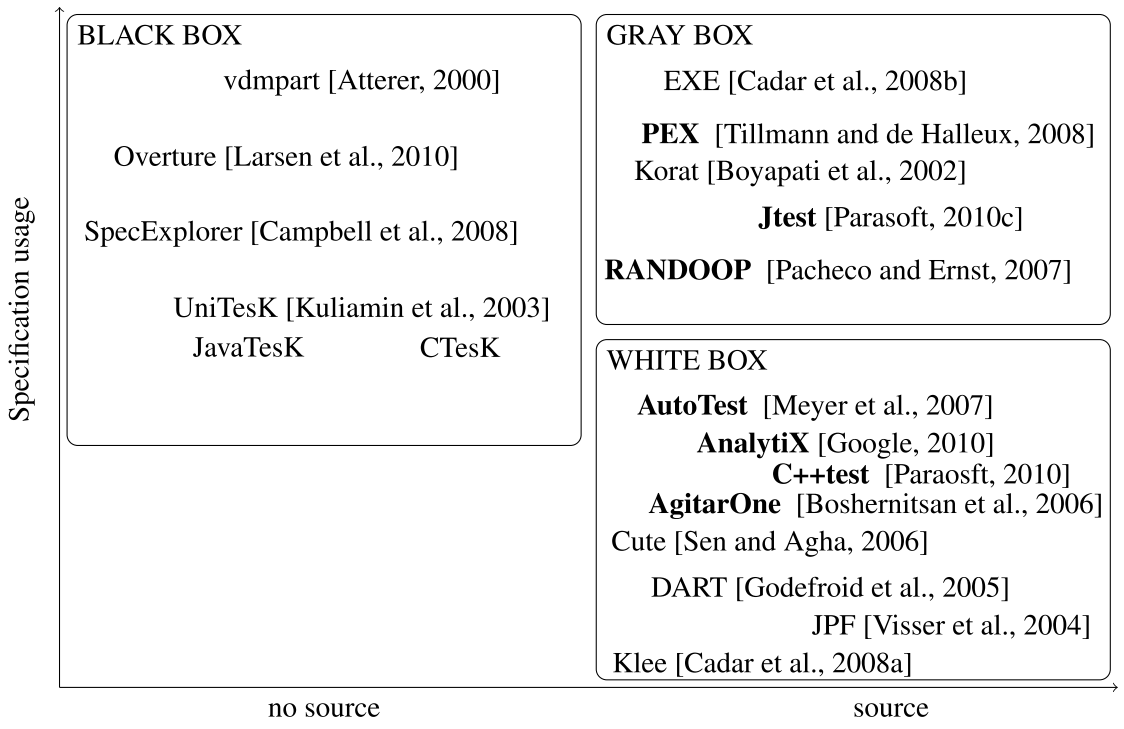

BLACK BOX
vdmpart [Atterer, 2000]
Overture [Larsen et al., 2010]
SpecExplorer [Campbell et al., 2008]
UniTesK [Kuliamin et al., 2003]
JavaTesK
CTesK
GRAY BOX
EXE [Cadar et al., 2008b]
PEX [Tillmann and de Halleux, 2008]
Korat [Boyapati et al., 2002]
Jtest [Parasoft, 2010c]
RANDOOP [Pacheco and Ernst, 2007]
WHITE BOX
AutoTest [Meyer et al., 2007]
AnalytiX [Google, 2010]
C++test [Paraosft, 2010]
AgitarOne [Boshernitsan et al., 2006]
Cute [Sen and Agha, 2006]
DART [Godefroid et al., 2005]
JPF [Visser et al., 2004]
Klee [Cadar et al., 2008a]
Specification usage
no source
source

Figure 2.8.: Tool Classification of Evaluated Tools. The highlighted tools are those that satisfied the required criteria to be part of the evaluation.

On the one hand we distinguish tools with respect to their access to source code. On the other hand we distinguish between tools that use no specification, use specification as test oracle only, and tools that use specification as test oracle as well as for steering the test input generation.

This survey focuses on state-of-the art test data generating tools. To ensure the quality of this survey we have to further filter the candidate list. First, only white-box or gray-box testing tools are considered for this survey. Second, the remaining tools are rated with respect to *availability*, *maturity*, and *activity*.

availability Tools have to be publicly available. Either as free download or as commercial tool.

maturity Only tools that are already applied to industrial size applications are considered. We therefore rate all tools from 1 to 4:

1. commercial tool
2. applied to (at least one) industrial size case study
3. applied to (at least one) case study
4. no information about case studies available

activity Tools have to be maintained. In other words, only tools updated in the last two years (i.e., in 2009 and 2010) are considered.

citation The amount of (scientific) publications that include references to the tool. Figures are extracted from Google scholar in October 2010.

tool	avail.	maturity	activity	citations
AgitarOne	✓	1	2010	50
AnalytiX	✓	1	2010	none
AutoTest	✓	1	2010	26
Check'n'Crash	✓	4	2005	99
C++test	✓	1	2010	0
Cute	✓	3	2006	395
DART	✗	3	2005	569
Eclat	✓	3	2005	118
EXE	✗	3	2008	265
Klee	✓	3	2009	117
Jcrasher	✓	4	2007	179
JPF	✓	3	2010	817/198*
Jtest	✓	1	2010	0
Korat	✓	3	2007	382
PEX	✓	1	2010	111
RANDOOP	✓	1	2010	136
SAGE	✗	4	2008	162

Table 2.2.: Automated Test Data Generation Tools Candidate List. Tools highlighted satisfy the listed criteria to be part of the evaluation.

Table 2.2 lists all white- and gray-box testing tools and summarizes the rating with respect to the introduced classification criteria. AgitarOne, CodePro AnalytiX, AutoTest, C++test, Jtest, RANDOOP, and PEX satisfy the criteria and are therefore highlighted in the table.

Evaluation Criteria

This survey aims for a uniform comparison and evaluation of the state-of-the art test data generation tools. Therefore, in Section 2.3.2 each tool is shortly introduced, its test data generation technique is explained in detail, and finally the evaluation result is presented and discussed.

The short introduction of the tool is summarized in a table that includes information on input and output of the tool, supported programming and specification languages, licensing issues and the user pace.

Additionally, we summarize in a table the particular test data generation techniques incorporated by the tool. This summary includes the following attributes:

approach primitive types What approaches are used to generate values for Boolean, byte, character and integer types (e.g., random, constraint solver)?

approach non-primitive types What approaches are used to generate instances for all object types?

*original JPF publication/test data generation publication

type	specification (arithmetic expression)			
	constant	simple linear arithmetic	non-linear arithmetic	inequality
Boolean	$a = true$	$a = b$	-	$a! = false$
character	$a =' b'$	-	-	$a >' b'$
integer	$a = 3$	$a = 3+5$	$a = 3*5$	$a > 5$
float	$a = 3.2f$	$a = 3.2f + 5.1f$	$a = 3.2f * 5.1f$	$a > 5.1f$
double	$a = 3.2d$	$a = 3.2d + 5.1d$	$a = 3.2d * 5.1d$	$a > 3.2d$
string	$a ='' abcd''$	$a! = null$ && $a.matches(''[a-z][0-9]+'')$	-	$a! = null$ && $a! = ''''$ && $a! ='' abcd''$

Table 2.3.: Type support tests. Each of the given specifications is used as precondition for a method.

test	paramters	specification
parameter dependencies	int a, String b	$b! = null$ && $b.Length = a$ && $a > 32$
null object	a:Stack	$a = null$
object type	a:Stack	$a! = null$ && $a.Count >= 2$
array type	a:int[]	$a! = null$ && $a.length > 2$ && $a[1] = 1$
forall quantifier	a:List$< String >$	$a! = null$ && $a.Count = 2$ && $\forall s \in a : s = ''abc''$
exists quantifier	a:List$< String >$	$a! = null$ && $a.Count = 2$ && $\exists s \in a : s = ''abc''$
scalene triangle example [Myers et al., 2004]	a,b,c:double	$a+b > c$ && $b+c > a$ && $a+c > b$ && $a! = b$ && $a! = c$ && $b! = c$

Table 2.4.: Structural tests. Each of the given specifications is used as precondition for a method.

specification usage In what sense does the approach use given specification (e.g., as oracle only, to steer input data generation)?

specification dependencies Is the approach able to deal with specifications where one parameter value depends on another, e.g., $param1 > param2.size()$?

quantifiers Can the approach deal with quantifiers in the specification?

object pooling Does the tool store already instantiated objects for later reuse?

manual objects Is it possible to add manually constructed objects to the tool?

special values Which hard-coded values does the tool use (e.g., min and max value of a data type)?

equivalence Does the tool check for equality, and if so how?

The actual evaluation of the selected tools is based on analysing automatically generated tests for the given benchmarks. To find out the limitations of each tool we use a structured set of benchmark tests. The benchmark tests are based on the different problem divisions of the annual SMT solver competition [SMTCOMP, 2010]: (*a*) integer difference logic, (*b*) real difference logic, (*c*) linear integer arithmetic, (*d*) linear real arithmetic, (*e*) nonlinear integer arithmetic, and (*f*) (quantified) arrays.

In addition we added nonlinear real arithmetic and similar expressions for Boolean, character and string types. Furthermore, we added the following tests that explicitly test specification dependencies with respect to parameters:

dependencies between parameters One parameter depends in any attribute from the value of another parameter.

requested null objects Explicitly requesting a *null* parameter.

object type parameters Explicitly requesting an object in a given state.

triangle example The specification for a scalene triangle [Myers et al., 2004].

Tables 2.3 and 2.4 show the specifications used for each of the benchmark tests. The evaluation of the tools should find out, which tool is able to generate data that satisfies the given specification. Therefore, we implemented for each benchmark test a method that requires input values that satisfy the corresponding specification. We stated the requirement either as precondition or as assertion in the first line of the benchmark method. Tools that are able to execute the return statement of the benchmark method are able to call it with input data that satisfies the benchmark tests specification. An example benchmark method implementation is presented for each tool.

2.3.2. Evaluation

The presentation order of the evaluation results are determined by the time of publication of the original paper or launch of the tool.

Jtest

Jtest is a comprehensive testing product of Parasoft [Parasoft, 2010b] for Java first introduced in 1997. It supports development teams in building new or improving quality of legacy Java applications likewise. Jtest facilitates static analysis, runtime analysis, code review process automation and unit testing. Static analysis includes coding standard checks, data flow analysis and common well-known coding style rules. Runtime analysis mainly provides different kind of coverage information, detects race conditions and security attack vulnerabilities. The code review process is supported through notification, documentation and tracking functionalities. In this evaluation we focus on the unit testing support of Jtest.

Jtest supports automatic generation of *JUnit* tests and automatic generation of regression tests. It can be used with and without Design by Contract™ specifications. In the former case, the methods postconditions are used as oracle. Furthermore, only tests that satisfy the precondition are exported to *JUnit* test files. In the latter case, Jtest uses thrown exceptions and assertion errors as oracle. For regression tests, the initial test execution run determines the expected return values, which are recorded and used in further execution runs. Jtest incorporates a powerful test data generation engine, which features are discussed in detail in Subsection 2.3.2. Furthermore, Jtest includes a tracing facility, which can be used to capture and replay interaction of Java applications with remote clients and servers.

Table 2.5 summarizes Jtest's general information. Jtest does not depend on the presence of Design by Contract™ specification, but it improves test quality and reduces test effort if present. It is a commercial tool with a 14-days evaluation license and comes with an extensive documentation.

institution	Parasoft
version (tool)	8.4
source code language	Java
specification language	Jcontract by Parasoft, very similiar but less expressive than *JML*
required input	Source code
optional input	Design by Contract™ specification in Jcontract syntax
output	JUnit test classes
introduced in	1997
last updated in	2010
user pace	Parasoft has more than 10.000 worldwide customers, 58% of Fortune 500 companies
license/price	Commercial license, 14-days evaluation license
documentation	Comprehensive user manual (ca. 750 pages), well structured with examples and step-by-step tutorials
test classification	All tests that satisfy the precondition are exported.

Table 2.5.: Jtest: General Information

Data Generation Approach Jtest mainly operates on an object repository. This repository is pre-populated with test data for all primitive types: e.g., the minimum and maximum value, 0, -1, +1 are instances of the value pool for the integer type. The pool can be manually populated with values.

Jtest uses those values and tries possible combinations for a given method under test. In addition, Jtest includes some more sophisticated value generation strategies for primitive as well as for non-primitive data types. It is not documented which technologies are used for generating primitive values that satisfy a given precondition, but our evaluation showed that Jtest was able to generate values that were not initially in the pool.

Furthermore, Jtest automatically generates stub objects. A stub is an object that overrides the real objects implementation and returns only hard coded values [Google, 2008]. Jtest stubs are able to return different values each time a method is called.

In case the value combination satisfies the precondition of the method under test the test is executed, the result is recorded and a *JUnit* test method is exported. The postcondition is evaluated and violations are reported.

Table 2.6 summarizes Jtests different strategies for test data generation.

Evaluation Figure 2.9 shows an example implementation of a benchmark method. For the evaluation we executed the Jtest command *"Generate Unit Tests"* followed by *"Run Unit Tests (Report All Severities)"*. The result is a set of *JUnit* tests and a report containing coverage information.

Table 2.7a summarizes the result of the evaluation on the set of test data benchmark problems. Jtest managed to generate valid input data for all benchmarks. Therefore, we conclude that Jtest incorporates some more sophisticated technologies than mentioned in the official documentation.

Table 2.7b shows the results of the structural benchmark tests. Jtest is able to generate values even for specifications that include variables where one variable is constrained by the value of another

approach primitive	Combinations of predefined values are candidates (integer: 0, -1, 1, 10, -10, ..., MAX int, MIN INT), but as it can be seen in the test results, more sophisticated technologies are present (not documented which)
approach non-primitive	Stubs are created for all external resources, such as databases, third-party libraries, file system and network I/O. For EJBs Jtest even provides a dummy container.
approach uses specification	Yes, details are not published, but manual inspection of all generated tests let us assume that there exists some mechanism to use the specification for test data generation
parameter dependency	Yes
object pooling	Jtest integrates an object repository which is populated by Jtest itself
manual objects	Yes, the object repository may be populated with manually generated objects
special values	Jtest pre-populates the object pool with special values, such as maximum and minimum value of a data type
equivalence (test data)	Not checked.
quantifiers	Not supported.

Table 2.6.: Jtest: Test Data Generation Details

```
/**
  * @pre (a==3.2f * 5.1f)
  * @post ($result == true)
  */
  public boolean floatNonLinear(float a)
  {
    return true;
  }
```

Figure 2.9.: Jtest: Example Evaluation Criteria Method. Jtest was only able to generate valid tests after adding the type specification postfix *f* to all float constants.

type	constraint			
	constant	linear	non-linear	inequality
Boolean	✓	✓	-	✓
character	✓	-	-	✓
integer	✓	✓	✓	✓
float	✓	✓	✓	✓
double	✓	✓	✓	✓
string	✓	✓	-	✓

(a) Results of Data Type Benchmark Tests

test	result
parameter dependencies	✓
null object	✓
object type	✓
array type	✓
forall quantifier	✗
exists quantifier	✗
scalene triangle example	✗

(b) Results of Structural Tests

Table 2.7.: Jtest results summary.

institution	Parasoft
version (tool)	7.3
source code language	C/C++
specification language	No specification
required input	Source and binary
optional input	
output	Unit tests
year	2010
user pace	Parasoft has more than 10.000 worldwide customers, 58% of Fortune 500 companies
license/price	Commercial license with 14-day trial
documentation	Well documented (tutorial, user manual, step-by-step tutorial, examples)
test classification	All tests are exported.

Table 2.8.: C++test: General Information

variable. The approach does not work for more complex dependencies as imposed by the scalene triangle specification, or quantifiers.

C++test

Parasofts C++test [Paraosft, 2010] is a commercial software quality improvement tool for C/C++, introduced in 1999. It comes as stand alone IDE or Eclipse plugin. C++test supports coding standard checks, static analysis, runtime analysis, and automates code review and unit test generation. C++test incorporates best practice rules such as those proposed by Meyers [Meyers, 2005]. C++test can be seen as the little brother of Jtest. Both have very similar feature lists, but due to advanced technologies provided by Java, e.g., Java reflection, Jtest implements more sophisticated data generation techniques than C++test.

Table 2.8 summarizes C++tests general information. C++test does not support any kind of specification. It is best used for generating regression tests, which detect changes in the systems under test behavior over time.

Test Data Generation C++test does not support any form of specification but simple assertion statements. Therefore, it cannot classify test input data in *meaningful* and *meaningless*. As a consequence, all generated combinations of test input data are exported as unit tests.

Based on the evaluation we can identify two different test data generation strategies: one for primitive and one for non-primitive types. For primitive types C++test selects randomly a value of

- a pre-defined pool of values, such as minimum and maximum value, -1, +1 and 0 for integer types, a string value that has more than 256 characters,
- the path of the file containing the method under test,
- the method under tests signature,
- constant values present in the method under tests body.

approach primitive	C++test chooses a value from a pool of random values, pre-defined constants, constants extracted from source, and other values, such as the methods signature and the source files path.
approach non-primitive	A public constructor of the requested class is used for instantiation.
approach uses specification	No.
parameter dependency	C++test generates combinations of randomly chosen values. Not all combinations are exported.
object pooling	Yes, by means of manually written factory methods.
manual objects	Yes, the manually written factory methods for the pooling can instantiate any object instance.
special values	Pre-defined set of values, i.e., string that contains more than 256 characters, min/max values for the given data type
equivalence (test data)	Not checked.
quantifiers	Not applicable, since no specification is supported.

Table 2.9.: C++test: Test Data Generation Details

```
1  bool floatNonLinear(float a)
2  {
3    assert(a == 3.2f * 5.1f);
4    return true;
5  }
```

Figure 2.10.: C++test: Example Evaluation Criteria Method

A non-primitive value is always constructed by means of a constructor call. In case a method requires multiple parameters, random combinations are generated. Manual inspection of the generated tests shows that not all combinations are generated. It is not documented, which combinations are generated.

C++test does not explicitly use any form of an object pool. But it is able to use manually written factory methods. These methods allow to establish a repository of valid object instances that are used in automatically generated tests. Furthermore, C++test supports stubs: user-defined and automatically generated stubs. It only generates stubs if no user-defined version is available. If it is not able to generate a complete stub definition, it will generate a template which has to be customized manually.

The summary of C++tests data generation techniques is given in Table 2.9.

Evaluation Since C++test does not support any Design by Contract™ specification we implemented all benchmark problems by means of assertion statements as can be seen in Line 3 Figure 2.10. Note that we could not evaluate the string linear benchmark test due to the missing native regular expression support of C++.

For each of those benchmark methods we let C++test generate unit tests. Therefore, we followed Parasofts recommendation for automatically generating and executing unit tests [Paraosft, 2010]:

type	constraint			
	constant	linear	non-linear	inequality
Boolean	✓	✓	-	✓
character	✓	-	-	✓
integer	✓	✓	✓	✓
float	✓	✓	✓	✓
double	✓	✓	✓	✓
string	✓	-	-	✓

(a) Results of Data Type Benchmark Tests

test	result
parameter dependencies	✗
null object	✗
object type	✓
array type	✗
forall quantifier	✗
exists quantifier	✗
scalene triangle	✗

(b) Results of Structural Tests

Table 2.10.: C++test results summary.

1. generate test cases
2. generate stubs
3. build test executable
4. execute test cases

C++test generated more than 200 tests. Most of them fired the assertion statement and were therefore classified as *meaningless*. But some parameter combinations successfully passed the assertions. Tables 2.10a and 2.10b summarize the evaluation result.

For the character benchmark tests C++test used the integer number of the character (i.e., the character *b* is represented by the integer value 98 in the ASCII format) given in the specified assertion and the off by one values (i.e., 97 and 99). Furthermore, it created two tests that passed the maximum and minimum character value, respectively, to the method under test. These two simple techniques for data generation enabled C++test to pass all character benchmarks.

C++test generated nine tests for the integer constant benchmark. Besides the special predefined values already mentioned in the general description (0, -/+1, max/min value) C++test uses the values in the assertion along with the off by one values. Furthermore, C++test uses the result of mathematical expressions present in the source code. For example, C++test generated tests that pass eight ($3+5$ taken from the linear integer specification) and 15 ($3*5$ taken from the non-linear integer specification) to the method under test.

The predefined values for *float* and *double* type values include in addition to the already mentioned values from the integer domain, the minimum negative and positive value. But the resulting test cases do not include any slightly modified values by means of mathematical operation, such as the off by one values for integer and character types.

For non-primitive types C++test always chooses a constructor. Therefore, C++test is not able to generate test input that is required to be *NULL*. Furthermore, C++test does not try to modify the object any further, i.e., no other methods are called after the constructor to change the object state. This conclusion is based on our empirical evaluation of C++test. The C++test User's Guide [Paraosft, 2010] does not say anything about the object creation strategy.

C++test failed also on the array and quantifier benchmark tests. For the array it passed *null*, which did

institution	Agitar Technologies
version (tool)	5.1.0.000021
source code language	Java
specification language	None. AgitarOne can handle normal Java assertion statements, and even suggests assertions based on dynamic source code analysis.
required input	Source code.
optional input	
output	A set of *JUnit* tests, including coverage information. Furthermore, AgitarOne provides a management dashboard to clearly structure and summarize all related information.
introduced in	2004
last update in	2010
user pace	Hundreds of organizations worldwide, from Global 100 to Silicon Valley startups.
license/price	Commercial tool with 30-day trial version
documentation	Scientific publication, installation guideline
test classification	Al tests are exported.

Table 2.11.: AgitarOne: General Information

not satisfy the specification. C++test has the technologies at hand to pass the quantifier benchmark tests, but unfortunately it did not use the required combination of parameter values.

AgitarOne

In 2004 Agitar Technologies released AgitarOne, a commercial tool based on academic research results. AgitarOne can be used as standalone IDE, as Eclipse add-on, or from the command-line. It includes automated *JUnit* test generation, code rule enforcement technologies and a management dashboard. Furthermore, it suggests assertions it revealed while dynamically analyzing the software under test.

In our evaluation we focus on the automated *JUnit* test generation feature. Since AgitarOne does not base its analysis on a specification language, it misses an oracle for all generated tests. Therefore, AgitarOnes is useful for automatically generating a regression test suite that captures the current behavior of the software under test.

During test generation AgitarOne collects observations of the code's behavior. Those observations are similar to invariants detected by Daikon [Ernst et al., 1999]. In fact, Daikon and AgitarOne use very similar techniques, which were developed independently at about the same time. The user can then decide whether those observations should be *promoted* to assertions. Thus, AgitarOne helps the software engineer to write specification by means of Java assertions. Those assertions are used in later iterations of the test generation process.

AgitarOne includes a mocking library and is able to run automatically generated and hand-written *JUnit* tests side-by-side. It is a good example for transforming academic research into a usability friendly and scaling commercial product.

```
public boolean floatNonLinear(float a)
{
  assert a == 3.2f * 5.1f;
  return true;
}
```

Figure 2.11.: AgitarOne: Example Evaluation Criteria Method

Data Generation Approach Agitar Technologies coins the term *agitation* [Boshernitsan et al., 2006] for the process of test data generation. It includes: (*a*) static and dynamic analysis of the software under test, (*b*) automatic input generation, (*c*) exercising the code based on the generated input, and (*d*) collecting observations by means of mathematical relationships between variables.

Static and dynamic analysis focus on collecting path constraints. A constraint solving system then provides required input data to generate tests that steer the execution along a specific path. In all those analysis steps, AgitarOne focuses on performance and scalability. Thus, AgitarOne prefers a fast "good guess" over a correctly calculated value which requires more time. The following paragraphs are based on the publication 'From Daikon to Agitator' [Boshernitsan et al., 2006] and a manual inspection of the generated *JUnit* tests for all benchmarks.

For all numerical types, i.e., *integer*, *float*, *double*, AgitarOne uses all constants found in the source code, the negation of them and the constants +/- a delta value from the original constant. For example, AgitarOne finds the integer constant five in the source code, then it uses the constants four, five and six as test input. For double values, it uses a delta of 0.001.

AgitarOne provides a string solver that can handle constraints imposed by the string API. This solver enables AgitarOne to generate string values that satisfy a regular expression specified through the *matches(...)* method of the *java.lang.String* class. Furthermore, AgitarOne uses *NULL*, the empty string, any string constants from the source code and random combinations of alphanumeric values.

Object types are generated by randomly calling a constructor and zero or more (state-changing) methods of the requested type. Required arguments are generated recursively. All generated objects are kept in a pool to be modified and reused at a later stage in the test generation process. Furthermore, AgitarOne includes a mocking library and enhanced technologies to specify the expected behavior of those mock objects. It records the interaction with the mock object and generates a unit test that expects the same behavior.

Evaluation We implemented all benchmark tests by means of Java assertions and check if AgitarOne is able to generate tests that cover the return statements after the assertion. If so AgitarOne generated a value that satisfies the specification. Figure 2.11 shows one of the implemented benchmark methods.

AgitarOne generated 112 *JUnit* tests. Tables 2.13a and 2.13b show that AgitarOne passed all benchmark tests. AgitarOnes capability to generate tests for the *forall* and *exists* benchmarks is very impressive. Due to the lack of a supported specification language we had to manually write the quantifiers by means of Java assertions (see Figure 2.12). The automatically generated *JUnit* test is presented in Figure 2.13. AgitarOne creates a new array and adds random values (Figure 2.13, Lines 5- 6). Finally,

approach primitive	In addition to constraint solver AgitarOne uses constants found in the source code, and values close to them by means of mathematical addition and subtraction operations.
approach non-primitive	Randomly generating objects by calling constructor and other methods of the requested type. Furthermore, AgitarOne includes a mocking library.
approach uses specification	AgitarOne has no specification language support, but is able to use Java assertions to steer the generation process.
parameter dependency	Values with dependencies between each other can be generated as long as they are specified by means of Java assertions.
object pooling	Yes, AgitarOne uses factories for each type, which behave similar to object pool.
manual objects	AgitarOne allows manual refinement of test input data, and provides factories for each data type, which can also be manually adapted to return manually specified objects. Furthermore, AgitarOne provides a pattern - strategy technology, which allows to specify which generation strategy should be used for different automatically detected patterns in the source code.
special values	AgitarOne uses pre-defined values, such as min/max value for each type.
equivalence (test data)	Not checked.
quantifiers	AgitarOne does not have a special specification language, but understands Java assertions very well, even those in loops.

Table 2.12.: AgitarOne: Test Data Generation Details

type	constraint			
	constant	linear	non-linear	inequality
Boolean	✓	✓	-	✓
character	✓	-	-	✓
integer	✓	✓	✓	✓
float	✓	✓	✓	✓
double	✓	✓	✓	✓
string	✓	✓	-	✓

(a) Results of Data Type Benchmark Tests

test	result
parameter dependencies	✓
null object	✓
object type	✓
array type	✓
forall quantifier	✓
exists quantifier	✓
scalene triangle	✓

(b) Results of Structural Tests

Table 2.13.: AgitarOne results summary.

```
1  public boolean Exists(List<String> a)
2  {
3    assert a!=null && a.size() == 2;
4    boolean exist = false;
5    for(int i=0; i<a.size(); i++) {
6      if(a.get(i).equals("abc")) {
7        exist = true;
8      }
9    }
10   assert exist == true;
11   return true;
12 }
```

Figure 2.12.: AgitarOne: Exists benchmark.

```
1  public void testExists()
2      throws Throwable {
3    List arrayList = new ArrayList(100);
4    boolean add =
5      arrayList.add((Object) null);
6    arrayList.add("testString");
7    arrayList.set(0, "abc");
8    boolean result =
9      new Benchmark().Exists(arrayList);
10   assertTrue("result", result);
11 }
```

Figure 2.13.: AgitarOne: Generated test for the *for all* benchmark.

it determines that it has to set at least one element in the array to “abc” (Figure 2.13, Line 7) to satisfy the assertion statement in Line 10 of Figure 2.12.

The *forall* benchmark test looks very similar to the *exists* benchmark test, but it asserts in each iteration that the current value has to be equal to “abc”. However, the generated *JUnit* test features AgitarOnes *Mockingbird* mock library. Figure 2.14 shows the generated test. Wherever complex objects have to be constructed, AgitarOne replaces the actual object with a mock object (Line 3). The Lines 6 to 16 define the behavior of the mock object. For each expected method call the return value is defined. Furthermore, the sequence of the method calls is defined and asserted.

In this case, the *ArrayList* has a *size* of two, and will return “abc” for both calls of *get* - once with argument 0, once with argument 1. This generated test satisfies the assertion.

Note, sometimes AgitarOne used the mock library for the *exists* test too. This let us conclude that some nondeterministic approaches are used.

```
public void testForAll() throws Throwable {
 Benchmark benchmark = new Benchmark();
 ArrayList a = Mockingbird.getProxyObject(ArrayList.class);
 Mockingbird.enterRecordingMode();
 Mockingbird.setReturnValue(false,a,"size","()int",
                            new Object[] {},new Integer(2),1);
 Mockingbird.setReturnValue(false,a,"size","()int",
                            new Object[] {},new Integer(2),1);
 Mockingbird.setReturnValue(false,a,"get","(int)java.lang.Object",
                            new Object[] {new Integer(0)},"abc",1);
 Mockingbird.setReturnValue(false,a,"size","()int",
                            new Object[] {},new Integer(2),1);
 Mockingbird.setReturnValue(false,a,"get","(int)java.lang.Object",
                            new Object[] {new Integer(1)},"abc",1);
 Mockingbird.setReturnValue(false,a,"size","()int",
                            new Object[] {},new Integer(2),1);
 Mockingbird.enterTestMode(Benchmark.class);
 boolean result = benchmark.forAll(a);
 assertTrue("result", result);
}
```

Figure 2.14.: AgitarOne: Generated test for exists benchmark.

AutoTest

AutoTest [Ciupa and Leitner, 2005] started as research tool at ETH Zürich and is by now part of the commercially available Eiffel Studio. Eiffel Studio is the integrated development environment for Eiffel. AutoTest automatically generates tests for Eiffel programs. It uses different random based approaches for generating test input data. AutoTest exports only tests that reveal an error. Furthermore, AutoTest implements two different minimization algorithm to reduce the amount of exported tests.

Table 2.14 summarizes the general information of AutoTest. It is limited to the Eiffel programming language with its built-in support for Design by Contract™ specifications. Eiffel has very prominent clients as listed, but it is not known which of those use AutoTest as well.

Data Generation Approach AutoTest implements a random based test data generation approach. It uses the Design by Contract™ specification as oracle only. In other words, AutoTest generates test input first and then checks whether it satisfies the precondition of the method under test or not.

AutoTest has two slightly different approaches [Meyer et al., 2007] for generating primitive and object type input data.

primitive types For the Eiffel primitive types *INTEGER*, *BOOLEAN*, *CHARACTER*, and *REAL* AutoTest maintains a list of preset values for each type. Candidate values for the *INTEGER* type are, e.g., minimum and maximum value as well as 0, -1, +1, -2, +2, -10, +10. On request it randomly chooses one of those values.

institution	ETH Zürich, Eiffel Incorporation
version (tool)	6.6.8.3355 GPL
source code language	Eiffel
specification language	Eiffel
required input	Source code and Design by Contract™ specification. The specification is required because AutoTest exports only tests that cause a postcondition or invariant violation.
optional input	Test generation can be customized through a configuration file.
output	A set of Eiffel test classes (inheriting from EQA_TEST_SET).
introduced in	2005
last update in	2010
user pace	Group of researcher at ETH and worldwide customers such as AXA Rosenberg Investment Management, Boing, EMC^2.
license/price	Both, commercial and open source license.
documentation	Online documentation, scientific publications
test classification	Through the configuration file different minimization algorithms can be activated to reduce the amount of exported tests.

Table 2.14.: Eiffel: General Information. The years mentioned in the table refer to AutoTest, the test generation tool of Eiffel Studio, not Eiffel Studio itself.

object types AutoTest maintains a pool of already created objects for each type. On request it randomly chooses one of the existing object instances from the pool. A predefined probability defines how often (in case of an empty pool always) a new instance for the requested type is generated and added to the pool. Furthermore, again with a preset frequency AutoTest chooses randomly an instance from the pool and calls modifier features (state changing methods) on it to diversify the pool.

Whenever a new instance has to be created AutoTest executes the following steps [Meyer et al., 2007]:

1. choose one of the creation procedures (constructors) of the class
2. generate values for all arguments, recursively
3. call the creation procedure with those arguments

Table 2.15 summarizes all analyzed aspects of AutoTests test data generation technologies.

Since there is a very close connection between Eiffel Software Inc. and ETH Zürich, research initiatives eventually become part of Eiffel Studio. Two AutoTest features recently developed at ETH Zürich are 'Adaptive Random Testing for Object-Oriented Software' [Ciupa et al., 2008] and 'Satisfying Test Preconditions through Guided Object Selection' [Wei et al., 2010].

The former enhances the random selection process of values from the pool. Instead of randomly selecting a value, it selects the one value with the highest distance to all already selected values in previous iterations. The distance of two integer values is their mathematical difference. The distance function of objects takes recursively the distance of all members and the distance in the inheritance hierarchy into account. Details are explained by Ciupa et al. [Ciupa et al., 2008].

The latter enhances the object pool by replacing it with a map from specification predicates to objects. For each object it is recorded which predicates it satisfies. Therefore, the pool can deliver objects that

approach primitive	Randomly choosing one value from a list of predefined values.
approach non-primitive	Randomly generating instances through calls to the public interface of the type (enhanced random approaches such as ARTOO are supported as well).
approach uses specification	Specification is not used for test data generation, only as test oracle.
parameter dependency	Not applicable, since specification is not used at test data generation time.
object pooling	Yes, objects are stored for later reuse. Extensions exist that improve the pool by means of remembering which precondition predicates the object has already satisfied.
manual objects	Yes, one can add manually generated values to the pool.
special values	The pool is pre-filled with values, e.g., min/max value for each type.
equivalence (test data)	Equivalent tests can be ignored. Two different equivalence definitions, which are called minimization strategies in AutoTest, are supported: (1) slicing (2) ddmin.
quantifiers	Eiffel does not provide quantifier keywords.

Table 2.15.: AutoTest: Test Data Generation Details

```
floatNonLinear(a :REAL) :BOOLEAN
  require
    a = 3.2 * 5.1
  do
    Result := true
  ensure
    Result = false
end
```

Figure 2.15.: AutoTest: Example Evaluation Criteria Method

will likely satisfy the given precondition in case similar preconditions are given for multiple methods in the same system under test.

Evaluation Figure 2.15 shows the implementation syntax of one of the benchmark methods in Eiffel. *require* and *ensure* are the Eiffel keywords for specifying a methods pre- and postcondition, respectively. Note, AutoTest only exports test cases that violate the postcondition. Therefore, we implemented all methods such that they cause a postcondition exception, i.e., all methods return *true* and the postcondition requires *false*. All test cases that satisfy the precondition will fail on the postcondition and therefore get exported as unit tests.

AutoTest generated 117 tests. Tables 2.16a and 2.16b summarize the results.

AutoTest was able to generate valid test input for all inequality tests. Since each specification consists of only one inequality expression, the likelihood to select a value different than the one given in the specification is very high.

type	constraint			
	constant	linear	non-linear	inequality
Boolean	✓	✓	-	✓
character	✗	-	-	✓
integer	✓	✓	✗	✓
float	✗	✗	✗	✓
double	✗	✗	✗	✓
string	✗	-	-	✓

(a) Results of Data Type Benchmark Tests

test	result
parameter dependencies	✗
null object	✓
object type	✗
array type	✗
forall quantifier	✗
exists quantifier	✗
scalene triangle	✗

(b) Results of Structural Tests

Table 2.16.: AutoTest results summary.

Furthermore, AutoTest was able to generate tests for the constant and linear integer benchmark. In both cases it generated exactly the required value which let us assume that AutoTest may include some more sophisticated approaches for integer values than random. For all other tests AutoTest failed, which is reasonable because it is very unlikely to generate the value *16.32* randomly, which for example is required to satisfy the non linear float benchmark.

CodePro AnalytiX

Google, Inc. bought CodePro AnalytiX from Instantiations, Inc. in 2010. Along with the change in ownership, the previously commercial tool became publicly available under Apache License 2.0.

CodePro AnalytiX is a tool that helps to improve the quality of Java programs. It seamlessly integrates into Rational Developer, IBM WebSphere Studio and Eclipse [Google, 2010]. CodePro AnalytiX includes - as all commercial tools - a rich set of metrics and a user-friendly reporting of them. Furthermore, CodePro AnalytiX is able to find similar code snippets in the system under test and can check the source code against security and style conventions. In the following we focus on CodePro AnalytiXs capabilities of automated *JUnit* test generation.

CodePro AnalytiX provides a rich set of configuration possibilities such as (*a*) which parts of the project should be tested? (*b*) how many tests should be generated? (*c*) if tests that cause an exception should be exported? (*d*) where the generated tests should be saved? For each method under test CodePro AnalytiX

- generates input values for all parameters,
- determines combinations,
- computes the result of executing the method under test,
- validates the result, and
- generates *JUnit* test files.

Typically, not all combinations of generated test input data can be tested, due to limited resources. Therefore, CodePro AnalytiX includes some rules to reduce the amount of combinations to a reasonable level. Afterwards, the result of executing the method under test with the determined set of

institution	Google Inc.
version (tool)	7.0.0
source code language	Java
specification language	Java assertions at the beginning of a method are interpreted as precondition. Furthermore, CodePro AnalytiX claims to support Design by Contract™ specification in Java comments with a syntax similar to Jtest. Unfortunately, it did not work for us.
required input	Source code.
optional input	Java assertion statements or a tool specific Design by Contract™ specification.
output	*JUnit* tests
introduced in	before 2007
last updated in	2010
user pace	Unknown
license/price	Apache License 2.0
documentation	It exists only a general overview in PDF and HTML format. In addition, a user forum is maintained.
test classification	All tests are exported.

Table 2.17.: CodePro AnalytiX: General Information

combinations is calculated. CodePro AnalytiX records the result value of a non-void method and all thrown exceptions and determines how it can check these results in the *JUnit* test. Finally, exporting the result to *JUnit* test files is straight forward.

CodePro AnalytiX claims to support simple Design by Contract™ specifications for class invariants, and method pre-/postconditions in JavaDoc comments. Unfortunately, we could not see any different results in terms of generated tests when adding Design by Contract™ specification. Manually writing a test that definitely violated the contract of the test did not result in any Design by Contract™ specific violation message. Thus, we conclude that Design by Contract™ support is not working currently.

Data Generation Approach Only a few details on the test data generation approach are available. CodePro AnalytiX analyzes the method under test to determine the usage of the parameters. Based on that analysis CodePro AnalytiX tries to generate values that help to explore the different behaviors of the method. For example, if an integer parameter is used in a switch statement, then it uses each of the values explicitly listed in non-empty case labels as well as some values that are not in any of the case labels [Google, 2010].

In case CodePro AnalytiX does not find any values in this first phase it uses pre-defined default values for all well known types. Well known types are all primitive types and non-primitive types such as *java.lang.String*.

For all other cases, CodePro AnalytiX searches, in the given order, for zero-argument static accessor methods, constructors and multi-argument static accessors. It uses the first entry found to instantiate an object of that type. Values required as arguments are generated recursively.

CodePro AnalytiX features *EasyMock* [Freese, 2002]. *EasyMock* is a well-known mock library, which

approach primitive	CodePro AnalytiX analysis the usage of the parameter in the method under test and tries to generate values accordingly. In case this approach fails, pre-defined values are used for all known types (e.g., integer, string, ...).
approach non-primitive	For non-primitive types CodePro AnalytiX calls zero-argument static accessors, constructors and multi-argument static accessors.
approach uses specification	Design by Contract™ specification support did not work for the evaluation, but the approach filtered values that did not satisfy Java assertions.
parameter dependency	Worked. Furthermore, CodePro AnalytiX includes heuristics to prune the set of all possible parameter value combinations.
object pooling	No
manual objects	Yes, through *Factories* the user can provide specific instances that should be used as test input data.
special values	Yes, CodePro AnalytiX uses pre-defined values.
equivalence (test data)	No
quantifiers	No specification support for those quantifiers, but they can be written as Java assertions.

Table 2.18.: CodePro AnalytiX: Test Data Generation Details

provides easy instantiation of mock objects and their configuration of the expected behavior. CodePro AnalytiX can be configured to use *EasyMock* objects for all interfaces by default. In addition, one can manually specify which classes should be mocked as well.

Evaluation Results We started our evaluation with Design by Contract™ specifications as claimed in the documentation [Google, 2010]. Unfortunately, we could not see any different error messages when adding Design by Contract™ specifications. Therefore, we added again assertion statements in the first line of each benchmark method. Figure 2.16 shows the *floatNonLinear* benchmark test method, including the Design by Contract™ specification that did not work, and the Java assertion statement in Line 7.

The generated test suite of in total 78 tests was able to satisfy most of the benchmark tests. Tables 2.19a and 2.19b summarize the evaluation result.

CodePro AnalytiX satisfies all primitive constant, linear and non-linear benchmark tests due to the fact, that the required input data is present as constants, or mathematical operations on constants in the method under tests source code. For example, CodePro AnalytiX is able to generate the input value 16.32 for the *floatNonLinear()* benchmark given in Figure 2.16, since it finds the constant term $3.2 * 5.1$ and the result satisfies the assertion statement.

The inequality benchmarks are not satisfied due to the same reason. To satisfy those specifications, the result has to be modified slightly by means of a mathematical addition operation. But CodePro AnalytiX only uses exactly the constants present in the source.

For a similar reason CodePro AnalytiX does not perform very well on the structural benchmarks, which mostly deal with object type parameters. After calling the constructor of the object type CodePro AnalytiX does not call any further methods on it. Therefore, it does not change the initial state

```java
/**
 * @pre a==3.2f*5.1f
 * @post $result==true
 */
public boolean floatNonLinear(float a)
{
  assert a == 3.2f*5.1f;
  return true;
}
```

Figure 2.16.: CodePro AnalytiX: Example Evaluation Criteria Method

type	constraint			
	constant	linear	non-linear	inequality
Boolean	✓	✓	-	✓
character	✓	-	-	✓
integer	✓	✓	✓	✓
float	✓	✓	✓	✗
double	✓	✓	✓	✗
string	✓	✓	-	✗

(a) Results of Data Type Benchmark Tests

test	result
parameter dependencies	✓
null object	✓
object type	✗
array type	✓
forall quantifier	✗
exists quantifier	✗
scalene triangle	✗

(b) Results of Structural Tests

Table 2.19.: CodePro AnalytiX results summary.

of the object, which in turn does then not satisfy the precondition of the method under test. The same reason prevents CodePro AnalytiX from generating tests for the *forall* and *exists* benchmark.

No constants are present in the scalene triangle benchmark test. Therefore, CodePro AnalytiX uses the set of pre-defined values only.h They are not sufficient to find a combination to pass the scalene triangle benchmark test.

RANDOOP

Pacheco et al. introduced in 2007 RANDOOP [Pacheco and Ernst, 2007, Pacheco et al., 2007]. In 2008 he ported the Java version to .NET and used it internally at Microsoft to test a very important component of the .NET framework [Pacheco et al., 2008].

RANDOOP is a tool implementation of *feedback–directed random testing*, which addresses random generation of unit tests for object-oriented programs. A non-primitive type is created by building a method sequence. Each generated method sequence is immediately executed to ensure that only non-redundant and legal objects are used. Two objects are redundant if their construction sequences are equivalent. In other words, if the generated code for two sequences modulo variable names is equal. An object is legal if it satisfies all *contracts* and *filters*. *Contracts* are methods that use the current state of the system and return either *violates* or *satisfies*. User can write *contracts* by implementing a class that inherits from *randoop.UnaryObjectChecker*. In addition, RANDOOP provides a default set of contracts, such as *NullPointer* occurences and *assertion* violations. Furthermore, for objects RANDOOP checks if *o.equals(o)* holds and methods such as *equals()*, *hashCode()*, and *toString()* do not throw any exception.

Data Generation Approach RANDOOP is a test generation tool for object-oriented programs. Therefore, it incorporates only weak data generation techniques for primitive types.

primitive types RANDOOP selects randomly an element from the pool. In the current implementation the pool contains a small set of primitives:

- Boolean: true, false
- char: 'a', '4'
- byte, integer: -1, 0, 1, 3, 10, 100
- float: $0.0f$, $1.0f$, $10.0f$, $100.0f$
- double: $0.0d$, $1.0d$, $10.0d$, $100.0d$

object types For object types RANDOOP uses either *NULL*, or uses a sequence from the pool. New sequences are generated by combining two sequences from the pool with *m* calls to a randomly selected method. Candidate methods are public methods of the corresponding class. RANDOOP adds *m* calls to the existing sequence, since especially container classes often require more than one element in the container. Therefore, it makes sense to call for example *add(...)* multiple times in a row. A newly generated sequence is executed to determine that it is not redundant and constructs an object not violating any contracts.

institution	MIT CSAIL
version (tool)	1.3.2
source code language	Java, .NET
specification language	Contracts and filters
required input	Assembly.
optional input	List of user-defined contracts and filters, and a configuration file that specifies limits with respect to time, amount of tests generated, and length of tests generated.
output	Unit test suite of all passing and/or failing test cases.
introduced in	2007
last updated in	2010
user pace	Unknown
license/price	MIT license
clients	Microsoft
documentation	Scientific publications of the technique (i.e., *feedback-directed random testing*), the Java and .NET tools including case study reports.
test classification	Each test case is executed and is classified as error-revealing, passing or illegal. Only error-revealing and passing tests can be exported (user defines, which of them should). Furthermore, equivalent test input data, with respect to the objects *equals(...)* method, is skipped.

Table 2.20.: RANDOOP: General Information

approach primitive	Selects a value from a fixed pool of values.
approach non-primitive	Either use null, or an existing sequence from the pool.
approach uses specification	It uses *contracts* and *filters* to check the constructed sequence before it gets executed on the method under test or exported to a unit test.
parameter dependency	Not applicable, since specification is given in terms of *contracts*. And RANDOOPs *contracts* are methods, that take the current state of the system and return *satisfied* or *violated*.
object pooling	Yes.
manual objects	Yes.
special values	Yes, the pool of primitive values is, e.g., populated with -1, 0, 1, 'a', *true*, and others.
equivalence (test data)	Yes, RANDOOP avoids to equal sequences. Two sequences are equivalent if they translate to the same code, modulo variable names [Pacheco et al., 2007, p.3].
quantifiers	Not applicable, since no formal specification used. (see parameter dependency)

Table 2.21.: RANDOOP: Test Data Generation Details

```
public boolean floatNonLinear(float a)
{
  assert a == 3.2f * 5.1f;
  return true;
}
```

Figure 2.17.: Example Evaluation Criteria Method

type	constraint			
	constant	linear	non-linear	inequality
Boolean	✓	✓	-	✓
character	✗	-	-	✗
integer	✓	✗	✗	✓
float	✗	✗	✗	✓
double	✗	✗	✗	✓
string	✗	✗	-	✓

(a) Results of Data Type Benchmark Tests

test	result
parameter dependencies	✗
null object	✓
object type	✓
array type	✓
forall quantifier	✗
exists quantifier	✗
scalene triangle	✗

(b) Results of Structural Tests

Table 2.22.: RANDOOP results summary.

Evaluation Results We evaluated the Eclipse plugin of RANDOOP for Java. The .NET implementation is equivalent to the Java implementation. RANDOOP per default uses Java assertion statements to filter sequences that generate illegal object states. Therefore, we implemented our benchmark methods by means of Java assertions, as can be seen in Figure 2.17.

In addition to the class containing the benchmark methods, we told RANDOOP to use *java.util.ArrayList*, *java.util.LinkedList* and *java.util.Stack* and set the null object generation probability to 0.3. Otherwise, RANDOOP does not know them but they are required by some of the benchmark tests. The results did not improve when we increased the default timeout from 100 seconds to 300, or even 1000 seconds.

RANDOOP targets mainly the challenge of testing object-oriented programs. It is therefore obvious that it does not very well perform on any primitive type benchmarks. Table 2.22a summarizes the expected weak performance of RANDOOP on the primitive benchmark tests. Comparing the inequality benchmark specifications with the primitive values in the pool (see Section 2.3.2), shows that the pool contains at least one element for each type that satisfies the specification, but for the character type. For the character type the pool contains only an ’a’, and the benchmark expression requires a character greater than ’b’.

More interesting are the structural benchmark tests that include more object types. Table 2.22b summarizes RANDOOPs performance on this set of benchmark tests. Unfortunately, RANDOOP did not perform that well either. We expected that RANDOOP cannot satisfy the specifications for the *parameter dependencies* and the *scalene triangle* benchmarks. Those two test require sophisticated primitive value generation capabilities.

Manually inspecting why RANDOOP did not pass the two quantifier benchmarks, revealed that it was able to generate *java.util.List* objects with enough elements, but never with the expected values. This can be reduced to RANDOOPs weak primitive value generation capabilities.

PEX

Microsoft Research started a few years ago the development of PEX, a white-box test generation tool for .NET [Tillmann and de Halleux, 2008]. Meanwhile it is not only a research tool but part of the Visual Studio 2010 Power Tools that help unit testing .NET applications. PEX started as a tool that creates a test suite, which achieves high branch coverage based on dynamic symbolic execution. Today, it perfectly incorporates other tools and research results. PEX uses the Z3 [de Moura and Bjø rner, 2008] SMT solver for solving the path constraints collected during dynamic symbolic execution; It uses REX [Veanes et al., 2010] for generating string values specified by means of regular expressions; PEX supports Code Contracts [Barnett et al., 2009], which is a Design by Contract™ specification language for .NET; and features Moles [de Halleux and Tillmann, 2010], which is a light-weight mocking library from Microsoft Research. Furthermore, PEX fully integrates with Microsoft Visual Studio.

Table 2.23 summarizes the general information about PEX. It works on the intermediate language of .NET so it can be used for testing programs in any .NET language. Note, currently only unit tests for C# can be exported.

The core of PEX is a test data generator. It supports not only test data generation for Design by Contract™ specification but features specifications as parameterized unit tests [Tillmann and Schulte, 2005] as well. Parameterized unit tests are unit test methods that have parameters. In other words, a parameterized unit test specifies the behavior of the method under test for all possible input values. One specific parameter combination is equivalent to a traditional unit test.

Data Generation Approach PEX starts with simple random input for a given method under test. While executing the method PEX collects runtime information, e.g., symbolic values for all variables and path constraints. At each condition statement PEX collects information about the branching criteria. PEX re-executes the method with input values that satisfy all path conditions. This process is called dynamic symbolic execution [Cadar et al., 2008a, Godefroid et al., 2005]. It is also known as concrete symbolic (concolic) execution [Sen et al., 2005].

Therefore, it is able to explore all feasible path of the method under test. The values are calculated by passing the path constraint to the Z3 SMT solver. Z3 is able to solve constraints on propositional logic, fixed-sized bit-vectors, tuples, arrays and quantifiers. Arithmetic constraints over floating point numbers are approximated by a translation to rational numbers. Heuristic search techniques are used outside of Z3 to find approximated solutions for floating point constraints [Microsoft-Research, 2010]. Recently, PEX integrated REX, a technology for generating string values that are formalized by means of regular expressions [Veanes et al., 2010].

Implementation details regarding the instrumentation process for symbolic execution, and the symbolic representation of values, pointers and objects can be found in a lot of technical reports and publications of Microsoft Research [Tillmann and de Halleux, 2008, Microsoft-Research, 2010].

Evaluation Results To evaluate PEX we implemented one method for each evaluation critiera. Its method body consists of a single *return true* statement. Figure 2.18 shows an example implementa-

institution	Microsoft Research
version (tool)	0.91 on Visual Studio 2010 Ultimate
source code language	Theoratically any .NET language, but test export is currently only available for C#.
specification language	PEX supports specification in terms of parameterized unit tests or Code Contracts.
required input	Source code.
optional input	A specification.
output	Unit tests/parameterized unit tests in one of the supported unit test framework formats (Visual Studio Unit Test, NUnit, Mb Unit, xUnit.net).
introduced in	2008
last updated in	2010
user pace	At least 10 research cooperations with world-wide research institutions, open source community.
license/price	Academic and commercial license (Microsoft Visual Studio 2010 Power Tools Software Terms).
documentation	Well documented at different technical levels including step-by-step tutorials and scientific publications.
test classification	Configurable what tests should be exported.

Table 2.23.: PEX: General Information

approach primitive	Z3 SMT solver [de Moura and Bjø rner, 2008] and REX [Veanes et al., 2010] for string values
approach non-primitive	Z3 and REX: objects are encoded as maps of their members as in ESC/-Java [Flanagan et al., 2002]
approach uses specification	PEX does not only use Code Contracts specification but also collects all path constraints so that it is able to generate a test input data set that achieves high branch coverage.
parameter dependency	Yes, encoded in SMT constraint.
object pooling	No
manual objects	No
special values	No
equivalence (test data)	No
quantifiers	Support of forall and exists.

Table 2.24.: PEX: Test Data Generation Details

```
public bool floatNonLinear(float a)
{
   Contract.Requires(a == 3.2f * 5.1f);
   return true;
}
```

Figure 2.18.: Example Evaluation Criteria Method

type	constraint			
	constant	linear	non-linear	inequality
Boolean	✓	✓	-	✓
character	✓	-	-	✓
integer	✓	✓	✓	✓
float	✓	✓	✓	✓
double	✓	✓	✓	✓
string	✓	✓	-	✓

(a) Results of Data Type Benchmark Tests

test	result
parameter dependencies	✓
null object	✓
object type	✓
array type	✓
forall quantifier	✓
exists quantifier	✓
scalene triangle	✓

(b) Results of Structural Tests

Table 2.25.: PEX results summary.

tion.

We evaluated the data generation facility of PEX by letting it explore all paths. The result is a set of test data combinations such that all feasible paths are executed. The Code Contracts preconditions are recognized and PEX interprets them as different branching statement. In other words, it tries to generate test data such that each clause of the specification is once fulfilled and once not. The result is a set of test input data. Tests not satisfying the precondition are marked *meaningless*. Tables 2.25a and 2.25b show that PEX is able to pass all benchmark tests. This does not necessary mean that PEX is able to test everything on the spot, but it is definitely the most advanced tool at the moment.

For all tests that included parameters of type *float* or *double* PEX issues a 'testability issue in floating point equality' warning. This warning tells the user that for floating point operations PEX only uses heuristics. Still PEX was able to generate correct input data for all those benchmarks.

2.4. Related Test Data Generation Tools

This sections provides a short summary of other test data generation tools. It does not include any UML based tools. Therefore, we do not consider tools such as QuviQ testing tools [Arts et al., 2006], CONFORMIQ [Huima, 2007], LEIRIOS [Jaffuel and Legeard, 2006], and the BZ testing tool from Legeard [Legeard et al., 2002].

This section is categorized in two parts:

- test data generation tools that were not considered to be part of the evaluation in Section 2.3.1 due to not fulfilling the required criteria listed, and

- Black-Box testing tools.

The tools mentioned in the upcoming sections are ordered chronologically. We use the citation count to decide which tools are mentioned and which not. Section 2.4.1 includes only those tools that are cited at least 150 times. Section 2.4.2 requires at least 30 citations. The citation count was determined through Google Scholar on October 8^{th} 2010.

2.4.1. Test Data Generation Tools

Korat [Boyapati et al., 2002] *(citation count: 384)* is a test case generation tool based on Design by Contract™ specification. It uses a methods post condition as oracle, and uses the precondition to generate complex test input data. Korat uses a *repOK()* and a *finitization()* method for constructing all non-isomorphic test input data up to a given bound. The *finitization* method implements the search for new input. Korat observes access to precondition predicates and class fields to prune the search space. Furthermore, the bound is specified in the *finitization* method. Korat provides a preliminary implementation and the user is able to enhance it if necessary. The *repOK* method implements the precondition check. It returns true if the generated input satisfies the specification, and false otherwise.

Visser et al. introduced JPF [Visser et al., 2003] in 2003 as a tool for model checking Java programs. It is a very mature tool, which was already applied to real world case studies. Among them the real-time operating system DEOS from Honeywell [Penix et al., 2005] and prototype Mars Rover [Brat et al., 2004]. Based on the JPF framework Visser et al. introduced a test input data generation extension [Visser et al., 2004] *(citation count: 198)* years later. Similar to PEX, the test input data generation extension of JPF uses symbolic execution of a *repOK* method, the methods precondition, to generate all (non-isomorphic) input data. A manual bound for the input data size is given. Multiple extensions to JPF exist that even add Design by Contract™ support, but unfortunately most of them are research prototypes or even not more than research ideas. JPFs test data generation capability is not included in this survey due to missing industrial size case studies.

In 2005 Sen was co-author of two very successful tools with respect to their publication count: DART and Cute. DART [Godefroid et al., 2005] *(citation count: 569)* is a test data generation tool that tries to cover all paths in the system under test, by combining concrete and symbolic execution. Cute [Sen and Agha, 2006] *(citation count: 395)* combines concrete and symbolic (concolic) execution, as well, but extends it to pointer structures. Note, concolic execution is equivalent to dynamic symbolic execution of PEX. DART combines three main techniques: (*a*) automated interface extraction, (*b*) automatic generation of test driver, and (*c*) automatic generation of new test input data based on dynamic analysis of program behavior with respect to its input data. Program crashes or assertion violations build the test oracle. DART gathers path constraints while executing the program under test with initially random input values. New input values for the same test force the execution to take a new path (for all reachable paths). Due to concolic execution DART can replace a path constraint, which the corresponding constraint/SAT solver cannot solve, with a concrete value, e.g., *true* or *false*. This allows DART to be used for more complex case studies. Cute is a close work to DART, but improves some steps in the process of test data generation. Sen points out in the related work section [Sen and Agha, 2006, p. 271] that unlike DART, Cute can handle pointers and data structures as input parameters and it implements a new constraint solver that significantly speeds up the analysis.

EXE [Cadar et al., 2008b] *(citation count: 258)* is the “youngest” test data generation tool that already achieved enough citations to be mentioned in this survey. EXE uses symbolic execution to generate input data that forces the program under test to crash. The programmer can mark variables, i.e., memory locations, to be traced symbolically. The program is then instrumented to execute all feasible paths. In case a path terminates, e.g., program crashes, a call to *exit()*, or an assertion fails, a concrete value is generated which can reproduce the error/crash when executing the original program without instrumentation. EXE performed well on the BSD and Linux packet filter implementations, udhcpd DHCP server, the pcre regular expression library, and three Linux file systems [Cadar et al., 2008b].

2.4.2. Black-Box Testing Tools

SpecExplorer [Campbell et al., 2008] *(citation count: 76)* is a set of tools working on programs written in the Spec# [Barnett et al., 2005] language. Programs written in Spec# are model programs, that include a formal specification and can be executed. SpecExplorer is designed to handle non-deterministic and multi-threaded software. Therefore, SpecExplorer distinguishes between *controllable* and *observable* actions. Furthermore, SpecExplorer provides a facility to specify accepting states through a condition. This is important since multi-threaded and non-deterministic programs do not always terminate. A model program may correspond to an infinite large automaton. A *test purpose* can be used to slice a model to the parts a test is interested in. SpecExplorer supports both offline and online testing. Offline tests are generated from that model either to provide some kind of coverage or based on a random walk in the state space. Online tests are created on the fly as testing proceeds [Campbell et al., 2008]. At each step one controllable action is selected, based on predefined or dynamically updated weights, to be executed.

UniTesK [Bourdonov et al., 2002] *(citation count: 40)* is a general architecture for test generation with two specialized versions: JavaTesK for Java and CTesK for C and C++ programs. UniTesK test sequence generation tool family works on the specification only. It requires *Mediator* instances to link the specification with a given implementation and keep those two independent systems synchronized during test execution. UniTesK traverses all states of the specification which are limited by an arbitrary coverage criterion. This coverage criterion can be described by a set of predicates, which values are calculated based on the system's state, the current operation and all its parameters. As with all other approaches and tools UniTesK uses the given specification as test oracle. UniTesK and its related tools for Java and C are developed to be used in testing industrial software.

2.5. Friends you can depend on

Mackinnon, Freeman and Craig introduced *mock objects* [Mackinnon et al., 2001]. *mock objects* replace code with a dummy implementation. Therefore, tests that feature *mock objects* are independent from system components, such as databases, network connections, file systems, and the current date and time. Furthermore, *mock objects* reduce object construction overhead in unit tests. A *mock object* does not require any constructor parameters, whereas the mocked object may have required other objects passed to the constructor. For all state-of-the-art programming languages libraries exist that simplify the instantiation and configuration of *mock objects*.

Becoming more and more popular, *mock objects* are categorized into five different groups depending on their execution behavior [Google, 2008]:

Dummy accepts all input values.

Stub can further be configured to return hard-coded values. It is therefore an object that does not change its state.

Mock focuses on the interaction. It records which methods are called in what order and raise an error when the client will misuse it.

Spy focuses on the actual state, and does not care about the interaction sequence.

Fake is the most advanced object. It swaps out the real implementation, but still behaves as if it would be the original.

2.6. Case Studies

The presented approaches are all evaluated on a common set of case studies. They are commonly introduced in this section. The thesis was part of an industry project, where the industry partner had to provide the case studies for the evaluation. Unfortunately, this took some time since the partner had to learn how to write Design by Contract™ specifications and the case study changed over the years. I therefore specified the *StackCalc* case study, and the *Design Patterns* by myself to have at least a case study at hand to evaluate the approaches. The industry partner provided the *StreamingFeeder* case study.

StackCalc is a student project implementation of a stack based calculator. In total we have 19 different operators for manipulating the stack's content. In addition, the *StackCalc* case study includes controller classes, the factory design pattern and a textual console output. The specification was added by the authors years after the implementation.

The *Design Patterns* case study is available online from the homepage of the Design Patterns in Java [Metsker and Wake, 2006] book. The implementation consists of all *Design Patterns* introduced by Gamma et al. [Gamma et al., 2002]. I use this case study because well-designed software heavily builds upon *Design Patterns*. We believe that the approach is useful in more general software components as well when we can show an improvement in the testability of *Design Patterns*.

The *StreamingFeeder* case study is a real-world application developed by our industry partner - an event processing framework. Events are required to be configured specifically. The specification was written by our industry partner before they started with the implementation.

Table 2.26 compares the size of the three case studies. One can see that the industry case-study is twice as big as the student project in terms of lines of code (non commenting source statements = NCSS), even though they are equally in terms of number of methods. The implementation of industrial projects tend to be more complex than the student work.

Another key point to mention is that the *StreamingFeeder* is about double the size in terms of postconditions. The reason for this difference might be in the fact that postconditions are required for getting automatic test case results - postconditions are used as oracle. The main benefit for the project partner to use Design by Contract™ is in improving their automatic test generation techniques. Therefore, they especially focused on writing postconditions.

Combining the figures leads to the observation that only a third of all methods in both case studies are annotated with preconditions at all. This can be explained by the fact that very often methods may be

attribute	*StackCalc*	*StreamingFeeder*	*Design Patterns*
classes	51	36	161
methods	147	167	825
NCSS	868	1456	3223
@Pre	52	48	225
@Post	69	125	191

Table 2.26.: Case studies in figures.

called at any time. For example all getter and setter methods do have *true* as their only precondition. Since *true* is the default, those methods are not explicitly annotated.

More interesting is the fact that only half of the methods are specified with postconditions. Since even getter and setter methods do have an effect, at least they update the state of the object, we would expect at least one postcondition for each method. The gap can be explained, by the ratio of the complexity to write the specification and the profit the software developer sees to write it.

Part II.

Contributions

Chapter 3

Concept

3.1. Overview

Figure 3.1 recalls all participants in the process of automatically generating and executing unit tests from Section 1.3. A test generator writes a unit test. Therefore, it has to instantiate all parameters of the method under test before actually calling it. During test execution the method under test may interact with the parameters and the environment, i.e., read or write a file, open a network connection.

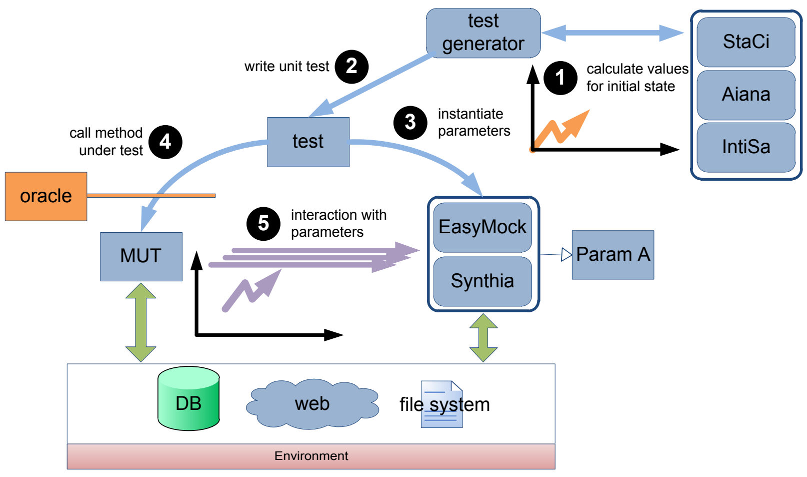


Figure 3.1.: Concept Overview of StaCi, AIana, Synthia, and IntiSa

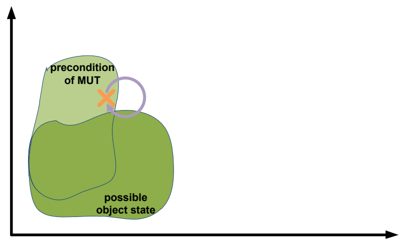
For primitive parameter types technologies exist that are able to generate values such that the precondition of the method under test is satisfied. For example, rewriting the precondition as constraint problem and passing it to a constraint solver [Girgis, 1993] or SMT solver [Tillmann and de Halleux, 2008] returns a set of value assignments that satisfy the given constraint.

For non primitive parameter types no such simple solution exists. Most used technologies are based on random approaches and are therefore not considered to be goal-oriented.

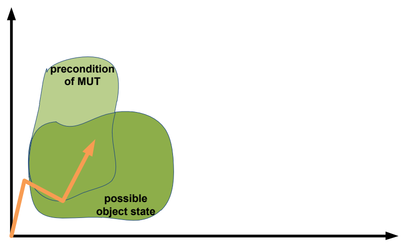
This thesis introduces the idea of replacing all non-primitive parameters with automatically synthesized fake objects. The fake object provides a mechanism to set its initial state. During test execution it behaves similar to any possible implementation of the corresponding parameter type.

For the realization of this vision I worked on two different types of technologies:

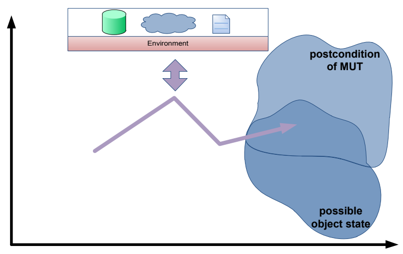
- Calculating the initial object state for non-primitive parameter types.
- Synthesizing a fake object that replaces the original non-primitive parameter type implementation.



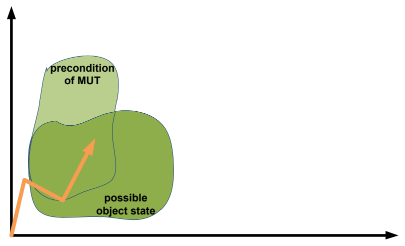
Instead of instantiating complex objects, human testers typically use mock objects. As described in Section 2.3 some tools use mock objects as well. The first approach presented in this thesis is STACI. STACI *a*) calculates an initial state that satisfies the precondition of the method under test, *b*) instantiates a mock object for all non-primitive parameter types, and *c*) configures the mock object with the calculated initial state. Still STACI may calculate an initial state that is not reachable through the public interface of the corresponding class and the mock configuration is static. In other words, the mock object will pass the precondition check of the method under test but any interaction of the method under test with the mock object does not trigger a state update.



For calculating the initial state, I picked up the idea of using an AI planner to calculate a method sequence that instantiates the non-primitive parameter object such that it satisfies the precondition. The Design by Contract™ specification is therefore mapped to PDDL, the plan domain description language, which is a standardized input to state-of-the art planner. Unfortunately, the input space of PDDL is smaller than that of Design by Contract™. For example, PDDL effects require a calculation prescription and not a state description as given by Design by Contract™ postconditions.



SYNTHIA is a fake object, which initial value can be set and which calculates the state update of each method call by means of an SMT solver at test execution. The initial state of the SYNTHIA FAKE can be easily set by the test. Furthermore, it behaves according to the specification of the requested type, and it removes any dependencies to the environment.



Finally, INTISA uses bounded model-checking to calculate an initial state for all involved parameter types that together satisfy the precondition of the method under test. The model is given in terms of a finite state machine, where a state is given by means of all object state variables and the transitions are given by means of method calls. This model is checked against the precondition of the method under test. Therefore, the result is a state that *a*) satisfies the precondition of the method under test, and *b*) can be reached by calling methods of the class' public interface.

3.2. Common Assumptions

All approaches underly the following assumptions:

- The program under test is single-threaded.
- Whatever might be of interest is specified.
- The program under test terminates.

Common Design by Contract™ specifications state what holds before and after execution of a method. In the case of multi-threaded applications, especially the postconditions may have to be weaker than for single-threaded applications since there might be a thread switch between the end of method execution and the contract checking. State-of-the art Design by Contract™ runtime assertion checker do not implement any mechanism to prevent this thread switch.

All presented approaches heavily rely on the Design by Contract™ specification. One advantage of Design by Contract™ is its incomplete specification capability, which allows to incremantally write specification as it is required. Whatever, is specified will be used by all presented approaches. But unspecified behavior cannot be automatically synthesized to SYNTHIA FAKE objects are used to model the finite state machine for the model checker.

Common Design by Contract™ specification does not feature any special keyword to state that a method may not return in certain circumstances. The presented approaches assume that all methods terminate. The actual tool implementation takes care of this assumption by means of a test generation timeout.

Chapter 4

Common Definitions

Parts of the contents of this chapter have been published by Stefan J. Galler, Christoph Zehentner, Franz Wotawa in "AIana: An AI Planning System for Test Data Generation" at 1st Workshop on Testing Object-Oriented Software Systems [Galler et al., 2010c], by Stefan J. Galler, Martin Weiglhofer, Franz Wotawa in "Synthesize it: From Design by Contract™ to Meaningful Test Input Data" at SEFM 2010 [Galler et al., 2010b], and by Stefan J. Galler, Thomas Quaritsch, Martin Weiglhofer, Franz Wotawa in "The IntiSa approach: Test Input Data Generation for Non-Primitive Data Types by means of SMT solver based Bounded Model Checking" to QSIC 2011 [Galler et al., 2011]

AIANA, SYNTHIA and INTISA use the same definitions for a class, their virtual state and similar concepts. Therefore, this section introduces all common definitions and tries to motivate their usage with small examples.

4.1. Running Example

Figure 4.1 presents the running example used throught the thesis to point out all concept details.

```
class SummationOperator {
  @Pre("stack.size()>=2")
  @Post("stack.size()=1")
  void sumUp(Stack stack) {
    double sum = 0.0;
    while(!stack.empty()) {
      sum += stack.peek();
      stack.pop();
    }
    stack.push(sum);
  }
}
```

```
@Model(name="mSize", type="int")
@Invariant("size()>=0")
class Stack {
```

```
16    private int size_;
17
18    @Post("mSize=@Old(mSize)+1")
19    void push(double elem) { ... }
20
21    @Pre("size()>0")
22    @Post("size()=@Old(size())-1")
23    void pop() { ... }
24
25    @Pure
26    @Post("@Return=mSize ∧ @Return>=0")
27    int size() { ... }
28
29    @Pure
30    @Pre("size()>0")
31    double peek() {...}
32
33    @Pure
34    @Post("@Return=(mSize=0)")
35    bool empty() {...}
36
37  }
```

The method under test in our running example is the *sumUp()* method (Line 4). The intention of this method is to replace all values on the stack with their sum. It requires a *Stack* object with at least two elements. The *Stack* object with its Design by Contract™ specification is given in Lines 13 to 37.

Note that the given specification of it is tailored for this thesis to let us show all important steps of AIANA, SYNTHIA FAKE, and INTISA. For example the model field is introduced to show that we can deal with this concept as well.

The syntax of the annotations throughout the thesis is based on *Modern Jass* [Rieken, 2007] but the specification itself is written in logic notation, i.e., the single equals character (=) states equality.

4.1.1. Classes, their Virtual State and Design by Contract™ contracts

This thesis presents automatic test data generation approaches for object-oriented programming languages. Therefore, one of the fundamental concepts is the *class*.

Definition 1 (Class) *A class C is a triple $\langle c, f, m\rangle$ where c denotes the non-empty set of constructors represented by the m-tuple $\langle cid, p_1, \ldots, p_n\rangle$, $f := f^{pub} \cup f^{prot} \cup f^{priv}$ is the union of public (f^{pub}), protected (f^{prot}), and private (f^{priv}) fields, and m the set of methods represented by the n-tuple $\langle mid, vis, ret, p_1, \ldots p_m\rangle$ where mid is a unique identifier, $vis \in \{public, protected, private\}$ defines the visibility of the method, and $ret \in \{void, prim, nprim\}$ determines the abstracted return type of the method, which distinguishes between no (void), primitive (prim), or object type (nprim) return type. p_i in the definition of c and m determines the i-th parameter type. We introduce the short notation m^{vis}_{ret} to reference the set of all methods in m that satisfy the two given attributes.*

Example 4.1. Consider the running example from Figure 4.1. The non-empty set of constructors contains only the default constructor $c_{default} := \langle Stack\rangle$. The Stack class does not contain any field, i.e., $f := \langle\rangle$. The set *m* contains an n-tuple for each of the *Stack* methods *push*, *pop*, *size* and *peek*. For example the method *push* is public, has no return type and takes one argument of type *double*. The n-tuple is therefore given by $m_{push} := \langle push, public, void, double \rangle$. Putting all together leads to

$$\begin{aligned} Stack := \langle \langle Stack \rangle, \langle\rangle, \langle \langle push, public, void, double \rangle \langle pop, public, void \rangle \\ \langle size, public, int \rangle \langle peek, public, double \rangle \rangle \rangle \end{aligned}$$

□

Furthermore, the presented approaches are based on the presence of Design by Contract™ specifications. The following definition denotes the Design by Contract™ specification of a particular class.

Definition 2 (Contract) *Given a class* $C := \langle c, f, m \rangle$ *then the Design by Contract™ specification of this class is denoted by* $DbC(C) := \langle mf, inv, pp \rangle$ *where* mf *is a set of model fields, inv is a set of invariants, and pp is a set of sets where each* $mc_{m_i} \in pp$*, denoted as method contract, is a tuple* $mc_{m_i} := \langle mb, pure \rangle$ *with mb, denoted as method behavior, being a set of pre-/postcondition pairs* $mb := \langle \langle P_1, Q_1 \rangle, \ldots, \langle P_n, Q_n \rangle \rangle$ *for a method* $m_i \in m$ *and pure being a boolean flag defining whether* m_i *is pure or not.*

Commonly, Design by Contract™ has special semantics in the case of automated test case generation. Given a precondition P and a postcondition Q for a method the usual semantics says, if the precondition is not satisfied, then there is no constraint on the methods behavior:

$$P \implies Q$$

However, for automated testing the postcondition Q is used as test oracle. In other words, if the result of the method execution does not satisfy Q, then the test verdict is *fail*. As a consequence, test input that does not fulfill the precondition is discarded as *meaningless*. Thus, in the case of automated testing, the precondition P is used as a guard rather than as the antecedence [Ammann and Offutt, 2008, p.78]. Furthermore, some Design by Contract™ specification languages support multiple pre- and postcondition pairs per method. This allows one to specify multiple behaviors for one and the same method. Its logical interpretation for test data generation is given in the Definition 3.

Definition 3 (method behavior) *A method behavior* $MB := \langle \langle P_1, Q_1 \rangle, \ldots, \langle P_i, Q_i \rangle \rangle$ *is a set of tuples, each consisting of a precondition P and a postcondition Q. The logical interpretation of a method behavior with multiple pre-/postcondition pairs is given by*

$$MB =_{df} (P_1 \wedge Q_1) \vee \cdots \vee (P_n \wedge Q_n)$$

where all variables in P and those annotated with @Old *in Q are unprimed, thus talking about the pre-state, and all other variables are primed, and thus talking about the post-state.*

Finite state machines are often used to model the *state* of an object [Ammann and Offutt, 2008, p.77ff]. Each state is an abstraction over a set of interesting values. The labels between the states correspond to methods that update the state.

In this thesis a *state* of the finite state machine is given by all publicly observable elements, i.e. public members, public methods with return values and model fields of the specification. This definition of a *state* is denoted as the *virtual state* of a class and is formally given by Definition 4.

Definition 4 (Virtual State) *Given a class* $C := \langle c, f^{pub} \cup f^{prot} \cup f^{priv}, m^{priv}_{void} \cup \cdots \cup m^{pub}_{nprim} \rangle$ *and its contract* $DbC(C) := \langle mf, inv, pp \rangle$*, then the virtual state of the class C, denoted by* $VS(C)$*, is given by* $VS(C) := f^{pub} \cup m^{pub}_{prim} \cup m^{pub}_{nprim} \cup mf$.

Example 4.2. Consider again the *Stack* of Figure 4.1 where $f^{pub} := \{\}$, $m^{pub}_{prim} := \{size, empty, peek\}$, $m^{pub}_{nprim} := \{\}$, and $mf := \{mSize\}$, thus the virtual state of the class *Stack* is given by $VS(Stack) := \{size, empty, peek, mSize\}$. Note that *empty* is a boolean variable, *size* and *mSize* are integer values while the domain of *peek* is double. □

4.1.2. Relations and Their Composition

Inspired by the Unifying Theories of Programming (UTP) [Hoare and He, 1998] we use an alphabetized version of Tarski's relational calculus [Tarski, 1941] to formally reason over Design by Contract™ specifications.

UTP uses alphabetized predicates, i.e. alphabet-predicate pairs. All free variables of the predicate are members of the alphabet. The relations are predicates over initial and intermediate or final observations. UTP uses undecorated variables (e.g., x, y, z) to denote initial observations and primed variables (e.g., x', y', z') to represent intermediate or final observations. Note that this corresponds to Design by Contract™: a method's precondition is given in terms of undecorated variables, while a postcondition is given in terms of undecorated and primed variables. The former referring to pre-state values, the latter to values after method execution.

The alphabet of a predicate P is denoted by αP. This alphabet is divided into before-variables (undecorated), i.e. $in\alpha P$, and into after-variables (primed), i.e. $out\alpha P$.

Definition 5 (Relation) *A relation is a pair* $(P, \alpha P)$*, where P is a predicate containing no free variables other than those in* αP*, and* $\alpha P := in\alpha P \cup out\alpha P$ *where* $in\alpha P$ *is a set of unprimed variables standing for initial values, and* $out\alpha P$ *is a set of primed variables standing for final values.*

The alphabetized predicates are combined by the use of standard predicate calculus operators (e.g., $\wedge$, $\vee$). However, the definitions of these operators have to be extended by an alphabet specification. For instance, in the case of a conjunction the alphabet is the union of the conjoined predicates: $\alpha(P \wedge Q) := \alpha P \cup \alpha Q$.

While UTP formalizes many programming concepts, our approach heavily relies on the concept of sequential composition. The sequential composition of two program statements expresses that the program starts with the first statement and after successful termination the second statement is executed. The final state of the first statement serves as initial state of the second statement, however, this intermediate state cannot be observed. All this is formalized in UTP using existential quantification.

Definition 6 (Sequential Composition)

$$\begin{aligned} P(v,v'); Q(v,v') &=_{df} \exists v_0 \bullet P(v,v_0) \wedge Q(v_0,v') \\ in\alpha(P(v,v'); Q(v,v')) &=_{df} in\alpha P \\ out\alpha(P(v,v'); Q(v,v')) &=_{df} out\alpha Q \end{aligned}$$

Note that the above definition uses the following substitution syntax: some or all of the free variables of a predicate are listed in brackets, e.g., $P(x)$. Let f be a list of expressions, then $P(f)$ denotes the result obtained by replacing every variable in x with the expression at the corresponding position in f.

4.1.3. Specification Enhancements

Usability is crucial for a technology to be applied. Therefore, Design by Contract™ typically allows specifications in the same language as the program is written. As a consequence a Design by Contract™ specification may call methods and constrain its return value. On the other hand, Design by Contract™ languages typically provide a set of keywords as specification shortcuts. The *pure* keyword is one of them.

It is therefore necessary to deal with those specification constructs. The following paragraphs define the logical interpretation of method calls in a specification and the framing problem, which is correlated to the *pure* keyword, used in this thesis.

Method calls in Specifications

A method's pre- or postcondition may make use of pure methods. That is, one may call pure methods in a specification.

Example 4.3. Consider for example a postcondition that states that two elements in an array are sorted $Q := elementAt(index) <= elementAt(index+1)$. □

Therefore, I introduce the term method reference, which denotes a single reference to a method and includes the actual arguments in case the referenced method requires parameters.

Definition 7 (method reference) *A method reference MR is given by the j-tuple $\langle m, arg_1, \dots, arg_i \rangle$ where m denotes the referenced method and arg_1, …, arg_i denote the possibly empty i-tuple of arguments passed to m.*

Example 4.4. The postcondition from Example 4.3 with $index = 1$ includes two method references to the same method:

$$MR_{elmentAt_1} = \langle m_{elementAt}, 1 \rangle$$
$$MR_{elmentAt_2} = \langle m_{elementAt}, 2 \rangle$$

where $m_{elementAt} = \langle elementAt, pub, prim, \langle index, int \rangle \rangle$. □

Furthermore, the specification may contain multiple method calls to the same method with different parameters. In that case, each method reference has to be *instantiated*, i.e., all parameters in the corresponding method behavior have to be replaced with the actual arguments passed and the *@Return* keyword has to be replaced with a unique identifier to be able to reason about the result of all method calls with different parameters.

Definition 8 (instantiated method reference) *Given a method reference $MR := \langle m, arg_1, \dots, arg_i \rangle$, the corresponding method $m := \langle mid, vis, ret, p_1, \dots, p_i \rangle$, and MB denotes the method behavior of m.*

Then the instantiated method reference is given by replacing all parameter occurrences in MB with the actual argument values $arg_1, \ldots, arg_i$. More formally,

$$[MR] =_{df} MB \left[\begin{array}{c} un(MR)\,/@Return \\ arg_0/p_0 \\ \ldots \\ arg_n/p_n \end{array} \right]$$

where $un(MR)$ returns a unique identifier for the given method reference.

Example 4.5. Consider again the first method reference from Example 4.4 $MR_{elmentAt_1} := \langle m_{elementAt}, 1\rangle$ where $m_{elementAt} := \langle elementAt, pub, prim, \langle index, int\rangle\rangle$ and the method behavior of the *elementAt* method $MB_{elementAt} := size() > index \wedge @Return = mArray[index]$ where *mArray* is a model field of type array of Double. The instantiated method reference replaces all general parameters in the method behavior with actual values from the method reference:

$$[MR_{elementAt}] := size() > 1 \wedge @Return = mArray[1]$$

□

Note p_0 to p_n may not only be a constant value but a method reference as well.

Lemma 1 *The instantiation of a method reference as given in Definition 8 takes $p * O(1) = O(p)$ time where p is the amount of parameters given.*

All those referenced methods may have a specification as well. Even though, methods referenced in a specification may not change the objects state, the contract may include information that can be useful.

Example 4.6. Consider the postcondition $Q_{pop} := size()' = size() - 1$ of the *pop* method from Figure 4.1. It only specifies the result of executing the method with respect to size(). The model field *mSize* is not updated. The postcondition of *push* $Q_{push} := mSize' = mSize + 1$, however, specifies with respect to the model field only. The link is missing. □

Thus, we have to extend the specifications with the contracts of all referenced methods. We call this *contract chaining*.

Definition 9 (contract chaining) *Given a specification R and let $MR_1, \ldots, MR_m$ and $MR_n, \ldots, MR_z$ be the method references for all methods used in terms of undecorated and decorated variables in R, respectively. Furthermore, $un(MR)$ returns a unique identifier for the given method reference, the same identifier as in Definition 5. Then the chained specification, denoted by R^{∞}, is given by*

$$R^{\infty} =_{df} \mu \bullet \left(\begin{array}{l} [MR_1]^{\infty}; \ldots ; [MR_m]^{\infty}; \\ R \left[\begin{array}{c} un(MR_1)/MR_1 \\ \ldots \\ un(MR_z)/MR_z \end{array} \right]; \\ [MR_n]^{\infty}; \ldots ; [MR_z]^{\infty} \end{array} \right)$$

where $MR^{\infty} =_{df} MB^{\infty}$ with MB being the method behavior of the referenced method in MR and

$$MB^{\infty} =_{df} (P_1 \wedge Q_1)^{\infty} \vee \cdots \vee (P_n \wedge Q_n)^{\infty}$$

Example 4.7. Consider the postcondition $Q_{pop} := (size()' = size() - 1)$ for the *pop()* method of our running example in Figure 4.1. The instantiated method behavior for *size* is given by $[MB_{size}]\, true \wedge size()' = mSize' \wedge mSize' >= 0$, and $un(size()) = size()$. The chained contract is given by

$$\begin{aligned}
Q_{pop}{}^{\infty} :=& MB_{size}; Q_{pop}[un(size())/size()]; MB_{size} \\
:=& true \wedge size()' = mSize' \wedge mSize' >= 0; \\
& (size()' = size() - 1); \\
& true \wedge size()' = mSize' \wedge mSize' >= 0
\end{aligned}$$

□

Lemma 2 *Chaining a specification as given in Definition 9) is in $O(t * p * d^p)$ time, where t is the total amount of methods present in the system under test, d is the domain of the parameters type, and p is the maximum number of parameters present for each method.*

Proof 1 *The recursive definition of R^{∞} consists of three parts: (a) the recursive chaining of all present pre-state method references in R, (b) the instantiation of all method references in R, and (c) again the recursive chaining of all present post-state method references in R. Step 1 and 3 builds a set of all unique method reference combinations with respect to method name and arguments passed. The upper bound of each of these sets is given by $O(t * p * d^p)$, where t being the total number of methods present in the system under test, p being the maximum parameters present for a method, and d being the domain of a parameters type. Therefore, Steps 1 and 3 require each $O(t * p * d^p)$. Step 2 replaces each method reference in the given specification with the unique name from Steps 1 and 3, which requires again $O(t * p * d^p)$. Therefore, the runtime complexity for Definition 9 is in $O(t * p * d^p) + O(t * p * d^p) + O(t * p * d^p) = O(t * p * d^p)$.*

Pure: Non-State Changing Methods

Annotating a method with the *@Pure* keyword is an abbreviation to stating that the object state does not change. In our terminology the object state is defined by the virtual state (see Definition 4).

Definition 10 (pure framed method behavior) *Given a method behavior MB of a method of class C, the framed method behavior, denoted by $\boxed{MB}$, is given by*

$$\boxed{MB} =_{df} \begin{cases} MB \bigwedge \forall v \in VS(C) \bullet v' = v) & \textit{if m is pure} \\ MB & \textit{otherwise} \end{cases}$$

Example 4.8. For example, the pure framed method behavior for the *size()* method of Figure 4.1 with the precondition *true* and the postcondition *@Return=mSize* $\wedge$ *mSize>=0* and the set of state variables $VS(Stack) = \{size, empty, peek, mSize\}$ (from Example 4.2) is given by

$$\begin{aligned}
\boxed{\langle true, @Return = mSize \wedge mSize >= 0\rangle} =& (true \wedge @Return = mSize \wedge mSize >= 0) \wedge \\
& size' = size \wedge empty' = empty \wedge \\
& peek' = peek \wedge mSize' = mSize
\end{aligned}$$

□

Lemma 3 *Pure framing of a method behavior as defined in Definition 10 has a runtime complexity of $O(s)$, where s is the number of state variables of the class.*

The expression $v' = v$ means that the value of v does not change. It is also well defined for primitive types, but not yet for arrays or object types.

For array types it basically means that all elements of the array do not change. In case of an array of object types the presented interpretations apply recursively.

Definition 11 (array type framing) *Given an array type variable v of length l the framing expression $v' = v$ is defined by $\bigwedge_{i=1}^{l} v'[i] = v[i]$*

Example 4.9. Given the array of double values $elem := 2.3, 4.5, 7.8$ with length $l = 3$ the framed variable $elem'$ is equal to $elem$. For the given example, $elem' = elem := 2.3, 4.5, 7.8$. □

For objects two different interpretations are possible: (a) a by-value framing, or (b) a by-reference framing.

All approaches use a by-value object type framing, which means that all state variables of the object do not change.

Definition 12 (by-value object type framing) *Given an object v the framing expression $v' = v$ is defined by $\bigwedge_{s \in VS(v)} s' = s$.*

Example 4.10. Consider a state variable v that is of type *Stack* from the running example in Figure 4.1 and the set of state variables from Example 4.2 $VS(Stack) := \{size, empty, peek, mSize\}$. The framing condition $v' = v$ then turns into $\bigwedge_{size,empty,peek,mSize} s' = s := size' = size \wedge empty' = empty \wedge peek' = peek \wedge mSize' = mSize$. □

Object-Oriented Aspects

A common concept of objects in object-oriented programming languages is that they have to be constructed before any other (non-static) method may be called on them. This concept is well-known and therefore not explicitly modeled in the Design by Contract™ specification of an object. Since some of the presented approaches reason over the Design by Contract™ specification this fundamental concept has to be modeled as well.

Therefore, we introduce a boolean variable *instantiated* for each class, which models the knowledge of being instantiated.

Definition 13 (object-oriented specification concept) *Given a Design by Contract™ specification S, which is either a precondition or a postcondition, the final specification that models the object-oriented aspect is given by*

$$OO(S) := \begin{array}{l|cc} \textit{method type} & \textit{precondition} & \textit{postcondition} \\ \hline \textit{constructor} & S \wedge !\textit{instantiated} & S \wedge \textit{instantiated} \\ \textit{non-static method} & S \wedge \textit{instantiated} & S \wedge \textit{instantiated} \\ \textit{static method} & S & S \end{array}$$

Lemma 4 *Adding one predicate to a specification as done by Definition 13 is in $O(1)$.*

Instantiated, Chained, Framed, Object-Oriented Method Behavior

Lemma 5 *Calculating the instantiated, chained, framed, object-oriented specification is in $O(t * p * d^p) + O(s)$ time, where p is the maximum number of parameters possible for a method, t is the total amount of methods present in the system under test, d being the domain of a parameters type, and s is the number of state variables of the class.*

Proof 2 *It follows directly, by summing up, from Lemmas 1, 2, and 3, and 4*

$$O(p) + O(t * p * d^p) + O(s) + O(1) = O(t * p * d^p) + O(s)$$

where p is the maximum number of parameters present for each method, t is the total amount of methods present in the system under test, d is the domain of the parameters type, and s is the number of state variables of the class.

Chapter 5

StaCi

Parts of the contents of this chapter have been published by Stefan J. Galler, Andreas Maller, Franz Wotawa in "Automatically Extracting Mock Object Behavior from Design-by-Contract Specification for Test Data Generation" at AST 2010 [Galler et al., 2010a]

5.1. Introduction

Freese and his group developed the concept of *mock objects* to make test driven development more efficient [Freese, 2002]. A *mock object* replaces the implementation of the original object with one specified by the tester. Tests featuring *mock objects* can therefore focus on one issue at a time and eliminate dependencies to other systems. Originally, the concept of *mock objects* was used by human testers.

Tillmann et al. [Tillmann and Schulte, 2006] and Saff et al. [Saff and Ernst, 2004] use *mock objects* to automatically generate test input data. They require in addition to the program under test, a set of system tests. The given test set is executed and the interaction of the method under test with its parameters is recorded. This information is later used in two different scenarios:

- The next time the test set is executed, the parameter objects are replaced by *mock objects* to reduce test execution time.
- Automatically generated unit tests use *mock objects* as test input which are configured to exactly behave as recorded earlier.

Both approaches do not guarantee that the behavior of the *mock object* corresponds to the intended one - since no formal specification is present. In addition, they may find errors, that are caused only by a change in the method call sequence, even tough the semantics did not change at all. Finally, they require a set of initial unit tests, which we aim to automatically generate.

StaCi is an approach embedded in a test generation framework, that automatically generates unit tests. The only input is the system under test and a Design by Contract™ specification for it. In the generated unit tests, all object type parameters of the method under test are replaced with *mock objects*. The Design by Contract™ specification is used to automatically configure the *mock object* such that the return values of all methods of the mocked object satisfy their post-condition and the

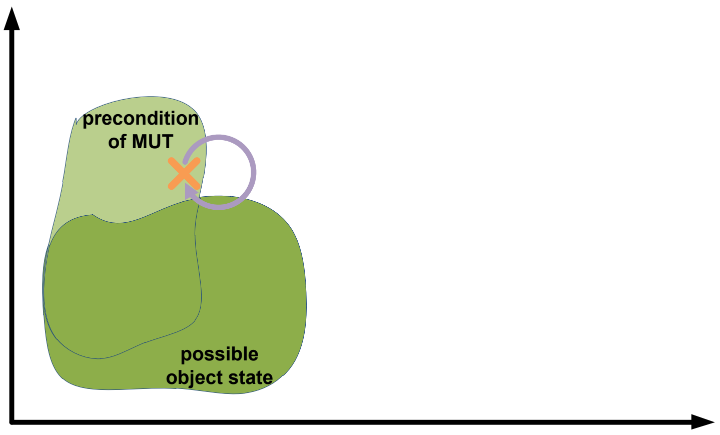


Figure 5.1.: STACI calculates an initial value and a static behavior for mock objects, which replace all object type parameters of the method under test. Each of the axis models the value domain of a state variable of the object.

pre-condition of themethod under test. The approach increases overall line coverage, since it is able to generate objects that satisfy the pre-condition of the method under test. It does not have to construct multiple objects to generate one that fits, such as the random approach has to.

Using STACI in automated unit test generation combines the benefits of mocking and automated test generation. STACI

- generates less meaningless tests, thus a higher line coverage can be achieved
- generated tests separate bugs in the method under test from those in the environment, i.e., the implementation of the test input data, since they get mocked
- removes system dependencies (e.g., network connections, databases, file system)
- reduces object construction overhead (method calls to transform the object into a required state, construction of objects that are required by the constructor or any other method call on the parameter object)

Figure 5.1 shows an example two-dimensional problem domain. The precondition of the method under test and the area of the possible object state for one parameter of the method under test are highlighted. STACI generates an initial state for the *EasyMock* object that satisfies the precondition of the method under test. The figure also points out the two major limitations of the approach: (*a*) the initial state may not be reachable through method calls on the actual parameter object and is therefore outside the area marked with *possible object state*, and (*b*) the static behavior of the *mock object*, as emphasized in the figure through the self-loop.

5.2. Concept

The idea of STACI is to replace all object type parameters by *mock objects*. The *mock objects* are configured to behave according to the Design by Contract™ specification of the mocked type. Therefore, the postcondition for each method of the original object is conjuncted with those parts of the precondition of the method under test, that constrain the corresponding method. Then an existing test data generation approach, such as random, is used to generate values that satisfy this specification. The *mock object* is configured to return those calculated values. This leads to less meaningless tests, since at test execution time the *mock object* return values that satisfy the precondition of the method under test. This approach may generate failing tests that would not fail when executed on the original implementation. We welcome this behavior, since it points to so called *hidden errors*. Hidden in the sense of not present in the current implementation due to the fact that the implementation is stronger than the actual specification. As soon as the implementation changes, within the border of the not-changing specification, the error will be emphasized when executing the test on the implementation as well.

STACI generates iteratively an object for each object type parameter type of the method under test. It takes the type of the parameter and its specification as input and executes the following steps:

1. the *mock object* for the given type is instantiated
2. the required behavior of the *mock object* is derived from the Design by Contract™ specification of the given type
3. the *mock object* is configured through the API of the used mock library to behave accordingly
4. the generated *mock object* is returned and used as test input.

The required behavior of the *mock object* is determined based on the Design by Contract™ specification of the corresponding type and the given pre-condition of the method under test. The Design by Contract™ specification of the type defines the principal behavior of the *mock object*. The pre-condition of the method under test strengthens this specification further, to guarantee that the *mock object* reacts as if it is in a state that satisfies the pre-condition.

Therefore, we configure all methods of the *mock object*, such that they return a value according to the conjunction of the pre-condition of the method under test and the post-condition of the mocked method. The detailed description of this configuration process is presented in Section 5.2.2.

STACI leads to higher code coverage, in terms of covered lines, since it generates test input data that satisfy the precondition of more methods under test. Satisfying more preconditions leads to more executed methods, which in turn increases the amount of executed source code lines.

5.2.1. Supported Specification Syntax

Figure 5.2 defines the syntax of the pre-condition currently supported by the implementation. The pre-condition is a conjunction/disjunction of Boolean expressions. Each of the Boolean expressions may consist of a left-hand side, a comparison operator and a right-hand side. The latter two are optional in case that the left-hand side evaluates to a boolean value. The left hand-side is a method call or a single statement of multiple successive method calls, e.g., `stack.pop().toString()`. The right-hand side defines the comparison value for the return value of the last method call, in the example the call to *toString()*. All other method calls on the left-hand side are considered to return a valid instance, i.e., not null.

```
PRECOND := BEXPR
        |  BEXPR ∧ PRECOND
        |  BEXPR ∨ PRECOND
BEXPR   := var != null
        |  var MCALL
        |  var MCALL OP value
MCALL   := .methname(ARGLIST)
        |  .methname(ARGLIST) MCALL
ARGLIST := ε
        |  var
        |  var, ARGLIST
OP := == | != | ≤ | <= | ≥ | >=
var := [a-zA-Z][a-zA-Z0-0]*
```

Figure 5.2.: Grammar of the supported pre-condition syntax. The pre-condition (PRECOND) is build up by Boolean expressions (BEXPR). Each of them defining the return value of a method call (MCALL). The comparison of two method calls is not supported by the current implementation.

The currently implemented syntax does not support a method call on the right-hand side. The pre-condition of our running example in Figure 4.1 is matched by the specified syntax.

5.2.2. From Design by Contract™ to mock behavior

Automatically extracting *mock object* behavior from Design by Contract™ specification for test data generation requires the class type to generate and the Design by Contract™ specification. Each step of Algorithm 1 is explained in detail in the following paragraphs.

Algorithm 1 Design by Contract™ to map

```
1. build a map: method $\mapsto$ post-condition (postcond) of method
2. derive requested state of the mock object from the pre-condition of the method under test (precond)
    a) split pre-condition in conjunction of sub-conditions (subcond) where each Boolean expression in one sub-condition references always the same method: $subcond_1$ && ... && $subcond_n \Longrightarrow precond$
    b) update map to: method $\mapsto$ subcond && postcond
3. generate return value that satisfies the specification
4. generate Java code
```

Algorithm 1 is not sound since the calculated *mock object* configuration is static. The static behavior of the mock may cause a failing test, which may succeed when using the real implementation (see Section 5.3.4).

build a map First, a map containing all non-void methods of the *mock object* is created. Second, to each method the corresponding post-condition is mapped. Figure 5.3 shows a summary of the map for the *java.util.Stack<Double>* class of the running example (Figure 4.1).

java.util.Stack<Double>		
method		constraint
size()	$\mapsto$	@Return>=0
pop()	$\mapsto$	size()==@Old(size())-1
peek()	$\mapsto$	@Return! =null

Figure 5.3.: Map content after first step of transformation process. The map contains for each non-void method an entry: method $\mapsto$ post-condition.

derive requested state To derive the requested state for the *mock object*, the pre-condition of the method under test (which includes constraints on the *mock object*) is transformed, through re-writing of the specification by following a number of rules.

Therefore, for each method of the mocked object a sub-condition of the pre-condition is extracted, such that the sub-condition constrains only the according method. A sub-condition in that sense, is a conjunction/disjunction of Boolean expressions, with equal left-hand sides.

Algorithm 2 Extraction rules

```
split(spec)
{
  if(spec.matches($(m_1 \wedge o) \wedge m_2$)
    return(split($m_1 \wedge m_2$);
  if(spec.matches($(m_1 \vee o) \wedge m_2$)
    return split($m_1 \wedge m_2$);
  if(spec.matches($(m_1 \wedge o) \vee m_2$)
    return split($m_1$);
  if(spec.matches($m_1 \vee o \vee m_2$)
    return split($m_1$);
  if(spec.matches($m_1 \vee m_2$)
    return split($m_1$);
  return spec;
}
```

The generated specifications by Algorithm 2 are sound and complete. The input specification is always satisfied when the conjunction of the generated specifications is satisfied. In addition, for each model of the input specification the conjunction of the generated specifications is satisfied as well.

The conditions m_1 and m_2 in Algorithm 2 are two Boolean expressions matching the *BEXPR* rule from Figure 5.2. Both impose constraints on the same method m - one of those present in the map. Condition o refers to any other boolean expression.

The given rules are applied repeatedly until no rule is applicable anymore. No rule is applicable anymore when the pre-condition does not include any Boolean expression that references the currently processed method m.

Lemma 6 *Algorithm 2 terminates in $O(c)$, where c is the amount of clauses present in the specification.*

Proof 3 *Each extraction rule given in Algorithm 2 reduces the overall specification, in other words, the amount of present clauses decreases. The termination is given due to the fact, that the algorithm stops as soon as no rule is applicable anymore. Due to the decreasing number of clauses in the specification, the algorithm terminates latest when only one clause is left. Therefore, in the worst case the algorithm is executed recursively as often as clauses are present. The process of removing a clause costs $O(1)$. Therefore, Algorithm 2 terminates in $c * O(1) = O(c)$, where c is the maximum amount of clauses present in the specification at the beginning of the algorithm.*

Basically the rules are applied to the overall Boolean expression iteratively for each method *m* in the map. The result is a Boolean expression where all left-hand sides equal *m*, i.e., imposing constraints only on method *m*. Any Boolean expression *o*, which left-hand side differs from m_1 can be removed. Therefore, rules 1 and 2 only keep *m*1 and *m*2. Rules 3, 4 and 5 are similar to 1 and 2, but in those cases even *m*2 can be removed since the overall Boolean expression is satisfied as soon as *m*1 is satisfied. Following those rules guarantees, that if all sub-conditions are satisfied the original pre-condition is satisfied too.

Each generated sub-condition is then conjuncted with the already present specification of the corresponding method in the map. The resulting map of this step in the transformation process is given in Figure 5.4.

		java.util.Stack<Double>
method		constraint
size()	$\mapsto$	@Return>=0 $\wedge$ @Return>= 2
pop()	$\mapsto$	size()==@Old(size())-1
peek()	$\mapsto$	@Return! =null $\wedge$ @Return.doubleValue()<10.000

Figure 5.4.: Map content after the second step of the transformation process.

generate return values For each method in the map return values are generated, that satisfy the specification present in the map. This is done by calling the *value generator component* of the test data generation tool and passing the specification as argument.

Depending on the required return type, different value generators are either called with the given specification or called until the generated value satisfies the given sub-condition (or a maximum number of trials is reached). In our case a value for primitive data types is generated randomly and checked against the specification. If it does not hold, a new value is generated randomly. For object types the *value generator component* constructs a *mock object* recursively (up to a manually set depth). The provided specification is the sub-condition eliminated by all Boolean expressions that do not constrain any return value of method calls on the requested type itself. For this elimination the same algorithm as described in Figure 2 is applied again with m_1 and m_2 referencing the method called on the object under generation. In addition, a new temporary variable has to be introduced, referencing the required return value. In the specification the method call on the object under generation in m_1 and m_2 has to be replaced by the temporary variable.

Given the pre-condition of the *add()* method in Figure 4.1 and generating the return value for method call `peek()`, the provided specification to the *value generator component* is

$$stack.peek().doubleValue() < 10.000$$

before introducing the temporary variable. Finally, the passed specification is

$$tmp1.doubleValue() < 10.000$$

where tmp1 is of type *Double*. Figure 5.5 shows the map updated with exemplary return values.

		java.util.Stack<Double>
method		return value
size()	$\mapsto$	int: 5
pop()	$\mapsto$	Double: 2.4
peek()	$\mapsto$	Double: 2.4

Figure 5.5.: Map content after the third step of the transformation process.

generate Java code The last step of the transformation process generates the actual Java code. After instantiation of the *mock object*, for each method of the mocked object the corresponding *EasyMock* API call is generated. For each parameter of the mocked method a so called *parameter matcher* is instantiated. A parameter matcher specifies the expected value range. Currently, no constraints on parameters are considered since those are checked by the Design by Contract™ tool anyway. Thus, for a parameter of type Double the *EasyMock.anyDouble()* matcher is generated. For all other types similar API calls are available. Each method is expected to be called *any times*. If the method returns an object, the *andReturn()* API call is added as well.

The generated code for the running example is presented in Figure 5.6. Calling method *size()* returns always the value *5*. *push()* may be called with any valid *Double* but does not return anything. *pop()* returns a *Double* instance of value 2.4.

```java
Stack stack = EasyMock.createMock(Stack.class);
int v0 = stack.size();
EasyMock.expectLastCall().andReturn(5).anyTimes();
stack.push(EasyMock.anyDouble());
EasyMock.expectLastCall().atLeastOnce();
Double v1 = stack.pop();
EasyMock.expectLastCall().andReturn(2.4).anyTimes();
...
EasyMock.replay(stack);
```

Figure 5.6.: Resulting Java code. Automatically generated mock behavior configuration for the running example from Figure 4.1.

5.3. Discussion

This section points out the advantages of STACI as well as all known disadvantages and limitations. In addition, it contains a discussion of the algorithms runtime and general comments that clarify the applicability of STACI.

5.3.1. Advantages

STACI combines the advantages of a goal-oriented test data generation technique for object types with those of using *mock objects*.

In general STACI generates less meaningless tests by configuring the used *mock objects* to satisfy the precondition of the method under test. Configuring a *mock object* means to tell the *mock object* what values it has to return for each method call. Depending on the technology used upfront to calculate return values that satisfy the given specification, STACI may produce less meaningless tests than random approaches do or even no meaningless tests at all.

Furthermore, using *mock objects* instead of the real implementation (*a*) reduces construction overhead, (*b*) removes system dependencies, and (*c*) focuses on faults in the method under test.

Reduced Construction Overhead *mock objects* are not required to be constructed by means of calling a constructor and the required methods to transform the object into the precondition satisfying state. Instead *mock objects* have to be only instantiated and configured. This does not require the construction of other objects such as parameters required by method calls.

Reduced System Dependencies *mock objects* do not have any implementation. Therefore, they do not interact with any other component. Especially important is that *mock objects* are independent from any system resources, such as the file system, databases, network connection or even the current date and time. Tests that feature *mock objects* can simulate the behavior of the method under test at different points in time since even the time can be mocked.

Focuses on faults in method under test Replacing all parameters of the method under test with *mock objects* isolates the implementation of the method under test. Therefore, only faults that are present in the method under test may be revealed by the unit test. This makes the analysis and fixing process of each reported fault easier. Since every method in the system is tested at least once still the whole system is covered through tests.

5.3.2. Disadvantages and Limitations

This section deals with limitations imposed by the concept of STACI. It does not discuss in detail the limited applicability of *mock objects* to non-final classes or similar implementation related issues.

As already mentioned the splitting rules are not applicable to specifications with parameter dependencies. Besides that there are the following three major limitations.

Unreachable Initial State STACI generates values that satisfy the precondition of the method under test and the postcondition of the method. Therefore, it generates a value that satisfies any given Design by Contract™ specification. But it might be the case that this value is actually not achievable through method calls. INTISA (see Chapter 8) will target this limitation.

Static State Configuration of *EasyMock mock object* STACI is not applicable for implementations that expect state changes of the *mock object* nor Design by Contract™ specifications that specify the state of the mocked object after method execution. Since the generated *mock object* for the Stack in Figure 5.7 always returns the same value for each call to method *size()* - one that satisfies the pre-condition of the method under test - the given implementation leads to an endless loop. The test execution terminates after a specified timeout.

```
@Pre("stack.size()>1")
int sumUp(Stack stack) {
  int tmp = 0;
  while(stack.size()>0) {
    tmp += stack.pop();
  }
  return tmp;
}
```

Figure 5.7.: Example implementation of a method that expects a state change of the *mock object*. Currently the generated test results in an endless loop.

This is caused by the static behavior configuration of the *mock object*. The *mock object* configuration does not consider any internal state changes caused by method calls. SYNTHIA (see Chapter 7) targets this limitation.

```
public class Math {
  @Pre("x>0 ∧ y>0")
  @Post("@Return=x*y")
  int multiply(int x, int y) {
    return x * y;
  }
}
```

Figure 5.8.: Example specification of a weak post-condition for a method that returns the result of the addition of two integer values passed as parameters.

Return Value May Not Depend on Parameter Values Consider a method under test that requires the *Math* object in Figure 5.8 as test input. The presented approach uses the specified post-conditions to generate a return value that satisfies it, at test generation time.

The presented approach does not work, whenever the return value of the method depends on parameters of the method. The value of the parameter may not be defined before test execution time, therefore the test data generation algorithm may not generate a return value for the method at test generation time. This limitation is also target by SYNTHIA (see Chapter 7).

5.3.3. Runtime Analysis

Theorem 1 *The upper bound runtime of Algorithm 1 is $O(m * c)$ where m is the amount of methods in the parameters type and c is the amount of clauses of the precondition of the method under test, under the assumption that in step 3 a random value generator is used.*

Proof 4 *Step 1 of Algorithm 1 builds a map with an entry for each method of the parameters type. The insertion operation requires $O(1)$ for all m methods, resulting in $m * O(1) = O(m)$.*
Step 2 splits and updates the map for each method of the parameter under generation. The runtime for the splitting is given in Lemma 6 and updating of the hash map requires $O(1)$. Since both actions are executed for each method of the parameter under generations type, it totals in $m * (O(c) + O(1)) = O(m * c)$.
Step 3 depends on the value generation algorithm used. Assuming random is used as in our implementation runtime is given by $O(1)$.
Step 4 requires $O(1)$.
The upper bound runtime is therefore given by

$$O(m) + O(m * c) + O(1) + O(1) = O(m * c)$$

5.3.4. Comments

The following questions remain to be answered:

- Does STACI generate false positives?
- How does STACI deal with unsupported specification?

Does STACI generate false positives? Yes, STACI may generate two different kinds of false positive unit tests in terms of, failing unit tests that actually do not emphasize an actual fault in the implementation.
First, STACI uses the *EasyMock* library to configure a static behavior for the parameter. Any interaction of the method with the parameter does not effect the parameters state. Therefore, the *mock object* may cause postcondition violations that do not happen if tested with the actual implementation. This disadvantage of STACI is addressed by SYNTHIA (see Chapter 7).
Second, STACI uses only the specification to generate test data. A specification is an abstraction of the actual implementation. Therefore, STACI may generate test input that emphasizes a fault in the method under test implementation that would not be emphasized using the actual implementation of the parameters. This is a desired behavior of STACI, since it emphasizes hidden faults. Hidden in the sense of not currently observable, but as soon as the implementation may change within the bounds of the specification, the mismatch between specification and implementation may become an error.

How does StaCi deal with unsupported specification? StaCi basically includes two different technologies. First, it replaces all object type parameters with *mock objects*, thus it reduces test input construction. Second, StaCi configures these *mock objects* such that they satisfy the precondition of the method under test. Unsupported specification influences only the second step. If unsupported specification is present StaCi ignores the specification and uses a pure random approach. Since StaCi uses only those specifications that are relevant for the one value it currently tries to generate, an unsupported specification may only influence one single value. The value generation for all other state variables is not targeted. Therefore, in the worst case StaCi is as good as a random approach but it is very likely that StaCi performs better.

5.4. Evaluation

For our evaluation we implemented StaCi as an extension to the open-source *JET* [Cheon et al., 2005a, Cheon, 2007] tool. The original *JET* version is used as benchmark. We use the *easymock* [Freese, 2002] library for instantiating the *mock objects*.

5.4.1. Evaluation Process

The evaluation is conducted on multiple servers with the same hardware and software configuration (AMD Athlon 64bit Dual Core; each 2,2GHz; 2GB RAM).

Both, our approach and the random approach, are executed with 25 different parameter value combinations. Each of these configurations is executed 30 times for each approach. The two alternating parameters are:

1. *mutating probability*: This parameter defines the probability that after executing a method call on a generated object, another method is chosen to change the objects state (mutate the object state). This parameter is only used by the random approach. The presented results are averages over execution runs with one of the following values: 0.0, 0.2, 0.4, 0.6, 0.8.
2. *attempts*: This parameter defines how many tests are generated for each method of the case study. We used the following values: 1, 2, 3, 4, 5.

We claim that using automatic generation of mock behavior from Design by Contract™ specification reduces the amount of meaningless tests generated. Less meaningless test increase the amount of methods executed. Therefore, we use the *cobertura* [Doliner, 2006] library to measure line and branch coverage.

Using more sophisticated methods usually takes more resources. Since the approach is only applicable to industrial projects when the increase of test generation time and execution time is worth it, we also evaluate the approaches with respect to test generation time and test execution time.

5.4.2. Results and Discussion

The presented approach leads to a significant increase in line and branch coverage for both case studies.

	StackCalc				Design Patterns			
	RANDOM		MOCK		RANDOM		MOCK	
attempts	line	branch	line	branch	line	branch	line	branch
1	11%	1%	17%	8%	33%	13%	41%	20%
2	11%	1%	17%	9%	35%	15%	42%	22%
3	12%	1%	18%	10%	35%	16%	41%	22%
4	14%	2%	21%	12%	39%	18%	45%	24%
5	2%	2%	17%	10%	35%	17%	40%	23%
avg.	12%	1%	18%	10%	35%	16%	42%	22%

Table 5.1.: Achieved line and branch coverage for *StackCalc* and Design Patterns case study.

The improvement by 50% for the *StackCalc* case study (Table 5.1) is due to the design structure. About half of the methods under test operate on the Stack (e.g., sum up, multiply, subtract or divide the last two elements on the stack) and therefore have very similar pre-conditions. They require at least two elements on the stack. The random approach was not able to generate a Stack object with two elements in hardly any test run. Whereas, the presented approach always succeeded in generating a Stack object that satisfies the given pre-condition.

The enormous increase in branch coverage can be tracked down to the same issue. All methods operating on the stack use conditional statements. As soon as the test input data satisfies the pre-condition of the method under test, at least one conditional statement is executed.

The *Design Pattern* case study, as the more general one in terms of software design, still shows an increase in line coverage by 20% as presented in Table 5.1. Since this case study includes all design patterns this increase cannot be explained by a special software design anymore.

One may have suggested that the coverage decreases due to the fact, that *mock objects* replace real implementation. This is not the case, because each class is tested anyway, but separately. Only when the class is used as a parameter for a method the implementation is mocked.

As expected the presented approach requires slightly more time for both, test generation and test execution. Table 5.2 shows in the first column of each approach the required time to generate all tests divided by the amount of methods for which the approach tried to generate a test. Column two shows the required time divided by the amount of actually generated tests.

The increase of time required for generating tests is caused by heavy use of Java reflection and the overhead of instantiating a *mock object*. But with increasing size of the case study the premium decreases relatively to the overall generation time.

In addition, to the significant increase of line coverage of the system under test the presented approach inherits all advantages from using *mock objects*. The *Design Pattern* case study uses Java *File* objects. The random approach messed up the file system by generating thousands of randomly named files. They had to be deleted manually after the test generation process. The presented approach did not create a single file, since the *File* object was mocked.

It is to mention that one class of the *StackCalc* case study could not be tested due to the static behavior configuration of the *mock object*. This limitation is discussed in Section 5.3.2 and shown in Figure 5.7.

	StackCalc						Design Patterns					
	RANDOM			MOCK			RANDOM			MOCK		
att.	gen./ sum	gen./ succ.	exec.	gen./ sum	gen/ succ.	exec.	gen./ sum	gen./ succ.	exec.	gen./ sum	gen/ succ.	exec.
1	0.83	0.93	0.79	1.08	1.41	0.41	0.81	0.96	0.28	0.89	1.03	0.43
2	0.55	0.61	0.55	0.71	0.93	0.26	0.64	0.74	0.18	0.68	0.79	0.32
3	0.44	0.49	0.42	0.57	0.75	0.20	0.62	0.71	0.14	0.62	0.7	0.26
4	0.38	0.42	0.36	0.47	0.62	0.16	0.55	0.62	0.12	0.59	0.65	0.23
5	0.41	0.46	0.32	0.51	0.68	0.14	0.52	0.58	0.10	0.55	0.61	0.21
avg.	0.45	0.50	0.45	0.57	0.76	0.20	0.58	0.66	0.14	0.62	0.69	0.26

Table 5.2.: Required time per test for *StackCalc* and Design Patterns case study.

Chapter 6

AIANA

Parts of the contents of this chapter have been published by Stefan J. Galler, Christoph Zehentner, Franz Wotawa in "AIana: An AI Planning System for Test Data Generation" at 1st Workshop on Testing Object-Oriented Software Systems [Galler et al., 2010c] and in the master's thesis of Christoph Zehentner[Zehentner, 2010]

6.1. Introduction

When confronted with the problem of providing object type test input data for a given method under test, a human being will examine the interface of the object and try to *come up* with a method sequence that transforms the object to a state satisfying the precondition of the method under test. The process of automatically *coming up* with a solution is well known to computer scientists. It is called planning.

Different planning algorithm and problem description notations exist. Very popular and common are so called STRIPS style planners [Fikes and Nilsson, 1971]. For STRIPS style planners the set of possible actions is described by means of *preconditions* and *effects*.

The AIANA approach is based on the observation that both Design by Contract™ and STRIPS style planner specify the behavior of corresponding items with *preconditions* and *effects* or *postconditions*. Therefore, AIANA maps the problem of finding object type test input data that satisfies the precondition of the method under test to a STRIPS style planning problem, asks a state-of-the art planner to *come up* with a plan and then instantiates the object accordingly.

Figure 6.1 shows a two-dimensional object space. Each axis corresponds to the domain of one state variable. The figure highlights the reachable object state and the state required by the precondition of the method under test. The reachable object state is determined by means of public methods available to transform the object. AIANA tries to find a method sequence that transforms the object to a state that satisfies the precondition of the method under test.

The basic idea is similar to Howe [von Mayrhauser et al., 2000] and Leitner [Leitner and Bloem, 2005]. But in contrast to previous work, AIANA fully automates the approach and can deal with real Java implementations.

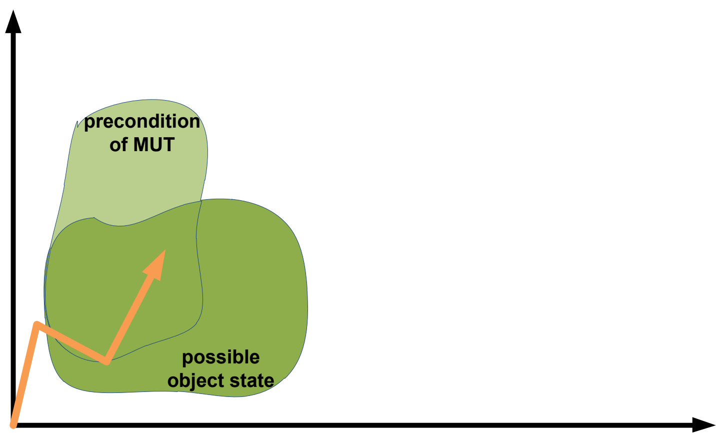


Figure 6.1.: AIANA uses an AI planner to calculate an initial state that satisfies the precondition of the method under test. Each of the axis models the value domain of a state variable of the object.

Note that there is a semantic difference between effects and postconditions. Therefore, AIANA is not applicable in every case.

6.2. Assumptions

In addition to the assumptions stated in Section 3.2 AIANA builds on

- the closed world assumption, and
- the frameing assumption.

6.2.1. The Closed World Assumption

The Closed World Assumption states, that if a term can not be proven to be true, it can be considered false. If a planner uses the Closed World Assumption only terms need to be given that are true in the current state. This reduces the amount of statements within planning domain descriptions. [Zehentner, 2010, Russell and Norvig, 2002]

6.2.2. The Frame Problem

The Frame Problem was first mentioned by McCarthy and Hayes [Mccarthy and Hayes, 1969]. It basically describes the problem that machines do not know which conditions change if they are not explicitly stated. Planner and similar technologies have to know how to deal with unconstrained variables in order to calculate the next state. A very common interpretation therefore is, that all unconstrained variables keep their value and all state updates have to be explicitly stated in the specification. [Zehentner, 2010]

6.3. Preliminaries

AIANA is the only approach of this thesis that deals with AI planner and their plan description languages. Therefore, the following two sections introduce shortly the planning problem, the planning domain, the initial and goal state, as well as what a plan is. Furthermore, the Planning Domain Definition Language (PDDL) [Ghallab et al., 1998] is introduced. Where necessary formal definitions are given for a precise understanding on which parts AIANA builds upon. The examples are already based on the AIANA concept, in other words, they use the mapping from Design by Contract™ specifications to PDDL actions.

6.3.1. AI planning

Planning can be described as finding a sequence of actions that will bring a system from its initial state *IS* to a desired goal state *GS*. This requires a description of the changes an action can induce in the world. This description is called planning domain.

Definition 14 (planning domain) *The planning domain is the set D of actions a_i: $D := \{a_1, ..., a_n\}$. An action is given by the tuple $a := \langle name, parameters, P, E\rangle$, where P is the precondition and E the effect of the action.*

Example 6.1. Consider the two actions *push* and *pop* from the *Stack* from Figure 4.1. Both actions do not have any parameters. The precondition of *push* is not explicitly given, it is therefore considered to be *true*. Whereas the precondition of *pop* is given by $size() > 0$. The postcondition or effect is stated for both actions. *push* increases size by one, *pop* decreases size by one. The resulting planning domain is therefore given by, $D := \{\langle push, \emptyset, true, size+1\rangle, \langle pop, \emptyset, size > 0, size-1\rangle\}$ □

The initial state and the goal state are lists of predicates that describe the state that is present before any plan is executed and the state that should be achieved, respectively.

Definition 15 (initial state) *We define the initial state IS to be the set of all predicates that describe the initial state of the world: $IS := \{p_i, ...p_k\}$*

Example 6.2. The initial state of an object is given by the default values of all variables with respect to the programming language. For the domain from Example 6.1 the initial state according to Definition 15 is: $IS := \{size = 0\}$. □

Definition 16 (goal state) *The planning goal is the set GS containing all predicates p that describe the desired state of the world $GS := \{p_1, ..., p_n\}$, where $p_i \in RS(P,I)$ for the method under tests' precondition P and the parameter to be constructed I where RS returns the result of applying Algorithm 2.*

Example 6.3. Using *sumUp* from Figure 4.1 as method under test, its precondition will describe the goal state of all parameters passed. It is therefore given by $GS := \{size() >= 2\}$. □

All three parts, the planning domain, the initial state, and the goal state, together define the planning problem.

Definition 17 (planning problem) *A planning problem is the triple $PP := \langle D, IS, GS \rangle$ where D is the planning domain consisting of all available actions: $D := \{a_1, ..., a_n\}$, IS is the initial state and GS is the desired goal state.*

Example 6.4. Using the Examples 6.1, 6.2, and 6.3 from this chapter the planning problem is given by

$$PP := \langle \{\langle push, \emptyset, true, size + 1 \rangle, \langle pop, \emptyset, size > 0, size - 1 \rangle\}, \{size = 0\}, \{size >= 2\} \rangle.$$

□

The description of which conditions must hold before execution of an action a is valid, can be referred as precondition P of the action, whereas the list of changes is called the effect E. In terms of Hoare logic [Hoare, 1969] such an action can be written as:

$$\{P\}\ A\ \{E\}$$

Therefore, a plan can be described as sequence of actions $a_1 \dots a_n$ that transforms the initial state IS such that in the end the goal state GS is satisfied. The plan is valid if the following sequence can be proven:

$$\begin{aligned} \{IS\}\ plan\ \{GS\} \Leftrightarrow \{IS \wedge P_1\}\ a_1\ \{E_1\} \wedge \\ \{P_2\}\ a_2\ \{E_2\} \wedge \\ \wedge \dots \wedge \\ \{P_i\}\ a_i\ \{E_i \wedge GS\} \end{aligned}$$

Definition 18 (plan) *We define a plan as an ordered list of actions $P := \{a_1, ..., a_n\}$ that transfers a given initial state IS into the goal state GS, where an action a can occur multiple times within P.*

Example 6.5. Considering Example 6.4 from this Chapter, where in the initial state *size* equals zero, and in the goal state *size* equals two. The planning domain provides two actions, one of which increases *size*. The shortest plan that achieves the goal state from starting in the initial state will therefore contain three elements: One element referencing the constructor, and the other two elements referencing the *push* action, which increases *size*. One possible, in that case the shortest, solution will be plan $P := \{Stack, push\_Double_, push\_Double\}$. □

6.3.2. PDDL- Planning Domain Definition Language

The Planning Domain Definition Language was developed by Drew McDermott and the International Planning Competition (IPC) committee [Ghallab et al., 1998]. It's attempt is to provide a single language to define classical planning problems, to make planners comparable that participate in the International Planning Competition [Council, 2010]. PDDL is syntactically based on LISP and hence it is a prefix language. In PDDL, a planning task consists of: [Zehentner, 2010]

Objects that exist in the planning domain.

Predicates that describe the state of the objects in the world.

Actions that can change the state of objects.

Initial state which describes the state of the objects before starting the planning process.

Goal state which describes the state of the objects that form up the goal to be achieved.

These five components are divided up into two files: the domain file and the problem file. The planning domain file consists of the objects, predicates and operators that are available in the planning domain, whereas the problem file contains the initial and goal states for one specific planning problem.

Actions may change the state of the world by manipulating the values of the predicates and fluents (the representation of numbers within PDDL). An action contains at least the following properties: parameters, the precondition and the effect. The parameters state on which objects in the planning domain actions should be applied. The precondition must be true, to execute the given action. When the action was executed, it has changed the world. Thus, the effect describes how the world changes. Listing 6.1 shows an example PDDL action description.

```
(:action load
   :parameters (?o ?v ?l)
   :precondition (and (object ?o) (vehicle ?v) (location ?l)
      (at ?v ?l) (at ?o ?l))
   :effect (and (in ?o ?v) (not (at ?o ?l)))
)
```

Listing 6.1: An example of a PDDL-action. It describes how to load an object **o** from a location **l** into a vehicle **v**. The action is taken from the logistic problem (logistics-strips), a standard planning problem, which is part of the International Planning Competition [Council, 2010] domains.

The PDDL standard has evolved over the last years and got more and more features. Not every planner supports all features of PDDL.

6.4. Concept

AIANA automatically solves the problem of generating test input data that fulfills the precondition of the method under test by means of AI planning. First the Design by Contract™ specification is used to generate domain and problem description in PDDL [Ghallab et al., 1998]. Therefore, the Design by Contract™ specification of methods is used to generate PDDL action descriptions. The precondition of the method under test is used as goal state. Based on this information a state-of-the-art planner calculates a method sequence such that the object satisfies the precondition of the method under test. The resulting method sequences are used to automatically generate unit tests.

AIANA is the first fully automated AI planner based approach that can handle real Java programs - not only Java programs limited to Boolean variables. Algorithm 3 shows the four main steps of AIANA.

AIANA as described in Algorithm 3 is sound but not complete. AIANA calculates a method sequence to construct objects at test execution time. It uses the specification for calculating the sequence, but the original implementation when executing the unit test. Therefore, all tests generated emphasize only errors that are actually present in the implementation. But since AIANA can not handle all input

Algorithm 3 AIANA generation algorithm

```
Object AIana(DbC(C), Pmut)
{
  D := generateDomainFile(DbC(C));
  G={IS, GS} := generateProblemFile(C, Pmut);
  P := runPlanner(D, G);
  object := convertPlanToJava(P);
  return object;
}
```

specifications it is not complete. There might be errors in the implementation that cannot be detected with the help of AIANA.

For describing the planning domain, goal and the resulting plan, PDDL [Ghallab et al., 1998] is used. Therefore, the approach is independent of the used planner. Different planners may be used for different software components, and improved planners in future years will further improve the results and applicability of AIANA.

AIANA is separately executed for each object type parameter of the method under test. Therefore, the planning domain is based on the Design by Contract™ specification of the parameters type only. The goal state is the precondition of the method under test, since the purpose of AIANA is to provide object construction sequences that lead to object instances that satisfy the precondition of the method under test and therefore produce less meaningless unit tests.

6.4.1. Generation of Planning Domain

The planning domain consists of a list of actions, as given in Definition 14. Therefore, basically each method that changes the object state of the requested parameter type is transformed to an action. More precisely, the disjunctive normal form of the method behavior is calculated, and for each conjunctive clause an action is generated. This is necessary since a PDDL action may only have one effect. The procedure of calculating the disjunctive normal form of preconditions for plan actions is well known and described for example in Weld [Weld, 1999].

Example 6.6. Consider the following method specification:

```
@Also({
  @SpecCase(pre="param>5 ∨ param<0", post="@Return=false")
  @SpecCase(pre="param>=0 ∧ param<=5", post="@Return=true")
})
public boolean isWithinLimit(int param) {...}
```

Obviously, two PDDL actions have to be generated, one for each different postcondition. Since some planner do have problems with disjunctions in the precondition the first *@SpecCase* is split into two actions with the same effect clause.

```
(:action isWithinLimit_1
   :parameters (?param)
   :precondition (> (param) 5)
   :effect (not (isWithinLimit))
)
(:action isWithinLimit_2
   :parameters (?param)
   :precondition (< (param) 0)
   :effect (not (isWithinLimit))
)
(:action isWithinLimit_3
   :parameters (?param)
   :precondition (and (>= (param) 0) (<= (param) 5))
   :effect (isWithinLimit)
)
```

□

Therefore, we have to define the set of action methods, which are those methods that are considered to change the class' virtual state $VS(C)$.

Definition 19 (action method) *We define the set of action methods AM of a class* $C := \langle c, f, {}^{con}m^{pub}_{void} \cup {}^{con}m^{pub}_{prim} \cup {}^{con}m^{pub}_{nprim} \cup \ldots \cup {}^{abs}m^{pub}_{nprim}\rangle$ *as the union of public constructors and public non-pure methods. More formally, as*

$$AM(C) := \{sm \in c \cup {}^{con}m^{pub}_{void} \cup {}^{con}m^{pub}_{prim} \cup {}^{con}m^{pub}_{nprim} | !Pure(sm)\},$$

where $Pure(sm)$ *returns the pure value from the method contract* $mc := \langle mb, pure\rangle$ *from the class contract* $DbC(C) := \langle mf, inv, pp\rangle$.

Example 6.7. Consider the *Stack* from Figure 4.1. In Example 4.1 the set of constructors is given by $c_{default} := \langle Stack\rangle$ and the set of public methods (with void, primitive or object type return type) is given by $m^{pub} := {}^{con}m^{pub}_{void} \cup {}^{con}m^{pub}_{prim} \cup {}^{con}m^{pub}_{nprim} := \{push(double), pop(), size(), peek(), empty()\}$. Only $push(double)$ and $pop()$ are not annotated with *pure*. The set of action methods for *Stack* is given by

$$AM(Stack) := \{Stack(), push(double), pop(), peek()\}$$

□

Furthermore, we have to define the interpretation function, which converts Design by Contract™ pre- and postconditions to plan precondition and plan effects, respectively. This includes the infix to postfix conversion of all clauses, as well as mapping all specification elements and operators to their PDDL equivalent.

Definition 20 (interpretation functions) *We define* $\Phi_{pre}(P) \rightarrow a^{pre}_i$ *and* $\Phi_{post}(Q) \rightarrow a^{effect}_i$ *to be two interpretation functions for transforming a pre- and postcondition to an action precondition and action effect, respectively.*

Example 6.8. Using the Design by Contract™ specification of the *pop()* method from the running example in Figure 4.1 results in

$$\Phi_{pre}(size() > 0) = (>\ (size)\ 0)$$
$$\Phi_{post}(size() = @Old(size()) - 1) = (increase\ (size)\ 1).$$

□

The domain file for class C is generated as given in Algorithm 4.

Algorithm 4 Domain File Creation

```
1. for each action method am in C, i.e., $\forall am \in AM(C)$, do:
   a) for each method behavior of the action method, i.e., $\forall mb \in MB_{am}$ where $mb := \langle P_i, Q_i \rangle$, do:
      i. calculate the disjunctive normal form of the precondition, i.e., $DNF(P_i)$ where the disjunctive normal form is given by $DNF(P_i) := P_{i_1} \vee \dots \vee P_{i_k}$.
      ii. for each conjunctive clause, i.e., $P_{i_j} \in DNF(P_i)$, do:
         A. write an action declaration with a unique action name uaname and store the mapping from uaname to $P_{i_j}$ and am,
         B. write the parameter declaration with all parameters of am, i.e., $p_1, \dots, p_k$ from either the constructor $c := \langle cid, p_1, \dots, p_k \rangle$ or the method declaration $m := \langle mid, vis, ret, p_1, \dots, p_k \rangle$,
         C. write the precondition clause by interpreting the chained (see Def. 9), framed (see Def. 10), object-oriented (see Definition 13) specification, i.e., $\Phi(\boxed{OO(P_{i_j})^{\infty}})$.
         D. write the effect clause by interpreting the chained (see Def. 9), framed (see Def. 10), object-oriented (see Definition 13) specification, i.e., $\Phi(\boxed{OO(Q_i)^{\infty}})$.
```

In case one of the interpretation functions is not able to write a correct PDDL declaration due to limitations of the transformation process, the whole action is discarded. This can be the case if the postcondition does not exactly specify how the state changes (e.g., $size() := @Old(size()) + 1$), but may only describe how the state will look like (e.g., $size() := @Old(size())\%2$). Therefore, Algorithm 4 is sound but not complete.

Theorem 2 *Ignoring all actions, which pre- or effect clause can not be translated due to limitations of the expressive power of the PDDL language, does not produce false positives. In other words,* AIANA *is sound.*

Proof 5 *Each action in the planning domain can be used by the planner to reach the goal state. Not all actions have to be used. If some methods could not be converted to actions, the planning domain is reduced, in other words, the choices for the planner are reduced. If the planner is still able to find a solution in the reduced planning domain, it is a valid solution in the original domain as well.*

The following paragraphs explain some aspects of the specification transformation process with respect to data types and operator mappings.

Data Type Mapping PDDL only supports predicates and function types. Functions return fluents and can therefore be used to model all number types. Table 6.1 shows the mapping between Java data types and PDDL types.

Arrays and therefore strings can be modeled by functions as well, but only limited operations are supported since PDDL functions do not yet support fluents as parameters. Therefore, one may not access the value of the array at $index + 2$.

	predicate	functions
Boolean	✓	
byte		✓
character		✓
integer,short,long		✓
float,double		✓
string		✓(limited)
built-in array type		✓(limited)

Table 6.1.: Data Type Mapping from *Modern Jass*/Java to PDDL.

equality		arithmetic		Boolean	
$a < b$	(< a b)	$a * b$	(* a b)	$a \|\| b$	(or a b)
$a > b$	(> a b)	a/b	(/ a b)	$a\&\&b$	(and a b)
$a <= b$	(<= a b)	$a + b$	(+ a b)		
$a >= b$	(>= a b)	$a - b$	(- a b)		
$a = b$	(= a b)				
$a! = b$	(not (a=b))				

Table 6.2.: *Modern Jass* to PDDL operator mapping in preconditions

Operator Mapping The major limitation of the approach are the differences in the expressiveness of effect specifications. Therefore, Tables 6.2 and 6.3 show the necessary mapping from *Modern Jass* or Java operators to PDDL operators.

The arithmetic operators in the precondition are only accepted as operands in equality statements.

Example 6.9. The PDDL clause `(> (size) (+ 2.0 1.0))` in the precondition of an action is valid, but `(+ (size) 1.0)` is not. □

The effect of a PDDL action allows only conjunctions of arithmetic clauses as can be seen in Table 6.3. The arithmetic operators $*$, $/$, $+$, and $-$ are mapped to `scale-up`, `scale-down`, `increase`, and `decrease`, respectively. As operands they accept terms that use the mathematical operators as well.

Example 6.10. The PDDL specification `(increase (size) (+ 2.0 1.0))` is allowed in an effect clause of an action. □

equality		arithmetic		Boolean	
$a < b$	✗	$a * b$	(scale-up a b)	$a \|\| b$	✗
$a > b$	✗	a/b	(scale-down a b)	$a\&\&b$	(and a b)
$a <= b$	✗	$a + b$	(increase a b)		
$a >= b$	✗	$a - b$	(decrease a b)		
$a = b$	✗				
$a! = b$	✗				

Table 6.3.: *Modern Jass* to PDDL operator mapping in postconditions

The generated domain file for the *Stack* from Figure 4.1 is given in Listing 6.2.

```
(:action java_util_stack
   :precondition  (not (stack_instantiated))
   :effect (and (stack_instantiated) (and (assign (size) 0.0))))

(:action push_int
   :precondition (stack_instantiated)
   :effect (increase (size) 1.0))

(:action pop
   :precondition (and (stack_instantiated) (> (size) 0.0))
   :effect  (decrease (size) 1.0))

(:action peek
   :precondition (and (stack_instantiated) (> (size) 0.0))
   :effect  (assign (size) (size)))
```

Listing 6.2: The PDDL Domain for the *Stack* Example.

6.4.2. Generation of Planning Problem

The plan problem file provides information about the initial state *IS* and the requested goal state *GS* of the planning problem.

The initial state is the set of all variables used in the planning problem. The values are set to their types default values with respect to the programming language of the system under test, in our case Java. The postconditions of the constructors have to specify, which values they update.

The planning goal is defined by the precondition of the method under test. AIANA aims to generate test input data that satisfies the method under tests precondition, therefore we use it as planning goal. Since AIANA focuses on one parameter at a time, only those parts of the precondition are of interest that impose a constraint on the parameter, for which AIANA calculates an instantiation sequence by means of planning. Therefore, AIANA uses the STACI approach (see Chapter 5) to extract those parts of the precondition that describe the goal state with respect to the parameter under consideration. Algorithm 2 shows the extraction rules that remove all those parts *o* from the precondition that do not impose a constraint on the object *m* for which currently an instantiation sequence is calculated.

Example 6.11. Consider the following method with the given precondition and the *Stack* type from Figure 4.1:

```
@Pre("p1>0 && p2.size()>=2")
public void mut(int p1, Stack p2) {... }
```

AIANA is used to calculate an instantiation sequence for parameter $p2$ of type *Stack*. The first clause of the precondition $p1 > 0$ does not impose any constraint on the *Stack*. Therefore, the extraction rules from Figure 2 remove it. The final goal state is given by $p2.size() >= 2$. □

Algorithm 5 lists all necessary steps to create the problem file for one test.

Algorithm 5 Problem File Generator

```
1. extract constraint on the parameter under generation
2. write *init* declaration, which initializes all variables used in the domain file and the goal declaration to the default values
3. write *goal* declaration
```

Algorithm 5 uses the extraction rules from Algorithm 1 in Step 1. Therefore, it is sound and complete as discussed in Section 5 for the STACI approach.

The resulting problem file for the running example from Figure 4.1 is given in Listing 6.3.

```
(define (problem assign_test)
  (:domain assign_test)
  (:requirements :strips :fluents)
  (:init (= (size) 0.0))
  (:goal (and (>= (size) 2.0 )))
)
```

Listing 6.3: The PDDL problem file for the running example.

6.4.3. Create Plan

After generating the domain and problem description files in the PDDL syntax, any planner that supports PDDL can be used to generate a plan. Furthermore, AIANA can choose a different planner for different planning problems depending on the required functionality.

If the planner is able to find a solution, a plan in an IPC compliant format is returned. If no plan can be found either another planner may be used or the higher-level test data generation algorithm is informed to use a different approach.

The planning domain and problem description file generated for the running example given in Figure 4.1 are given in Listing 6.2 and 6.3, respectively. Using a planner that supports PDDL produces a plan similar to the one given in Listing 6.4. This plan was generated by *sgplan*, which we used for evaluating AIANA.

```
0.001: (STACK) [1]
1.002: (PUSH_INT) [1]
2.003: (PUSH_INT) [1]
```

Listing 6.4: An IPC-conform Plan.

6.4.4. Plan to Java Method Sequence

The last step of AIANA is to transform the plan (see Listing 6.4) to actual method calls in Java. Each action $a_i := \langle name_i, \emptyset, P_{i_j}, E_i \rangle$ can be uniquely mapped to a concrete Java method and a precondition.

Note, the plan does not contain any information about actual values for the parameters required by the identified Java methods. Therefore, the higher-level test generation tool is asked recursively to generate values for each required parameter. This time using the parameters type for generating the domain file and the precondition P_{i_j} of the identified method as goal state specification.

Example 6.12. Consider the generated plan from Listing 6.4. Formally the plan is given by

$$\begin{aligned} P := \{ & \langle push\_1, \emptyset, \text{"}true\text{"}, \text{"}size() = @Old(size(), int) + 1\text{"} \rangle, \\ & \langle push\_1, \emptyset, \text{"}true\text{"}, \text{"}size() = @Old(size(), int) + 1\text{"} \rangle \} \end{aligned}$$

To instantiate the *push()* method call AIANA has to generate a value for each parameter, in this case only a *double* value for the parameter of *push()*. Therefore, it recursively calls the test data generation algorithm and provides the precondition specification for the corresponding action, in this case "true". □

Using this recursive approach, simplifies the planning domain and reduces plan generation time. The recursion is stopped after *r* steps.

Our test generation framework provides an internal representation for object construction and method calls, in other words method sequences. This facility is used to (*a*) generate an internal representation for the method sequence based on the plan, (*b*) represent the parameters, which actual values are calculated recursively, (*c*) export the object generation sequence to a *JUnit* test.

6.5. Discussion

The following sections discuss the advantages, disadvantages and limitations of AIANA. Furthermore, a runtime analysis is presented.

6.5.1. Advantages

The main advantages of AIANA are that it generates (*a*) meaningful test input, (*b*) which describes a valid object state, and (*c*) that the approach is independent from the actual planner used.

meaningful test input AIANA uses the precondition of the method under test as goal state specification. Therefore, if AIANA is able to find a plan it is guaranteed to satisfy the method under tests precondition with respect to the parameter that is generated through AIANA. Since AIANA may not be applicable for all parameters of the method under test, for example AIANA does not produce primitive values, it cannot be guaranteed that only menaingful tests are generated.

valid object states AIANA models all methods that may change the object state as plan action by means of their Design by Contract™ specification. It is therefore guaranteed that only object states that are reachable through the public interface of the object are considered.

independence of planner Using PDDL to represent the planning problem keeps AIANA independent from the actual planner used. Therefore, improvements on the planner side immediately take effect on the performance of AIANA. Furthermore, in future work AIANA may decide to use different planner for different generation problems.

6.5.2. Disadvantages and Limitations

Limitations on the AIANA approach are two-fold. Either they arise from limitations of the PDDL language, or from limitations of the underlying planner.

The most important limitations are:

expressiveness Design by Contract™ and PDDL specifications have a different level of expressiveness. Design by Contract™ specifications feature full first-order logic, whereas PDDL requires an update description as effect clause. In other words, an effect clause has to specify how to calculate the next state variables with respect to the current state variables and may not only impose constrains on the next state.

quantifiers The current PDDL version does not support quantifiers. Therefore, any specification including quantified formula may not be transformed to a PDDL action.

arrays/strings Current PDDL versions do not support uninterpreted functions and do not allow to have parameters of type *function*. Therefore, it is not possible to model arrays and strings in PDDL.

PDDL support State-of-the-art planner are typically research prototypes and focus on a limited set of PDDL only. Furthermore, each planner interprets PDDL slightly different and does not always support the newest version.

In general, PDDL and planner are still under development and may receive major updates in the upcoming years. Therefore, the applicability of AIANA may improve.

6.5.3. Runtime Analysis

Theorem 3 *Creating the domain file is in $O(b*2^x*t*p)+O(b*2^x*s)$ time, where b is the number of method behaviors of the class, x is the number of state variables present in the system under test, t is the total amount of methods present in the system under test, and p is the maximum number of parameters of a method.*

Proof 6 *Steps 1 and 1.a) of Algorithm 4 can be combined to one iteration over all method behaviors of all actions of a method, which is given by b. For each of these method behaviors the DNF is calculated and a PDDL action is written. Calculating the DNF is in $O(2^x)$, where x is the total number of state variables present in the system under test. This results in the worst case in 2^x PDDL actions. Writing each of them (steps A and B) cost $O(1)$. Writing steps C and D is also in $O(1)$ but for these steps the cost of calculating the instantiated, chained, framed specification, which is $O(p)+O(t*p)+O(s)=O(t*p)+O(s)$, where p is the maximum number of parameters present for each method, t is the total amount of methods present in the system under test, s is the number of state variables of the class, as given in Lemmas 1, 2, and 3, respectively. Therefore the total runtime complexity is given by*

$$\begin{aligned} b*(O(2^x)+2^x*(O(1)+O(1)+2*(O(t*p)+O(s)+O(1)))) &= \\ O(b*2^x)+b*2^x*(O(t*p)+O(s)) &= \\ b*2^x*(O(t*p)+O(s)) \end{aligned}$$

Theorem 4 *The problem file generation (Algorithm 5) requires $O(c+m)$ time, where c being the clauses present in the precondition, and m being the number of variables present in the domain and problem file.*

Proof 7 *Extracting the constraint for the parameter under generation takes $O(c)$ where c is the number of clauses present in the precondition of the method under test, as given by Lemma 6.*
Writing the init *declaration requires $O(m)$ time with m being the number of variables present in the domain and problem file.*
Writing the goal declaration requires $O(1)$ since it is a simple transformation of Java syntax to PDDL syntax. This totals in

$$O(c)+O(m)+O(1)=O(c+m).$$

Theorem 5 *Algorithm 3 requires $(O(b*2^x*t*p)+O(b*2^x*s)+O(c+m))*O((lp)^{r+1})$ time under the assumption the planner terminates and returns a plan with an action sequence of length l.*

Proof 8 *Algorithm 3 consists of four steps.*
Step 1 and 2 are the domain file creation and the problem file creation which are in $O(b*2^x*t*p)+O(b*2^x*s)$ (see Theorem 3), and $O(c+m)$ (see Theorem 4), respectively, where b is the number of method behaviors of the class, x is the number of state variables present in the system under test, t is the total amount of methods present in the system under test, p is the maximum number of parameters of a method, c being the clauses present in the precondition, and m being the number of variables present in the domain and problem file.
In step 3 a planner is called. Since this step does not terminate in general, for further runtime investigation of Algorithm 3 we assume that it terminates and returns a plan of length l.
In the worst case of step 4 Algorithm 3 is called recursively for each parameter of each method call in the plan. It terminates when reaching a planning depth of r. Therefore, we have to calculate the recursive runtime complexity which is given by the sum of runtime complexities for steps 1 to 3, whereas step 3 is the recursive call to AIANA *for each parameter (maximum p) of each method call in the plan of length l:*

$$\begin{aligned} T(A) &= O(b*2^x*t*p)+O(b*2^x*s)+O(c+m)+l*p*T(A) \\ &= (O(b*2^x*t*p)+O(b*2^x*s)+O(c+m))*\sum_{i=1}^{r}(l*p) \\ &= (O(b*2^x*t*p)+O(b*2^x*s)+O(c+m))*\frac{1-(lp)^{r+1}}{1-lp} \end{aligned}$$

Since $l*p>>1$ one can simplify the result to

$$(O(b*2^x*t*p)+O(b*2^x*s)+O(c+m))*O((lp)^{r+1})$$

6.5.4. Comments

No Plan Found

AIANA heavily relies on the planner used for calculating a plan. Depending on the actual technology of the planner it may return (*a*) an optimal plan with respect to any given criteria, (*b*) any non-specific plan, or (*c*) no plan at all. AIANA does not yet care about the quality (with respect to any given criteria) of a returned plan as long as a plan is returned. If the planner is not capable of generating a plan it may be either caused by (*a*) the specification, or (*b*) the planner.

Specifications may be incomplete or contradictory. A specification may be incomplete if the goal state is not reachable through the given planning domain.

Example 6.13. Consider a precondition of the method under test, i.e., the goal state, that requires a *Stack* with 10 elements. And the actual Stack deployed is bounded to only eight elements. If the specification does state the bounding constraint, the goal is not reachable and the planner will therefore return no plan. □

The specification is contradictory if, for example, the specification is too weak.

Example 6.14. Consider again the method under test which requires a *Stack* of *length* 10, but the actual *Stack* deployed has no method specification that changes the *length* attribute. □

For example state space explosion may cause the planner to terminate without a result.

Planning Domain Construction

Scalability and dealing with unknown information are important for test data generation strategies. This sections presents two possible strategies for testing with AI planner, which we call: Pessimistic Test Data Generation and Optimistic Test Data Generation. The advantages and disadvantages of both strategies are pointed out before we conclude why AIANA implements the Optimistic Test Data Generation strategy.

```java
@Pre("stack != null && stack.size() >= 2")
public void methodUnderTest(Stack stack) { ... }

class ComplexType{
        @Pre("a > 0")
        @Post("hasA() = true")
        public void setA(int a) { ... }

        @Pre("b < 0")
        @Post("hasB() = true")
        public void setB(int b) { ... }
}

class Stack {
    @Post("size() = 0")
    public Stack() { ... }

    @Pre("type != null && type.hasA() && type.hasB()")
    @Post("size() = @Old(size(), int) + 1")
    public void push(ComplextType type) { ... }
}
```

Listing 6.5: An exemplary method under test.

Consider the example given in Listing 6.5, where *methodUnderTest* is the method under test.

To generate a unit test for *methodUnderTest(Stack)* a Stack objects needs to be instantiated. Furthermore, this Stack needs to contain at least two elements of type *ComplexType* as stated by the precondition of the method under test in Line 1. To *push* them onto the stack, those *ComplexType* instances have to be in a special state, as required by the precondition in Line 18.

AIANA may either generate a planning domain for the whole problem, or focuses on one instance at a time.

The former approach tries to generate a plan that (*a*) instantiates the *Stack* and two *ComplexType* objects, (*b*) mutates the *ComplexType* objects such that they satisfy the precondition of the *push* method, and (*c*) finally pushes the *ComplexType* objects onto the *Stack*. With only one planning step, this approach returns a plan that generates test data input that satisfies all specifications.

The latter approach splits the problem. It first generates a plan that generates test input that satisfies the precondition of *methodUnderTest*. This plan contains two *push* actions. Each requires a *ComplexType* object as parameter. To instantiate those *ComplexType* objects, AIANA is called recursively (once for each parameter). This time using only the specification of *ComplexType* to build the planning domain and using the precondition of *push* as goal state specification. This strategy is an iterative approach that tries to minimize each subproblem and calls itself or other data generation techniques later if necessary.

We call the former Pessimistic Test Data Generation, and the latter Optimistic Test Data Generation approach. Pessimistic Test Data Generation in the sense that the approach tries to solve everything at a time and therefore does not trust that it can solve the subproblem later. Optimistic Test Data Generation in the sense that this approach is optimistic about the future, it tries to solve first a subgoal and trusts that it get another chance later for solving another subgoal. Table 6.4 summarizes the advantages and disadvantages of both approaches. AIANA implements the Optimistic Test Data Generation approach.

Strategy	Advantages & Disadvantages
Optimistic:	+ Better failover capabilities.
	+ Reusable planning domains.
	− Slower, as planner might needs to be invoked more than once.
Pessimistic:	+ Faster, as planner needs to be invoked only once.
	− Planning domains must be generated for each problem individually.
	− If any substep fails, the whole generation procedure will fail.

Table 6.4.: Comparison of the two previously describes generation strategies.

6.6. Evaluation

AIANA was implemented as extension to *JET* [Cheon et al., 2005a, Cheon, 2007], which we also used as benchmark. *JET* was adapted to use *Modern Jass* [Rieken, 2007] Design by Contract™

	StackCalc		*StreamingFeeder*	
Metric	AIANA	Random	AIANA	Random
avg. (stdev.) succeeding	83.40 (5.24)	56.20 (0.98)	185.53 (2.97)	126.97 (2.21)
avg. (stdev.) failing	7.40 (5.24)	4.00 (1.26)	6.13 (2.97)	5.62 (0.66)
avg. (stdev.) meaningless	17.80 (2.40)	49.60 (1.02)	51.34 (2.98)	110.41 (1.98)

Table 6.5.: Number of generated tests of the *StreamingFeeder* case study. Using AIANA, the amount of meaningless test drops significantly.

	StackCalc		*StreamingFeeder*	
metric	AIANA	Random	AIANA	Random
overall	162.07	107.02	495.96	166.93
per successful test	1.78	1.77	2.59	1.25

Table 6.6.: Average test generation time in seconds over all 100 runs.

annotations, instead of *JML* [Leavens et al., 2002]. It aims on generating test data using a random strategy. We used *sgplan* [Hsu et al., 2006] as planner.

6.6.1. Experimental Setup

For each test run both approaches were asked to generate a unit test for each method of both case studies. Each approach has one single shot for each method under test. After test generation all tests are classified as either *succeeding*, *failing* or *meaningless* (see Section 1.3). Only those tests that are classified as *succeeding* or *failing* are exported to unit tests.

6.6.2. Results

AIANA is able to generate complex objects for both case studies. For the *StackCalc* case study AIANA is able to construct test input data that satisfies the precondition of the method under test for about *50%* more methods than random. For the *StreamingFeeder* case study AIANA is able to test *45%* more methods than random, as given in Table 6.5. The deviation for the AIANA approach is due to its still random generation of primitive parameter values.

Using a planner increases test generation time. Table 6.6 shows the average generation time for both case studies (overall) in seconds. In addition, row "per succ." present the average time required to generate one successful test. For the *StackCalc* case study the planning domain and the resulting plan are small and trivial. Therefore, the overhead is relatively small compared to the *StreamingFeeder* case study. For the *StreamingFeeder* case study AIANA takes three times as long as the random strategy. This is due to the complex object construction problem for the *StreamingFeeder* case study. The longest object construction sequence necessary to fulfill the method under tests precondition included 30 method invocations. Each of them requires again objects as parameters in a given state.

With respect to successfully generated tests the overhead decreases for both case studies.

Important to notice is that AIANA was able to test 67 methods from the *StreamingFeeder* case study, which random never did in any of the 100 experiment iterations. Manually inspecting those methods lead us to the conclusion that all those cases are methods where very complex objects have to be constructed. It is unlikely that a random approach chooses 30 times in a row the correct method. In turn the random strategy was able to generate a test for one method AIANA could not test. This is due to an unsupported Design by Contract™ specification. The AIANA implementation does not support the “instanceof” operator in a precondition specification, which our industry partner provided.

Chapter 7

SYNTHIA

Parts of the contents of this chapter have been published by Stefan J. Galler, Martin Weiglhofer, Franz Wotawa in "Synthesize it: From Design by Contract™ to Meaningful Test Input Data" at SEFM 2010 [Galler et al., 2010b].

7.1. Introduction

Testing object-oriented software is very complex since it requires to instantiate many classes to be able to execute the actual method under test. To make this process even more tedious, many constructors require even more objects as parameters. Even small unit tests require large parts of the system under test instantiated. This also means that all external dependencies of the system have to be satisfied.

Therefore, software developers/testers use *mock objects* to test complex applications. A *mock object* is an object that replaces the actual implementation by means of a "dummy" implementation. By now, the "dummy" implementation is written manually. It focuses on the relevant parts for the actual test. Therefore, a *mock object* reduces dependencies to the external environment and other objects of the system. *Mock objects* are especially useful in cases where the original object interacts with database systems, the file system or network components. Depending on the application different types of *mock objects* are useful: *Dummy*, *Stub*, *Mock*, *Spy*, and *Fake*. Details on this classification are given in Section 2.5.

SYNTHIA automatically synthesizes fake objects from Design by Contract™ specifications, which we call SYNTHIA FAKE. To maintain type compatibility a SYNTHIA FAKE object inherits from the original class. All public methods are overridden by an implementation that uses the pre- and postcondition of the method and an SMT solver to calculate the state changes due to the methods execution. Methods that return a value use the calculated state information to decide what to return. This eliminates the effort on writing the *Fake* object manually and the necessity to execute the actual implementation. Furthermore, the SYNTHIA FAKE object provides a mechanism to set the initial state of the *Fake*. A unit test can therefore easily instantiate and configure a SYNTHIA FAKE in a way such that it satisfies any given specification - in other words the precondition of the method under test.

This approach is further integrated in an automatic unit test generation tool. Therefore, for each object type parameter of a method under test a SYNTHIA FAKE is instantiated. In the initial version - results

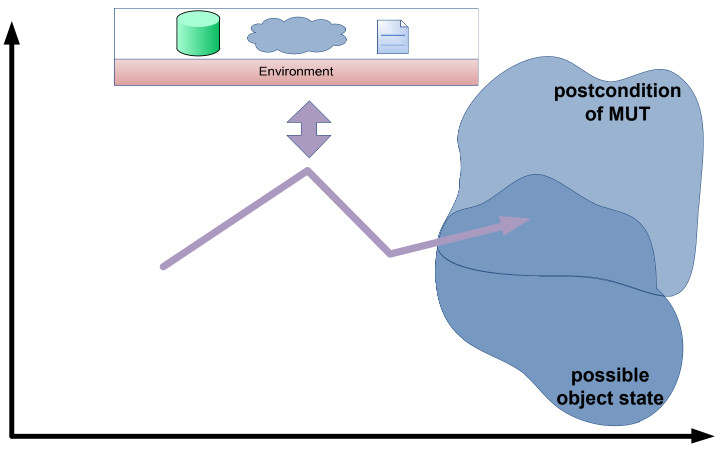


Figure 7.1.: SYNTHIA replaces the original implementation of all object type parameters of the method under test by one that uses an SMT solver to calculate the behavior according to the Design by Contract™ specification. Each of the axis models the value domain of a state variable of the object.

in this section are based on it - the STACI approach described in Section 5 is used to calculate the initial values for all SYNTHIA FAKE to satisfy the precondition of the method under test. Finally, those unit tests are exported.

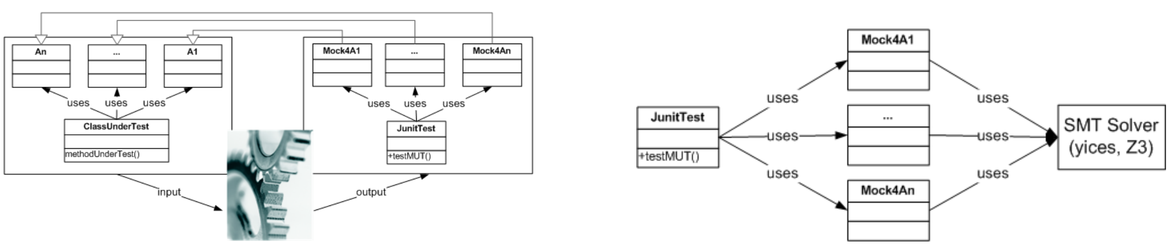
Figure 7.1 visualizes the approach. STACI is used to calculate the initial state for the SYNTHIA FAKE object. In contrast to STACI, a SYNTHIA FAKE object is dynamic. Therefore, it changes its state during test execution based on the methods called. Finally, it should end up in a postcondition satisfying state. Otherwise, a postcondition violation exception is thrown. This is triggered by the Design by Contract™ framework.

Using this approach for automated unit test generation has the following advantages:

- Less meaningless tests are generated, thus significantly higher function and line coverage can be achieved.
- Generated tests report only bugs in the method under test, not in the implementation of test input data.
- A method under test can already be tested before the final implementation of all parameter objects is finished.
- Dependencies to databases, network connections and the file system are removed.

7.2. Concept

The presented and evaluated approach automatically generates a unit test for each public method of the system under test. The unit tests instantiate a SYNTHIA FAKE object for each object type parameter of the method under test. The initial state of each SYNTHIA FAKE object is set such that it satisfies the precondition of the method under test. Finally the method under test is executed. The enabled Design



(a) SYNTHIA at Test Generation Time. All necessary SYNTHIA FAKE objects are synthesized based on their Design by Contract™ specification.

(b) SYNTHIA at Test Execution Time. *JUnit* uses the synthesized classes instead of the original implementation.

Figure 7.2.: Conceptual view on SYNTHIA FAKE.

by Contract™ checks determine whether the test succeeds or fails. The actual values for the initial state configuration are calculated at test generation time. The unit test therefore already contains the configuration source code.

At test execution time the method under test interacts with the public interface of the SYNTHIA FAKE objects. Instead of actually executing the real implementation, the SYNTHIA FAKE object calculates the executed methods behavior by means of its Design by Contract™ specification and an SMT solver. Thus, for the method under test there is no difference between the original implementation and the synthesized implementation of a class. However, the presented approach abstracts the SYNTHIA FAKE state by the set of the virtual state variables as defined in Definition 4. Therefore, any dependencies to other objects or the environment are eliminated.

The two perspectives of the presented approach - test generation time and test execution time - are showed in Figure 7.2.

This approach is based on the following observation: Given an initial state S, which constrains all relevant primed variables (e.g. $x' = 3$, $y' = 2$) and a method behavior $MB := \langle\langle P_1, Q_1\rangle, \ldots, \langle P_i, Q_i\rangle\rangle$ of method m of a SYNTHIA FAKE object, then the object's state after executing m is given by the primed variables of the sequential composition of S and MB as given by the expression:

$$S; ((P_1 \wedge Q_1) \vee \cdots \vee (P_n \wedge Q_n))$$

Example 7.1. Suppose we synthesize an implementation for the *pop()* method of the class *Stack* of Figure 4.1. The behavior is given by the predicate $(true \implies (size' = size - 1))$, which results after logically rewriting in $(size' = size - 1)$. Suppose we call *pop()* on a stack with two elements on it, i.e. our initial state is $size' = 2$ then we get

$$\begin{aligned}
(size' = 2); (size' = size - 1) &\qquad \text{(def. of ;)}\\
\exists s_0 \bullet s_0 = 2 \wedge size' = s_0 - 1 &\qquad \text{(one point rule)}\\
size' = 2 - 1 &\qquad \text{(def. of $-$)}\\
size' = 1 &
\end{aligned}$$

Thus, popping an element from a stack of size two leads to a stack of size one. □

Note that the result of this sequential composition can be simply obtained by using modern SMT solvers like yices [Dutertre and de Moura, 2006]. Therefore, we give the formula representing the instantiated sequential composition to the SMT solver. Because SMT solvers are incomplete when quantifiers are used we remove the leading existential quantification. If the existential quantified formula is true, then the formula without the quantifier is satisfiable. On the contrary, if the quantified formula is false, then the unquantified formula is unsatisfiable. Thus, removing the trailing existential quantifier of the sequential composition does not invalidate our approach.

7.2.1. Test Generation Time

At test generation time a SYNTHIA FAKE class file is generated for all object type parameters - if it does not yet exist. The SYNTHIA FAKE implementation depends only on static attributes of the class and is therefore independent of the usage scenario in the unit test class keeps track of the interaction sequence by means of the method behavior of each method call. References to parameter values in the method behavior specification are replaced by the actual values. Whenever a method has to return a value the SMT solver checks the sequential composition of all entries of the list for satisfiability and provides a model. The value provided by this model is returned.

The following sections describe the synthesized implementation of each SYNTHIA FAKE class in detail.

Synthesized Class Implementation

Since the automatically generated class will replace the actual implementation the types have to be compatible. Therefore, the generated class inherits from the required type, either an interface, an abstract class or concrete class. Consequently, the approach does not work for classes, which forbid inheritance (e.g., final classes in Java). Furthermore, the implementation of final or static methods cannot be replaced by the SYNTHIA FAKE approach. The generated class is placed in a different root namespace, so that there is no mixing between the hand written and the automatically generated implementation.

Let $C := \langle c, f, m\rangle$ be the class for which we synthesize an implementation. The synthesized class is generated as given in Algorithm 6.

Algorithm 6 SYNTHIA Class synthesis

```
1. create a public field $callSeq$,
2. write a constructor implementation (as described in Section 7.2.2) for each constructor $c$ in $C$,
3. write a method implementation (as described in Section 7.2.2) for each public method, i.e.,
   $m^{pub}_{void} \cup m^{pub}_{prim} \cup m^{pub}_{nprim}$,
4. write an empty method implementation for each protected or private abstract method, i.e.,
   ${}^{abs}m^{prot}_{ret} \cup {}^{abs}m^{priv}_{ret}$ with $ret \in \{void, prim, nprim\}$.
```

Test cases that feature SYNTHIA FAKE classes generated by Algorithm 6 and 7 are sound and complete with respect to the given specification, but not sound with respect to the implementation. This is due to the fact that a SYNTHIA FAKE class behaves according to the specification. Therefore, it is possible to generate tests that succeed on the real implementation but fail on the SYNTHIA FAKE instance due to a too weak specification.

The field *callSeq* is used to keep track of the method sequence called on the object. Each method call will add its method behavior specification to *callSeq* by means of sequential composition. Whenever a method has to return a value the SMT solver calculates a satisfying value to be returned based on *callSeq*. The *callSeq* member variable is public so that the initial state can be set directly from the *JUnit* test by adding a specification expression to *callSeq* that constrains the respective primed variable. By default *callSeq* holds a constraint that sets all state variables to their default value. Therefore, it is guaranteed that if the constructor does not provide information for all state variables, their value correlates to the value the programming language would initialize it with.

Mocking libraries typically provide static construction methods that return an actual mock instance. After the instantiation of a mock object, the behavior of it has to be configured. The constructor execution therefore does not change the behavior of the mock object, since it is overridden later through the configuration process. However, a SYNTHIA FAKE object should be used as if it is the actual object. Therefore, a client has to be able to call a constructor on the SYNTHIA FAKE object. Furthermore, the method behavior of the constructor is added to the *callSeq* member. The constructor method behavior and its actual parameter values influence the SYNTHIA FAKE objects behavior. Therefore, all constructors are overridden with implementations similar to those of public methods. In addition, all constructor parameter values have to be passed to the super constructor. Passing those values to the super constructor is necessary to satisfy the compiler (in our case the Java compiler) but does not have any influence on the SYNTHIA FAKE objects behavior, since all public methods are overridden in the SYNTHIA FAKE object and all return values are calculated by means of an SMT solver. Even if the super constructor stores the parameter values in member fields, these members are never accessed in the synthesized implementation.

7.2.2. Synthesized Method Implementation

As can be concluded from Section 7.2.1 two different types of synthesized method implementations are required: (*a*) one for public constructors and methods, (*b*) one for private or protected abstract methods.

Let $C := \langle c, f, m \rangle$ be the class for which we synthesize the implementation of a method $sm \in m$ and $MB := \langle \langle P_1, Q_1 \rangle, \dots, \langle P_i, Q_i \rangle \rangle$ the method behavior for *sm*. A synthesized method's body for public constructors and methods are generated as given in Algorithm 7.

Algorithm 7 SYNTHIA Method synthesis

```
1. Calculate the instantiated (see Definition 8), chained (see Definition 9), framed (see Definition 10), object-oriented (see Definition 13) method behavior, i.e., $\boxed{[OO(MB)]^{\infty}}$.
2. Calculate the sequential composition callSeq;$\boxed{[OO(MB)]^{\infty}}$ (without the leading existential quantification) and assign the result to callSeq.
3. If sm is a non-void method then pass callSeq to the SMT solver and ask for a model.
    • If sm returns a primitive type, return the corresponding value from the SMT model.
    • If sm returns an object type, instantiate and return a SYNTHIA FAKE object and set its initial state as given by the SMT model.
```

Generating the SMT problem at test execution time has the advantage of knowing the actual parameter values.

Example 7.2. Consider for example the *push(Double)* method from the *Stack*. Listing 7.1 shows the synthesized method for it.

```java
@Override
public Double push(Double value)  {
  ModelExpression mb =
    Synthia.getMethodBehavior("calculator.Stack", "push", "Double");
  ModelExpression chainedMB =
    Synthia.chain(mb.clone());
  ModelExpression framedChainedMB =
    Synthia.frame(chainedMB);
  ModelExpression instFramedChainedMB =
    Synthia.instantiate(framedChainedMB, value);
  callSeq =
    Synthia.sequentialComposition(callSeq, instFramedChainedMB);
  return Synthia.calculateReturnValue("push", "Double", callSeq);
}
```

Listing 7.1: Synthesized method implementation for *push(Double)* from the running example in Figure 4.1

Note that *getMethodBehavior* and *calculateReturnValue* require some additional parameters to be able to identify the method unambiguously. □

The dummy implementation synthesized for all abstract, either private or protected, methods is given in Listing 7.2. The return value and parameter list depends on the method signature. The methods return the default value of the corresponding return type.

```java
protected Double protectedPush(Double value)  {
  return 0.0d;
}
```

Listing 7.2: Synthesized method implementation for all private/protected abstract methods.

7.2.3. Test Execution Time

A typical unit test (*a*) instantiates the receiver object, (*b*) optionally transform the receiver object to the desired state, (*c*) instantiates all parameters required by the method under test, (*d*) calls the method under test, and (*e*) finally checks properties to determine whether the test succeeded or failed. Listing 7.2.3 shows a unit test for the *SummationOperator* from Figure 4.1, which was generated by our automated unit test generation framework and features SYNTHIA FAKE objects.

```java
@Test(timeout=60000)
public void testExecute() throws Throwable {
  calculator.SummationOperator var0 =
    new calculator.SummationOperator();
  calculator.Stack var1 =
 6      Synthia.createMock(calculator.Stack.class, new Object[]{  });
 7    java.util.List<JavaVariable> var2 =
 8      new java.util.ArrayList<JavaVariable>();
 9    JavaVariable var3 = new DoubleJavaVariable("peek$$$$");
10    var3.setValue(49.0);
11    var2.add(var3);
12    JavaVariable var4 = new IntegerJavaVariable("size$$$$");
13    var4.setValue(2);
14    var2.add(var4);
15    JavaVariable var5 = new DoubleJavaVariable("pop$$$$");
16    var5.setValue(45.0);
17    var2.add(var5);
18    JavaVariable var6 = new IntegerJavaVariable("listIterator$$$$");
19    var6.setValue(51);
20    var2.add(var6);
21    JavaVariable var7 = new DoubleJavaVariable("push$$value$$");
22    var7.setValue(47.0);
23    var2.add(var7);
24    Synthia.setInitial(var1, var2);
25    var0.execute(var1);
26  }
```

The receiver is instantiated in Line 4. The only parameter required by *execute*, the method under test, is a *Stack*, which is replaced with a SYNTHIA FAKE object in this test. The required initial state of it was calculated at test generation time, but the configuration is coded inside the unit test in Lines 8 to 24. Finally, the method under test is executed in Line 25.

Whenever the method under test calls a method on a SYNTHIA FAKE object and a return value has to be calculated the following steps are executed:

- the specification given by the *callSeq* member of the SYNTHIA FAKE object is transformed to an SMT problem,
- the SMT solver calculates the satisfiability of the problem and provides a model,
- the value that corresponds to the methods return value is extracted from the model.

SMT solving

The following paragraphs explain the transformation of the Design by Contract™ specification to an SMT problem in detail. The transformation process depends on the actual specification language and SMT solver used. The transformation presented here is based on the *Modern Jass* Design by Contract™ specification language for Java and the *yices* [Dutertre and de Moura, 2006] SMT solver with the following theories:

- uninterpreted functions
- difference logic
- linear integer arithmetic
- linear real arithmetic

- extensional arrays
- bit vectors
- first order quantification[*]

Data type mapping The *Yices*SMT solver supports bool, int, real data types and uninterpreted functions. Table 7.1 shows the mapping from Java data types to the corresponding *Yices* SMT types.

	bool	int	real	uninterpreted functions
Boolean	✓			
byte		✓		
character		✓		
integer		✓		
short		✓		
long		✓		
float			✓	
double			✓	
string				✓
built-in array type				✓

Table 7.1.: Data type mapping. How SYNTHIA maps from Java to SMT types.

Unfortunately, *Yices* does not provide built-in support for string data types such as other SMT solvers. Therefore, we have to model the string data type as character array. Arrays are modeled by means of uninterpreted functions, which are supported by *Yices*. Note that in the current implementation we do not provide a mapping for all string operators to operators on uninterpreted functions. Especially interesting will be mapping regular expressions to operations on uninterpreted functions, which is left for future work.

The array mapping is explicitly mentioned here, since arrays are built-in data types in Java. Lists, queues and other data structures are modeled as classes. Therefore, the presented approach expects Design by Contract™ annotations that specify the behavior of their methods. These behavior specifications may feature model fields of an array type.

Specification Elements Mapping All elements that occur in the specification but operators (see Section 2.1.1) are mapped to unique identifiers. The mapping from, for example, method calls on objects to unique identifiers is stored to be able to re-map values from the SMT model to the corresponding Java element.

Yices does support first order quantification, but not through the API we used for our implementation. Therefore, the implementation cannot deal with quantifiers in the specification. This limitation is based on the SMT solvers capability of solving quantified formulas and providing the required API interface. It is not a limitation of the conceptual design of the presented approach. Furthermore,

[*] First order quantification is considered as semi-decidable. *Yices* returns unsatisfiable if it fails to show satisfiability. [Dutertre and de Moura, 2006]

since SMT solvers generally have problems with quantifiers in specifications, future work can try to unroll all quantified expressions. Since the mapping from Design by Contract™ specifications to SMT problem formulas takes place at test execution time, all variable values are known at that point. Therefore, a *forall* loop over elements of an array may be unrolled to separate specification clauses for each element of the array.

7.3. Discussion

The following section discusses the advantages and disadvantages of SYNTHIA.

7.3.1. Advantages

SYNTHIA FAKE provides two major advantages over the simple use of *EasyMock* instances:

- dynamic behavior, and
- each method is immediately testable.

SYNTHIA FAKE is an automatically synthesized fake object (see Section 2.5), whereas STACI used a simple stub object. The difference is that a stub object will always return the same hard-coded values, which are configured at test generation time. One may configure a SYNTHIA FAKE object with an initial state, but in general it is not required. A SYNTHIA FAKE behaves from the moment of construction according to the Design by Contract™ specification of the original class.

Due to this feature SYNTHIA FAKE improves testability of the whole system from the very first moment of implementation. As soon as one method is implemented it can be tested, provided that all interfaces and classes are defined and specified in terms of Design by Contract™ specifications. Methods that use unimplemented classes can already be tested.

7.3.2. Limitations and Disadvantages

SYNTHIA FAKE removes all of STACIS limitations discussed in Section 5.3.2. The remaining two issues are (*a*) the possibility of unsatisfiable specifications, and (*b*) the limited applicability to non-final classes only.

The former limitation is already discussed in Section 7.3. The latter is an implementation issue only, which applies to all *mock objects*. To achieve type matching all mock objects inherit from the original class. This is not possible if the original class is marked *final* as supported by some programming languages such as Java.

7.3.3. Runtime Analysis

Theorem 6 *Synthesizing the class implementation is in $O(c*p)+O(m*p)+O(n*p)$ time, where p is the maximum number of parameters possible for a method, c is the amount of constructors, m the amount of public methods and n the amount of private or protected abstract methods of the class to be synthesized.*

Proof 9 *Step 1 in Algorithm 6 is in $O(1)$.*
Step 2 to 4 are each in $O(p)$, where p is the maximum number of parameters possible for a method. The implementation of a constructor, a method, or the empty method implementation is a string, where only the parameters and their types are replaced.
Synthesizing a class with c constructors and m public methods, and n private or protected abstract methods is therefore in

$$O(1) + c * O(p) + m * O(p) + n * O(p) = \\ O(c * p) + O(m * p) + O(n * p).$$

Theorem 7 *The runtime complexity at test execution time (see Definition 7) is in $O(t * p * d^p) + O(s)$ time under the assumption that the SMT problem is satisfiable and the SMT solver returns a model.*

Proof 10 *Calculating the instantiated, chained, framed, object-oriented method behavior is given by Lemma 5 with $O(t * p * d^p) + O(s)$ time, where t is the total amount of methods present in the system under test, p is the maximum number of parameters possible for a method, d being the domain of a parameters type, and s is the number of state variables of the class.*
Building the sequential composition in Step 2 is in $O(1)$.
Under the stated assumption that the SMT problem is satisfiable and the solver returns a model, Step 3 requires either $O(1)$ for extracting and returning a primitive value from the model, or in the worst case it has to synthesize on-the-fly a new class implementation which takes $O(c * p) + O(m * p) + O(n * p)$ time, where p is the maximum number of parameters possible for a method, c is the amount of constructors, m the amount of public methods and n the amount of private or protected abstract methods of the class to be synthesized (see Theorem 6).
Therefore, the overall runtime complexity is calculated by

$$O(t * p * d^p) + O(s) + O(1) + O(c * p) + O(m * p) + O(n * p) = \\ O(t * p * d^p) + O(s) + O(c * p) + O(m * p) + O(n * p).$$

The amount of all methods and constructors in the system under test t is typically much greater than the amount of constructors c, public methods m and private and protected abstract methods n of the class under synthesize ($t >> c$, $t >> m$, and $t >> n$). Therefore, the overall complexity of Algorithm 7 can be stated by

$$O(t * p) + O(s).$$

7.3.4. Comments

Since SYNTHIA FAKE objects inherit from the original class it is not applicable to final classes. In those cases *NULL* is returned and the above located test case generator either uses *NULL* or asks a different test data generation approach for object types to instantiate an object.

The SMT problem is build at test execution time. It might be unsatisfiable due to

- an unsatisfied precondition of a called method,
- an unsatisfied postcondition of a called method, or
- because quantifiers are present in the specification.

The precondition may never cause an unsatisfiable SMT problem, since at test execution time Design by Contract™ assertion checks are enabled. Therefore, the Design by Contract™ framework fires a precondition violation before the SMT problem is build and passed to the solver. In case of an unsatisfiable postcondition a postcondition violation exception is thrown.

The SMT solver may not be able to calculate a model in the presence of quantifier expressions (see Section 2.2). Quantifiers may occur either in the precondition or in the postcondition. Quantifiers in the precondition do not impose any problem, since they operate on the given prestate only. In other words, the variables are already determined and therefore it may not come to a state space explosion. Therefore, SYNTHIA FAKE may only fail if the postcondition contains quantifiers. Note, the SMT solver may still be able to come up with a model but it is not guaranteed due to limitations of the solver technologies. In the case of an unsatisfiable SMT problem SYNTHIA FAKE throws a designated exception. This exception is recognized and the test is marked as *meaningless* as well. Therefore, the user is not confronted with any unit tests that do not definitely emphasize an error in the implementation.

Regarding the specification two other issues are of interest:

- When should a method return a new instance?
- How regular expressions can be modeled as constraints on char arrays?

When should a method return a new instance? All methods of a SYNTHIA FAKE that return object types, return SYNTHIA FAKE objects as well. Therefore, the synthesized method implementation has to instantiate and configure a SYNTHIA FAKE object according to the current state of the corresponding state variable. Therefore, SYNTHIA FAKE always return new instances. But there may be methods that returns the same instance twice.

Example 7.3. Consider a stack class similar to the one presented in Figure 4.1, which holds objects and not primitive values. In that case, calling *peek* should return always the same instance. □

To distinguish those two cases *JML* introduced the *\fresh* keyword. Annotating a method with *\fresh* means that always a new instance is returned. Still, the absence of *\fresh* does not tell anything. Other Design by Contract™ specification languages still miss keywords to express at least those cases where a new instance has to be returned.

Finding a technique, including specification keywords, that helps test data generation tools to decide if a new instance has to be returned or the last value can be reused is part of future research.

How regular expressions can be modeled as constraints on char arrays? An SMT solver may either natively support the string type, or they are typically modeled as character arrays. A common way to specify the range of a string value is by means of regular expressions. Algorithms that generate a concrete string instance from a regular expression exist [Veanes et al., 2010], but they are not integrated in SMT solvers. Therefore, they have to instantiate all string type variables before or after the SMT problem is solved.

Interesting work can therefore be done on how to transform a regular expression into constraints understood by an SMT solver.

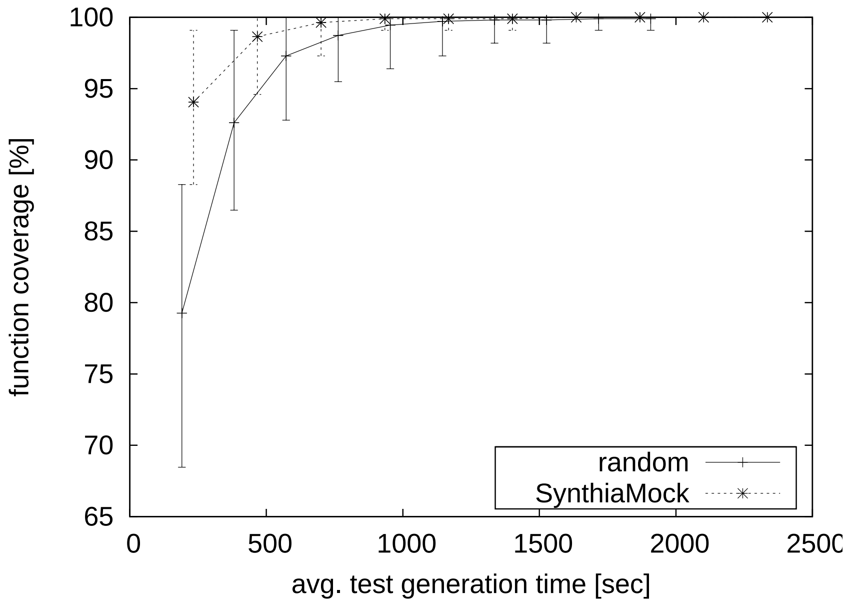


Figure 7.3.: *StackCalc* function coverage.

7.4. Empirical Evaluation

SYNTHIA is implemented as extension to *JET* [Cheon et al., 2005a, Cheon, 2007], which we use as benchmark as well.

7.4.1. Evaluation Setup

We ran our experiments on a computer with 2,2GHz AMD Athlon 64bit Dual Core CPU and 2GB RAM.

For our experimental evaluation we generated test input data for every method under test (MUT). First, we gave each approach (SYNTHIA FAKE and random) one chance to derive valid test input data. Recall, that valid test input data satisfies the MUT's precondition. We repeated this experiment 300 times and report on the minimum, maximum and average results (see Figures 7.3 and 7.4). Second, we gave each approach two tries to generate valid test input data (again 300 times). We repeated this procedure up to 10 tries per test data generation approach.

7.4.2. Results

SYNTHIA FAKE fulfills our expectation that it achieves 100% function coverage and outperforms the state-of-the-art random approach significantly in terms of function and line coverage with respect to generation time. The required test execution time heavily depends on the amount of interaction between the method under test and the automatically synthesized parameter objects.

Both, SYNTHIA FAKE and random, reach 100% function coverage on the *StackCalc* case study as shown in Figure 7.3. But SYNTHIA FAKE already tests on average 94% of all methods on the first

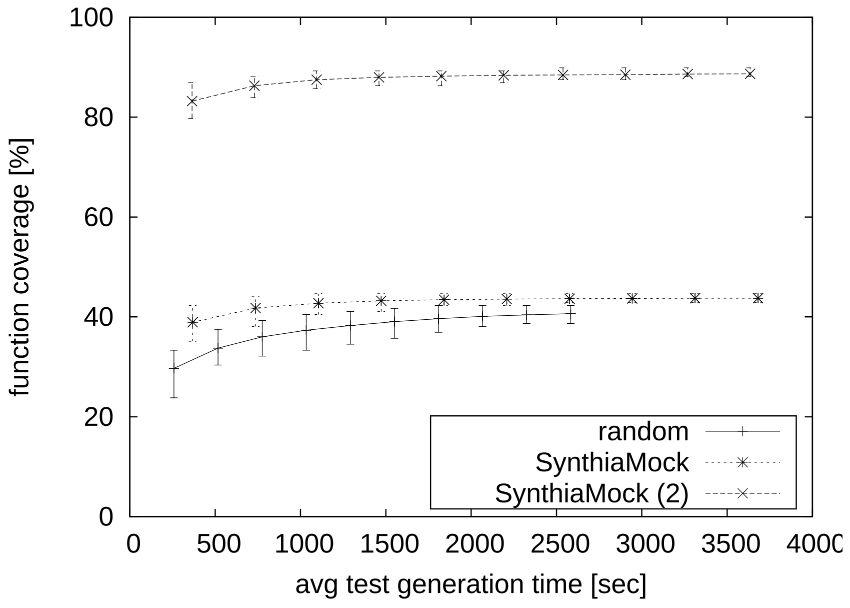


Figure 7.4.: *StreamingFeeder* function coverage.

	StackCalc		*StreamingFeeder*	
	gen./tc	exc.	gen./tc	exc.
random [sec]	1.77	0.24	4.10	0.53
SYNTHIA FAKE [sec]	2.12	0.47	5.11	0.47
+/- [sec]	0.35	0.23	1.01	-0.06
+/- [%]	19.77	95.83	24.63	-11.33

Table 7.2.: Generation and execution time per generated *JUnit* test.

attempt, whereas random requires more than twice as much attempts and time. The deviation on SYNTHIA FAKE is due to its still random integer generation approach for setting the initial state. Replacing this with a goal-oriented approach, e.g., an SMT solver, would result in 100% function coverage on the first attempt.

A similar picture is shown in Figure 7.4 for the industrial case study, *StreamingFeeder*. SYNTHIA FAKE again outperforms the random approach by at least ten percentage points. Still, on the original *StreamingFeeder* implementation SYNTHIA FAKE achieved about 42% function coverage only. Analysing the gap to 100% led us to five very weak pre-conditions on constructors, which did not specify exactly what the constructors require.

Figure 7.4 shows the results (SYNTHIA FAKE *(2)*) when adapting the implementation with respect to the weak pre-conditions. Still, SYNTHIA FAKE does not achieve 100% function coverage on this case study due to specifications that are not supported by our SYNTHIA FAKE implementation, such as the *instanceof* operator.

Using more sophisticated techniques for test data generation always requires more resources. Especially, since we use an SMT solver at test execution time, we have to discuss test execution time as well. Table 7.2 shows that the test generation time per successful test increases by about 22% on average. Whereas, the time required for test execution may change in either way. The reason for this inconsistent result is due to different interactions between a method under test and its parameters in both case studies. Whereas, each method under test in the *StackCalc* case study interacts heavily with the synthesized parameter objects, only little interaction can be observed in the *StreamingFeeder* case study. Since the SMT solver is called whenever an interaction exists, execution time suffers only in the *StackCalc* case study. The improvement in test execution time on the *StreamingFeeder* case study can be explained by the fact, that without any interaction the generated tests for the *StreamingFeeder* case study are smaller and less complicated. Less objects are constructed and no file system accesses are attempted.

Chapter 8

IntiSa

Parts of the contents of this chapter are submitted by Stefan J. Galler, Thomas Quaritsch, Martin Weiglhofer, Franz Wotawa as "The IntiSa approach: Test Input Data Generation for Non-Primitive Data Types by means of SMT solver based Bounded Model Checking" to QSIC 2011 [Galler et al., 2011]

8.1. Introduction

Even when using mock objects, such as SYNTHIA FAKE objects, for testing, one has to specify the initial state of those objects. Currently, mock objects are typically used for manually testing software. There are techniques such as STACI (see Chapter 5), which calculate the initial state required for mock objects, but they have one major drawback: STACI calculates an initial state that satisfies the precondition of the method under test, so that they reduce the amount of meaningless tests generated, but they do not guarantee that this calculated state would be reproduceable through the public interface of the object.

INTISA derives test input data from Design by Contract™ specifications, such that it, both, satisfies the precondition of the method under test and ensures that the generated input data represents a reachable object state in the program. As a side-product INTISA generates the method sequence that is required to bring the real object into this state. INTISA encodes possible state space changes of method calls for all parameters of the method under test. By means of an SMT solver this model is then verified against an adopted precondition of the method under test. In that way the SMT solver returns a valid object state and the methods to be called in order to reach this state.

Figure 8.1 shows this scenario, which is equivalent to the one showed for AIANA in Figure 6.1. The only difference is in the higher expression power of INTISA.

8.2. Preliminaries

INTISA is the only approach presented in this thesis that builds upon bounded model-checking. Therefore, a short introduction to model-checking by means of SMT solvers is given.

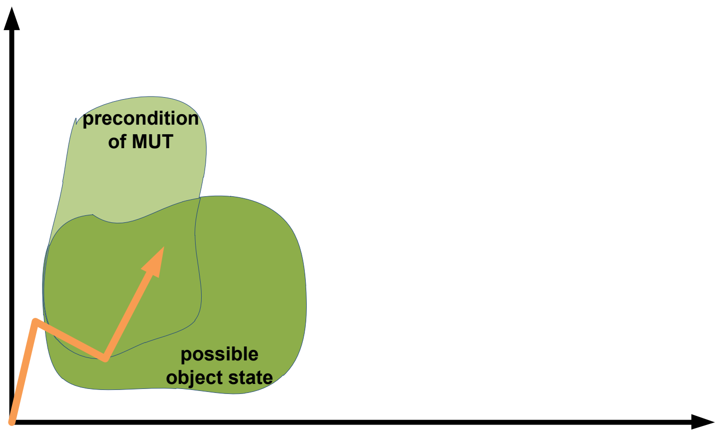


Figure 8.1.: INTISA uses bounded model-checking by means of SMT solver to calculate an initial state that satisfies the precondition of the method under test. Each of the axis models the value domain of a state variable of the object.

$$\begin{tikzcd} {} \arrow[r] & s_0 \arrow[r, bend left] & s_1 \arrow[l, bend left] \arrow[r, bend left] & s_2 \arrow[l, bend left] \arrow[r, bend left] & s_3 \arrow[l, bend left] \\ & \begin{array}{c} s=0\wedge \\ mS=0\wedge \\ e=true \end{array} & \begin{array}{c} s=1\wedge \\ mS=1\wedge \\ e=true \end{array} & \begin{array}{c} s=2\wedge \\ mS=2\wedge \\ e=true \end{array} & \begin{array}{c} s=3\wedge \\ mS=3\wedge \\ e=true \end{array} \end{tikzcd}$$

Figure 8.2.: A simple Kripke structure.

8.2.1. Bounded Model Checking via SMT Solver

Model-Checking is a technique to verify if a finite state machine model satisfies a specification. The specification is usually given in terms of a temporal logic (e.g. LTL or CTL). If the model does not satisfy the given specification, a counter example is generated.

For model-checking a finite state machine is represented by a *Kripke structure*.

Definition 21 (Kripke Structure) *A Kripke structure M is a tuple $M := \langle S, I, T, L\rangle$ where S is a set of states, $I \subseteq S$ is the set of initial states, $T \in S \times S$ is the transition relation and $L : S \rightarrow 2^{|A|}$ is a labeling function. A is the set of atomic propositions and $2^{|A|}$ denotes the power set of A. The labeling function attaches observations to states, i.e. $L(s)$ denotes the atomic propositions that hold in state $s \in S$.*

Example 8.1. Figure 8.2 shows a simple Kripke structure $M := \langle S, I, T, L\rangle$ with $S := \{s_0, s_1, s_2, s_3\}$, $I := \{s_0\}$, $T := \{(s_0, s_1), (s_1, s_0), (s_1, s_2), (s_2, s_1), (s_2, s_3), (s_3, s_2)\}$, and $L(s_i) =_{df} s = i \ \wedge mS = i$ where $i = 0, 1, 2, 3$. □

The sequential behavior of a Kripke structure is given in terms of paths, i.e., sequences of states.

Biere et al. [Biere et al., 2003] showed how to reduce the problem of bounded model checking to propositional satisfiability. This reduction allows to use efficient SAT solvers for bounded model-checking.

Bounded model-checking with a SAT solver of a Kripke structure M against a formula f with bound k is conducted by constructing a propositional formula $[\![M,f]\!]_k$. The constructed formula $[\![M,f]\!]_k$ encodes a constraint on a path $\pi := (s_0,\dots,s_k)$ such that $[\![M,f]\!]_k$ is satisfiable if π does not satisfy f. As LTL formulas are defined over all paths, M does not satisfy f, if we find such a π.

For the formulas involved in the IntiSa approach, we have the following translation formula for a Kripke structure and an LTL formula:

Definition 22 (Translation) *Given a Kripke structure $M := \langle S,I,T,L\rangle$, an LTL formula f and a $k \geq 0$, then*

$$[\![M,f]\!]_k =_{df} I(s_0) \wedge \bigwedge_{i_0}^{k-1} T(s_i,s_{i+1}) \wedge [\![f]\!]_k^0$$

where $[\![f]\!]_k^i =_{df} false$ if $i > k$ and otherwise:

$$[\![p]\!]_k^i =_{df} p(s_i) \qquad [\![f \vee g]\!]_k^i =_{df} [\![f]\!]_k^i \vee [\![g]\!]_k^i$$
$$[\![\neg p]\!]_k^i =_{df} \neg p(s_i) \qquad [\![f \wedge g]\!]_k^i =_{df} [\![f]\!]_k^i \wedge [\![g]\!]_k^i$$
$$[\![\boldsymbol{F}f]\!]_k^i =_{df} [\![f]\!]_k^i \vee [\![\boldsymbol{F}f]\!]_k^{i+1}$$

We use SMT solvers, which are an extension of SAT solvers. SMT solvers support arithmetics and other first-order theories like uninterpreted functions, and arrays [Dutertre and de Moura, 2006]. Hence, for our application, SMT solvers are better suited than pure SAT solvers as they allow a direct use of arithmetics.

8.3. Concept

IntiSa calculates initial values for all objects and primitive variables referenced by the method under tests precondition, such that

1. the precondition is satisfied, and
2. the calculated values compose a reachable state for all involved objects.

A reachable state is a state that can be achieved by calling public methods on the object.

Example 8.2. Consider the *Stack* and the precondition $stack.size() >= 2$ of the *sumUp* method from Figure 4.1. The value assignment $size() := 2, empty() := true$ composes a state which is satisfying but unreachable, since *empty()* may return true if and only if *size()* returns zero. A satisfying and reachable state is given by $size() := 2, empty() := false$. □

Existing test data generation techniques, such as constraint or SMT solvers, return satisfying states, but they may not be reachable. IntiSa, however, returns only states that are satisfying (the precondition of the method under test) and are reachable.

IntiSa achieves that by featuring the well-known technology of model-checking. Precisely, by SMT solver based model-checking as introduced by Biere et al. [Biere et al., 2003]. Existing testing approaches that feature model-checking focus on finding a state in the system under test in which a so called safety property is violated. In other words, checking against a safety property means looking for a state in which the property is violated. Therefore, these approaches model the system under test and return a method sequence for transforming the system under test into a state in which a property is violated.

INTISA does not model the system under test, but all parameters of a method call. For simplicity, we focus in this thesis on the call to the method under test, neglecting that the class under test has to be instantiated and transformed by method calls as well. But in general, INTISA is used for the empirical evaluation at each of those steps as well. INTISA also specifies a so called safety property, but with a slightly different meaning. The objective of INTISA is to find a state which satisfies the precondition of the method under test.

To achieve this INTISA has to (*a*) encode objects, (*b*) build the Kripke structure, and (*c*) specify the property to be checked. An SMT solver is then used to calculate the satisfiability of the given problem and returns a model. A model is a counter example, in other words, one possible value assignment to all variables of the problem, such that the formula is satisfied.

Algorithm 8 Calculating initial value assignments for testing method m

```
1. build SMT problem, i.e., $[\![M,f]\!]_k =_{df} I(s_0) \wedge \bigwedge_{i_0}^{k-1} T(s_i,s_{i+1}) \wedge [\![f]\!]_k^0$ by
    a) calculate the set of relevant objects, i.e., $s_0 := RO(m)$,
    b) set all state variables of the set of relevant objects to their default value, i.e., $I(s_0)$
    c) build the Kripke structure, i.e., $\bigwedge_{i_0}^{k-1} T(s_i,s_{i+1})$
    d) build the goal state formula, i.e., $[\![f]\!]_k^0$
2. let an SMT solver check it for satisfiability and
    • if the SMT problem is satisfiable, extract values from the model provided by the SAT
      solver for all relevant objects in the first step that satisfy the goal specification
    • else report to the test generation framework that no object state can be found that satisfies
      the precondition of the method under test
```

Those steps are explained in detail in Section 8.3.1. Furthermore, INTISA has to enhance the given Design by Contract™ specification similar to SYNTHIA (see Section 7).

The discussion of soundness and completeness for SYNTHIA applies for INTISA as well. Resulting test cases are sound and complete with respect to the specification but not sound with respect to the implementation. Both approaches, SYNTHIA and INTISA, rely on the given specification and SMT solver.

8.3.1. Building the SMT problem

The SMT problem to build from Definition 22 consists of the following parts :

1. calculating the set of *relevant objects*,
2. *initializing* all relevant objects to default values,
3. building the *transition relation* by means of a Kripke structure, and

4. building the *goal state* formula f, which describes the state that satisfies the precondition of the method under test.

Each of these steps is explained in detail in the following paragraphs.

Relevant Objects

The set of relevant objects is given by the list of objects required to be able to execute the method under test. Therefore, relevant objects are all object type parameters of the method under test and recursively all objects that are returned by methods of objects in this set of relevant objects.

Definition 23 (relevant objects) *The set of relevant objects RO is a set of tuples $ro_i := \langle id, C\rangle$, where id is an identifier and C denotes the type the identifier refers to. Given a method m with k object type parameters $np_1, \dots, np_k$ of types $C_1, \dots, C_k$ and ${}_{C_i}m^{pub}_{nprim}$, the set of public methods with object type return value of class C_i, the set of relevant objects is given by*

$$RO(m,l) =_{df} \begin{cases} \{\langle np_1, C_1\rangle, \dots, \langle np_k, C_k\rangle\} \bigcup\limits_{i=1}^{k} RO({}_{C_i}m^{pub}_{nprim}, l-1) & if\, l > 0 \\ \{\langle np_1, C_1\rangle, \dots, \langle np_k, C_k\rangle\} & else \end{cases}$$

Example 8.3. The *sumUp* method of the running example from Figure 4.1 has a single object type parameter $np_1 := stack$ of type $C_1 := Stack$. The set of public methods returning object types is the empty set ${}_{Stack}m^{pub}_{nprim} := \{\}$. The set of relevant objects therefore consists of only one tuple $RO(sumUp) := \{\langle stack, Stack\rangle\}$ □

The set of *relevant objects* is theoretically unlimited, since a method called on class A may return object B, where class B contains methods that may return an instance of class A. Therefore, the set calculation is aborted after l steps.

Initial state

All SMT variables are initialized with their default value of the programming language the system under test is written in. We do not initialize those values based on the constructor, because multiple constructor methods may exist that assign different values to the same state variable. Therefore, we model the constructor call similar to method calls in the Kripke structure. Still, this initialization process is required, since the constructor may not assign values to all state variables or the constructor specification may be incomplete.

Definition 24 *Given a method under test m the formula that initializes all state variables is given by*

$$I(s_0) =_{df} \bigwedge_{sv \in VS(RO(m))} sv := defaultValue(type(sv))$$

where type(X) *returns the type of state variable X and* defaultValue(T) *returns the default value of type T for the programming language in which the system under test is written in.*

Example 8.4. Given the virtual state $VS(Stack) := size, empty, peek, mSize$ from Example 4.2, INTISA initializes those for the SMT problem as follows: $size = 0 \wedge empty = false \wedge peek = 0.0 \wedge mSize = 0$. □

Kripke structure

The Kripke structure is obtained by transforming each method behavior of all methods provided by all relevant objects to a transition relation in such a way that only one method behavior after the other can be chosen at a time.

To ensure that exactly one method is executed at each step we add a Boolean variable bm_i. This guard does not only provide the construction sequence in terms of method calls, but is required to model a valid solution since we only talk about programming languages with sequential method execution. Therefore, the exclusive disjunction of all bm_i is added by means of conjunction to the transition relation for each step.

Definition 25 (transition relation) *Given a set of relevant objects RO, the Boolean guard variables $bm_{m_1}, \ldots, bm_{m_j}$, and their method behaviors $MB_1, \ldots, MB_j$, the transition relation $T(s_i, s_{i+1})$, where s_i denotes all pre-state variables, and s_{i+1} denotes all post-state variables, is given by*

$$\begin{aligned} T(s_i, s_{i+1}) =_{df} &(bm_{m_1} \implies \boxed{[OO(MB_1)]^{\infty}}) \wedge \cdots \wedge \\ &(bm_{m_i} \implies \boxed{[OO(MB_j)]^{\infty}}) \wedge \\ &(bm_{m_1} \oplus \ldots \oplus bm_{m_i}) \end{aligned}$$

Example 8.5. Consider the method behaviors $MB_{pop} := \langle\langle size() > 0, size()' = size() - 1\rangle\rangle$ and $MB_{push} := \langle\langle true, mSize' = mSize + 1\rangle\rangle$ from the two non-pure methods *pop* and *push* of the *Stack* from Figure 4.1. The transition relation contains the guarded method behavior for both methods and in addition an exclusive disjuncted (i.e., $\oplus$-ed) expression forcing exactly one guard to be true.

$$\begin{aligned} T(s_i, s_{i+1}) =_{df} &(bm_{pop} \implies size() > 0 \wedge size()' = size() - 1) \wedge \\ &(bm_{push} \implies true \wedge size()' = size() + 1) \wedge \\ &(bm_{pop} \oplus bm_{push}) \end{aligned}$$

Note, that this example does not calculate the instantiated, chained, framed method behavior for understandability reasons. □

goal state

The goal state formula is given by the precondition of the method under test. This precondition specifies the state of all relevant objects required by the method under test.

Since the goal may be reached in any of the with k bounded steps that are possible in the Kripke structure, the goal state formula has to check if it is satisfies in one of those steps. Therefore, the goal formula is a disjunction of the precondition of the method under test, where each of the disjunctive clauses reference the same variables at different steps. A boolean guard variable ($stepGuard_j$) for each step is introduced to be able to identify the step in which a state has been reached that satisfies the method under tests precondition. To make sure that the SMT solver tries to satisfy the goal state in at least one step, the disjunction of all guard variables is added to the overall goal formula by means of conjunction as can be seen in Definition 26.

Definition 26 (Goal State Formula) *Given the method behavior MB of the method under test the goal state formula is built as follows,*

$$[\![f]\!]_k^0 =_{df} \left(\bigvee_{step=0}^{k} stepGuard_{step} \wedge (P_1 \vee \cdots \vee P_n) \right)$$
$$\wedge \bigvee_{step=0}^{k} stepGuard_{step}$$

where $P_1, \ldots, P_n$ denote the n precondition specifications of MB and k the bound.

Example 8.6. Using the precondition of the method under test $P_{sumUp} := size() >= 2$ from Figure 4.1 and a bound $k = 3$ results in the following goal state formula

$$\begin{aligned}[\![f]\!]_2^0 := ((&stepGuard_0 \wedge size() >= 2)\vee \\ &(stepGuard_1 \wedge size() >= 2)\vee \\ &(stepGuard_2 \wedge size() >= 2) \\ &\wedge(stepGuard_0 \vee stepGuard_1 \vee stepGuard_2)\end{aligned}$$

□

Note, step guard $stepGuard_j$ is different from the method behavior guard bm_i in Definition 25. The step guard is not required to provide a valid solution but to identify the set of values that compose the state satisfying the goal condition. Whereas, the method behavior guard is required to provide a valid solution.

The SMT solver

An SMT solver is used to check satisfiability of the given Kripke structure and the safety property. In case it is satisfiable, the SMT solver returns a model. We extract the values of the step that satisfies the goal property, which can be identified by the $stepGuard$ variable. These values define the input values for all relevant objects for the call to the method (under test).

The advantage of SMT solvers is that they can be enhanced with theories, such as equality, non-linear and linear arithmetic, bit vectors and arrays. The availability of theories depends on the SMT solver. Our evaluation of INTISA is carried out with *Yices* [Dutertre and de Moura, 2006]. *Yices* supports uninterpreted functions, difference logic, linear real and integer arithmetic, extensional arrays, and bit vectors through the SMT-Lib interface. First order quantification is supported but considered as semi-decidable. *Yices* returns unsatisfiable if it fails to show satisfiability. Details about the usage of the *Yices*SMT solver for INTISA are equivalent to the usage in SYNTHIA. Therefore, everything stated in Section 7.2.3 applies to INTISA as well.

Objects that are not instantiated are mapped to object id zero. Therefore, INTISA can handle specifications that require objects to be null or not null. In one of the case studies, e.g., a specification requires one and only one of two parameters to be instantiated, but not both. INTISA is able to generate valid test input for it.

8.4. Discussion

INTISA is able to calculate instantiation sequences for object-oriented parameters of the method under test that satisfy the precondition of the method under test and are re-produceable through the public interface of the object. Unfortunately, it inherits some limitations from SYNTHIA since it builds on the same concepts of transforming the given Design by Contract™ specification to an SMT problem and solving it. In addition to a detailed discussion of those two points the following section includes a runtime analysis of INTISA.

8.4.1. Advantages

The main advantages of INTISA are that

- it calculates an initial state that satisfies the precondition of the method under test,
- the generated state is reproduceable through the public interface of the corresponding class, and
- therefore, it reduces the amount of meaningless tests generated, and
- furthermore, INTISA generates the initial state for all objects involved.

The initial state calculated by INTISA covers not only one parameter, but all variables involved. Even those primitive and object type parameters that are required to instantiate objects that in turn are again required as parameter by any method along the construction sequence. Precondition satisfying test input rule out any *meaningless* tests. Therefore, INTISA is very efficient in generating unit tests for a given method under test. Obviously the aim of all test input generation techniques should be to produce realistic value, in the sense that the actual objects involved may produce the same result. Otherwise, the test would test virtual behavior of a method, i.e., behavior that may not actually occur.

In addition to the values describing the initial state INTISA provides an ordered sequence of method calls that transforms the actual object into the goal state. Therefore, INTISA cannot only be used for configuring the initial state of a *mock object*, but can be used to instantiate the actual object and pass it to the method under test.

8.4.2. Limitations and Disadvantages

In addition to the challenge of possible unsatisfiable SMT problems, which are discussed in Section 8.4.4 two other issues have to be noted here:

- When should a method return a new instance?
- How can regular expressions be modeled as constraints on char arrays?

Both are already discussed in Section 7.3 and apply one-to-one for INTISA.

8.4.3. Runtime Analysis

Theorem 8 INTISA *calculates precondition satisfying initial state values for all relevant objects in* $O(l * m_{max} * p) + O(k * mbs * t * p) + O(k * mbs * s)$ *time under the assumption that the SMT solver returns a model.*

Proof 11 *Step 1a is bound to l iterations, in which for the maximum number of methods possible in a class* m_{max} *the maximum number of parameters p a method may have, a tuple containing the parameter and its type is added to the set of relevant objects. This is in* $O(l * m_{max} * p)$ *time.*
Step 1b builds the conjunction of default value assignments for all relevant objects. The assignment itself is in $O(1)$, *which is executed* $l * m_{max} * p$ *times, resulting in* $O(l * m_{max} * p)$.
The Kripke structure build in Step 1c contains one entry for all method behaviors mbs present in all relevant objects. Furthermore, a guard is introduced for each of the mbs and each method behavior is instantiated, framed, and chained. The former two steps are in $2 * O(mbs)$. *The latter is in* $O(t * p * d^p) + O(s)$, *where t is the total amount of methods present in the system under test, p is the maximum number of parameters present for each method, d being the domain of a parameters type, s is the number of state variables of the class, as given by Lemma 5. Therefore, the upper runtime complexity for all k transition steps in the Kripke structure is given by* $k * (2 * O(mbs) + mbs * (O(t * p * d^p) + O(s)))$ *which results in* $O(k * mbs * t * p + d^p) + O(k * mbs * s)$. *Putting all three steps together is in* $O(1)$ *and results in an overall runtime complexity for Algorithm 8 of* $O(l * m_{max} * p) + O(l * m_{max} * p) + O(k * mbs * t * p * d^p) + O(k * mbs * s)$, *which reduces to* $O(l * m_{max} * p) + O(k * mbs * t * p * d^p) + O(k * mbs * s)$.

8.4.4. Comments

There are two issues that may lead to no solution, in other words, an unsatisfiable SMT problem:

- the goal state is not reachable, and
- the specification is unsatisfiable due to limitations of the SMT solver.

goal state unreachable Three issues may cause the goal state to be unreachable. That are (*a*) a too weak specification, (*b*) a too small bound, and (*c*) an actually bounded implementation.

First, the given Design by Contract™ specification has to be strong enough to achieve the goal state. In other words, for all variables referenced in the goal state methods have to be specified that manipulate them.

Example 8.7. Consider for example the method under test and the *Stack* from Figure 4.1. The precondition of the method under test references the *size* method of the *Stack*. Therefore, the *Stack* has to provide at least one method that influences *size* (in favor of the precondition). There are two methods influencing *size*, but only *push* increases *size*. □

Second, the manually or hard-coded bound for building the Kripke Structure is too low. The basic concept of bounded model-checking is to find a path in a state machine that satisfies the given formula.

Due to the always present challenge of state explosion, bounded model checking reduces the problem to path with a given maximum size. The goal state may be well reachable with just one more step on the path, but this cannot be computed. Therefore, choosing a reasonable bound is crucial for the success of INTISA.

Third, the implementation may be actually bounded. Consider for example a bounded queue that may not hold more than x elements. If the goal state requires $x+1$ element the implementation may never reach the goal state. No matter how strong the specification might be.

SMT solver limitations SMT solvers are incomplete on quantified formulas. Since quantifiers may appear in any Design by Contract™ specification the approach may fail due to that limitation of the SMT solver. The discussion of the SYNTHIA approach (see Section 7.3) with respect to quantified SMT formulas applies here as well. In any of those cases INTISA is not able to calculate an initial state that both satisfies the precondition of the method under test and is reachable through the public interface of the corresponding class. Another approach has to be selected for calculating an initial state.

8.5. Experimental Evaluation

For our experimental evaluation we have implemented the INTISA approach by means of the *Yices* SMT solver [Dutertre and de Moura, 2006] and the *jConTest* [Quaritsch, 2010] test generation tool. *jConTest* automatically generates unit tests for all methods of a given Java application. The case study is specified in *Modern Jass*. For this evaluation we configured *jConTest* to combine the presented INTISA approach with SYNTHIA FAKE [Galler et al., 2010b] objects. In other words, for evaluating INTISA *jConTest* generates unit tests that instantiate SYNTHIA fake objects for all object type test input parameters, and configures their initial state with the values calculated by INTISA at test generation time.

8.5.1. Procedure

We compare INTISA with a random approach based on JET [Cheon et al., 2005a, Cheon, 2007]. The evaluation was conducted on servers with 2,2GHz AMD Athlon 64bit Dual Core CPU and 2GB RAM. Each approach gets n chances for each method under test to generate test input data. We start with $n = 1$ and increase n by one up to 10. This is due to the fact that random obviously will require more trials to generate precondition satisfying test, but on the other hand requires less time for one trial. We consider all public methods as methods under test. We do not generate tests for constructors. The generated tests are then exported to jUnit files, compiled and executed.

We present the results by plotting the achieved method coverage with respect to n, which can be also interpreted as with respect to the required time. By method coverage we understand the amount of methods covered by a unit test that satisfies the methods precondition with respect to the amount of methods under test. The presented results are average values over 30 runs, which is again to give the random a fair chance.

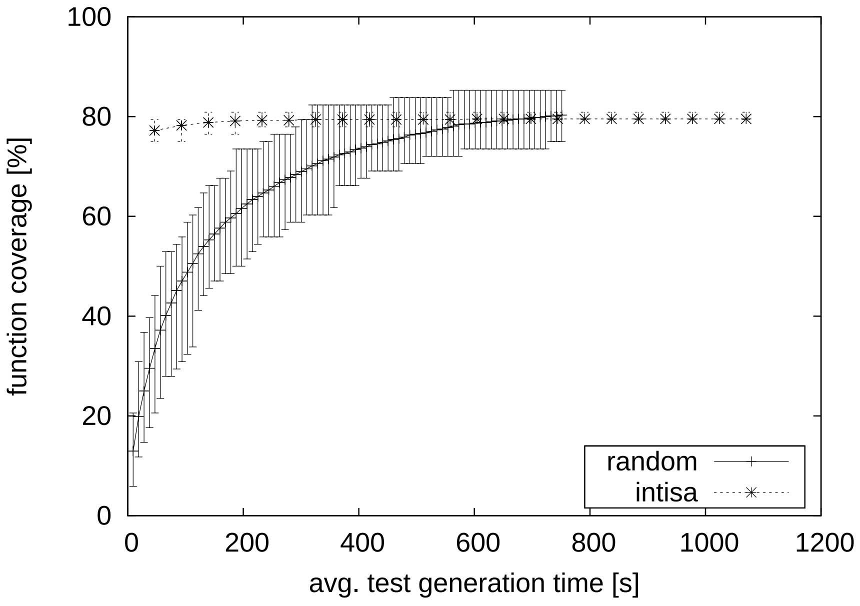


Figure 8.3.: *StackCalc*

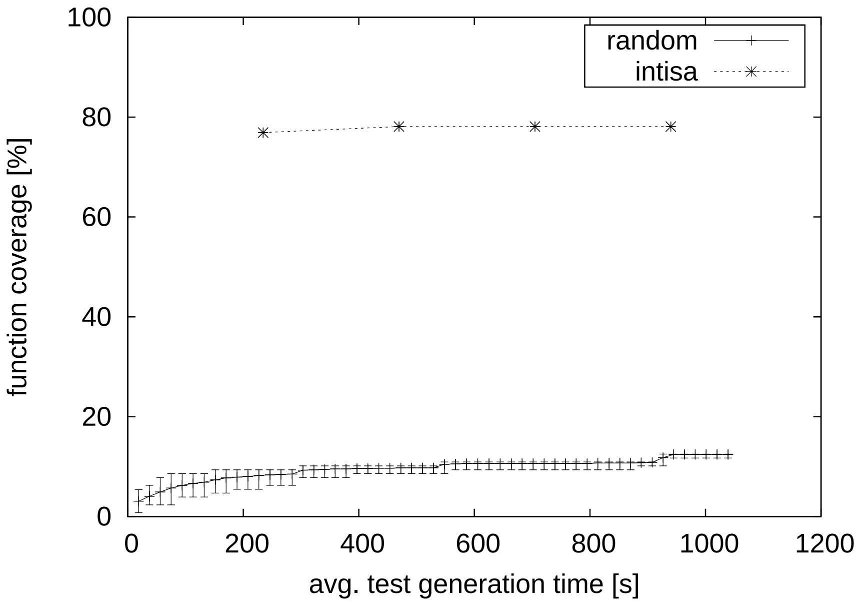


Figure 8.4.: *StreamingFeeder*

8.5.2. Empirical Results

INTISA is able to generate more tests that satisfy the precondition of the method under test, constantly and deterministically. And it achieves that high coverage with only one trial. The minimal improvement of INTISA over time as can be seen in Figure 8.3 and 8.4 is due to the fact that the implementation of INTISA still uses minor random decisions, e.g., for string values.

As expected, the random approach heavily improves over time. Especially for the structural less complex *StackCalc* random eventually catches up with INTISA and may probably even exceed the 80% of INTISA when given still more time. For the more complex real-world case study *StreamingFeeder*, random does not show that high improvement rate. This is mainly due to the more sophisticated in terms of constrained specification written for the industry case study.

INTISA still misses 20% points to cover all methods under test. This is due to (*a*) missing transformation of regular expression specifications to an equivalent SMT formula, (*b*) to specifications that are not supported by the tool that implements INTISA, such as the *instanceof* operator, and (*c*) due to runtime exceptions that may be thrown by methods under test.

Unfortunately, the results cannot be easily compared with the results of the pure SYNTHIA approach, since the specification of the case studies improved over time (we always used the newest specification version provided by the industry partner) and SYNTHIA was evaluated as a standalone implementation, not integrated into *jConTest*. *jConTest* was not available at that time.

Part III.

Discussion and Conclusion

Chapter 9

Conclusion

By now, in the year 2011, we are observing the second fundamental transformation process of information flow in history, it is the birth of the so called *Internet of Things* [Müller, 2011]. This means that after a decade and a half in which everybody got an computer and internet access, now every tiny piece of a *Thing* gets connected. The refrigerator connects with my mobile to remind me what I have to buy in the supermarket. In my car, the mobile informs the navigation system of my car that I have to do a detour to the nearest shopping center. Where the parking lot guidance system knows, based on my shopping list stored in the mobile, where I should park my car such that my shopping ways are more efficient. Finally, in the shop my mobile phone talks to the register to pay the bill.

With all those items interacting with each other, guiding and in the case of the car even controlling the way we live robustness, security and quality are becoming increasingly important. This increased demand and the high complexity of the involved systems makes testing already the most labour intensive and thus expensive task in software development. Therefore, research communities and companies such as Microsoft and Google, focus more and more on test automation. Test automation can be from as simple as automatically re-running manually generated tests and providing management-friendly reports to a fully automated test process which includes automatically generating (*a*) test input data, (*b*) test cases, and (*c*) test suites, (*d*) optimizing each of those steps with respect to given criteria, (*e*) running the generated tests, (*f*) classifying the test result in *succeeding*, *failing*, or *meaningless*, (*g*) and provide feedback to tester and management alike. Each of these possible automated generation steps are presented as one layer in the *Testing Pyramid* in Figure 9.1. It is pyramid since it only makes sense to talk about the next level when all levels below are solved. This thesis focuses on the first level, the test input data generation. It is the basis for all other testing approaches.

Having Design by Contract™ specifications present eases the automation of test generation. The postcondition of the method under test provides the test oracle, which classifies the test result. In addition, the precondition of the method under test can be used to steer the test data generation algorithm such that it generates only meaningful test input values, in other words values that satisfy the given precondition.

Nevertheless, initial experiments with case studies from our industry partner and then state-of-the art random approaches showed that only few meaningful tests are generated. The preconditions of the method under tests turned out to be too complex for the random based approach to generate values

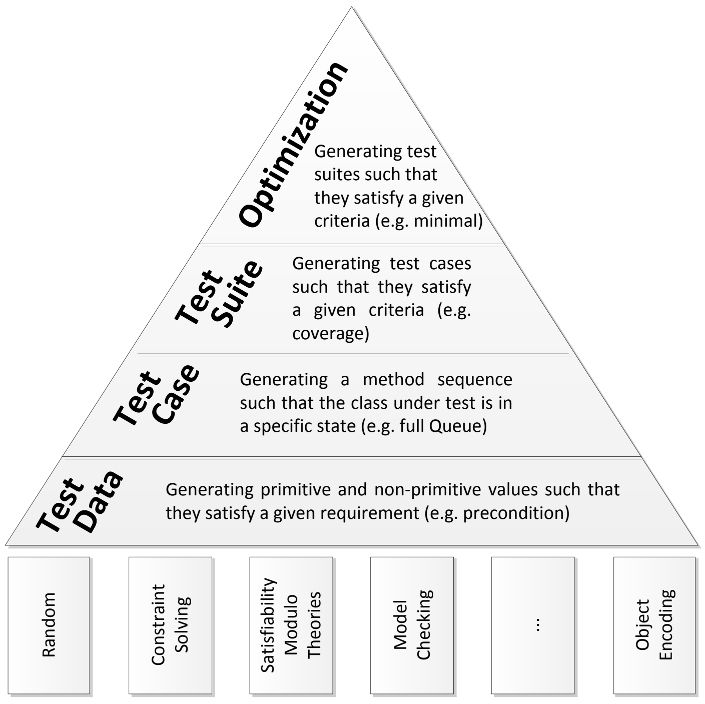
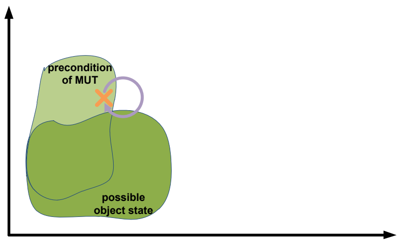
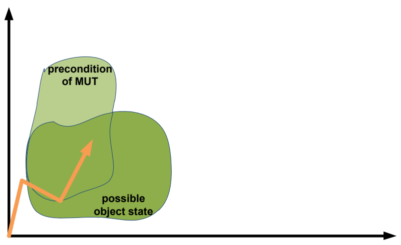

Optimization
Generating test suites such that they satisfy a given criteria (e.g. minimal)
Test Suite
Generating test cases such that they satisfy a given criteria (e.g. coverage)
Test Case
Generating a method sequence such that the class under test is in a specific state (e.g. full Queue)
Test Data
Generating primitive and non-primitive values such that they satisfy a given requirement (e.g. precondition)
Random
Constraint Solving
Satisfiability Modulo Theories
Model Checking
...
Object Encoding

Figure 9.1.: Testing Pyramid.

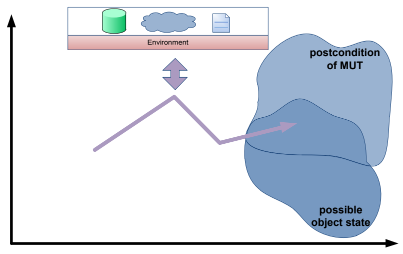
that satisfy it. From that observation I concluded that it is first necessary to focus on technologies that automatically generate test input data that satisfies any given specification, before working on any higher level of the *Testing Pyramid* or optimizing one of those steps.



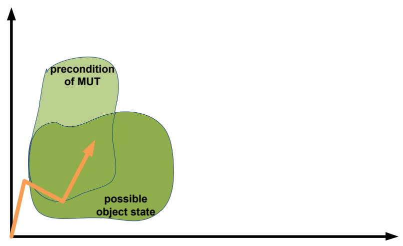
STACI features mock objects for all non-primitive parameter types of the method under test, which automatically configures the mock behavior based on the given Design by Contract™ specification. STACI only configures a static behavior of the mock, the approach can only be applied to one parameter of the method under test at a time and the configured state of the mock may not be reachable through the public interface of the original object.



AIANA is based on the observation that pre-/postcondition pairs of Design by Contract™ specifications and precondition/effect pairs of a STRIPS style action description in the AI planning domain are very similar. AIANA transforms the Design by Contract™ specification to a planning problem description by means of the Planning Domain Description Language (PDDL). State-of-the-art planner are then used to calculate a method sequence that constructs an object that satisfies the precondition of the method under test.



SYNTHIA is the first fully automated approach that on-the-fly generates stub objects that behave according to the Design by Contract™ specification of the original object. At test generation time the stub class is generated and compiled, at test execution time a SYNTHIA FAKE object keeps track of each method call and the actual parameter values. Whenever a return value has to be calculated for a SYNTHIA FAKE method, the Design by Contract™ specification of the called method sequence is transformed to an SMT problem and (if satisfiable) the corresponding value of the model provided by the *Yices* SMT solver is returned.



INTISA eliminates the major limitations of STACI and calculates initial values to be used for configuring SYNTHIA that satisfy the precondition of the method under test and model actual reachable states of all objects involved. Therefore, automatically generated unit tests that feature the combination of INTISA and SYNTHIA behave as the real object with respect to the Design by Contract™ specification. Furthermore, errors in the method under test which are hidden due to implicit assumptions of the programmer are reported.

This thesis has presented in its core a new approach for test input data generation for object types. Replacing objects with a synthesized implementation that behaves according to their specification improves automated test generation efficiency, since more meaningful tests can be generated in less time. In addition, the thesis showed especially through AIANA that even promising approaches do have limitations. Therefore, in the future we as a research community should work on making those limitations expressible in a formal way, such that future tools and approaches may choose which generation technique fits best for every single value to be created.

Chapter 10

Future Work

Current implementations of INTISA and SYNTHIA are still not able to generate test input that satisfies the precondition of the method under test for all given implementations and specifications. But the implementation of those concepts will improve through research success in the fields of SMT solving. Therefore, I suggest future work to focus

1. on optimizations of algorithms used in the *data generation* level,
2. on higher levels of the *Testing Pyramid*, and
3. on standardizing test data generation algorithm.

In the following I would like to briefly discuss (*a*) an optimization for test data generation algorithm based on techniques presented in this thesis and elaborated by Thomas Quaritsch in his master's thesis "Test Data Generation using Static Call Sequence Analysis and Design by Contract™ Specifications" [Quaritsch, 2011], and (*b*) an idea of creating a standardized test data generation description language, simliar to PDDL.

10.1. TESSAN

All presented approaches in this thesis do not yet focus on optimizing the test data generation process with respect to any criteria. This is due to my understanding of research: one step after another, as visualized in the *Testing Pyramid* in Figure 9.1. But as soon as you are able to generate test data for any given specification it is necessary to think of optimizing this process with respect to a given criteria. In the case of TESSAN the criteria is the capability of revealing an error in the system under test.

Consider a method M_1 with precondition P_1 and a method M_2 with precondition P_2. Furthermore, M_2 is called from M_1 and a parameter *param* of M_1 is passed to M_2. In Listing 10.1 method M_1 is *methodUnderTest* and M_2 is named *anyOtherMethod*.

```
@Pre("anObject.getSomething() > 42")
public void methodUnderTest(SomeClass anObject, OtherClass ...) {
 3          ...
 4          anObject.doSomething1();
 5          anObject.doSomething2();
 6          ...
 7          otherMethod(anObject);
 8          ...
 9  }
10
11  @Pre("anObject.getSomething() > 45")
12  public void otherMethod(SomeClass anObject) {
13          ...
14  }
```

Listing 10.1: Principal structure of the precondition mismatch problem [Quaritsch, 2011]

The preconditions P_1 in Line 1 and P_2 in Line 11 constrain *anObject*. TESSAN tries to find an object state for *anObject* that satisfies the precondition P_1 but violates P_2. Thus, having found an test input value that emphasizes a precondition mismatch. Sometimes the precondition mismatch will point out an specification error, other times it will actually reveal an implementation error in the method under test. The idea is similar to pre-/postcondition slicing [Harman et al., 2002]. Instead of eliminating lines from the source, as done in slicing, TESSAN models the source as SMT problem and tries to find an input value that violates the postcondition. Therefore, the result is not a set of lines which are correct, but a test case that highlights a specification mismatch and possible reveals an error.

TESSAN is a static analysis approach that uses a system dependence graph of the program to extract sequences of modifying instance method calls on parameter *param* in M_1 before *param* is passed to M_2. Both method preconditions and the call sequence inside M_1 before the call to M_2 are used to construct an SMT formula that is satisfiable if a precondition mismatch is feasible. From the SMT Solver results, a test case is generated which demonstrates the error to the programmer. [Quaritsch, 2011]

10.2. Test Data Generator Description Language (TDGDL)

Around the world many research groups at universities and in corporations work on test data generation algorithms. They are based on different base technologies, such as random, genetic algorithms, constraint solvers, or AI planners. But they all have in common that there are circumstances in which they produce good results, but still no technology is able to produce *always* perfect results. Each of those research groups tries to improve their technologies to reduce the limitations and disadvantages, but no single technology will solve the problem of test data generation ultimately.

Therefore, I suggest to develop a common language that allows to describe the capabilities of different test data generation approaches similar to the Planning Domain Description Language (PDDL). Based on such a common description language, the *Test Data Generation Description Language*, approaches and tools may be developed that are able to combine different data generation algorithms and choose the best fitting approach every single time a new value has to be generated in an automated test generation process. This description language can be based on the work of Aguirre and Maibaum [Aguirre and Maibaum, 2003], who work on languages to reason about component-based systems. this allows them to reconfigure large systems at runtime - similar to the idea of reconfiguring the test data generation approach at test generation time.

The test generation framework *jConTest* used for evaluating the approaches presented in this thesis is already designed to handle completely different test data generation algorithm. The design allows to extend the framework with an additional level on the top, which knows about different test data generation technologies capabilities and can decide based on given criteria (which are evaluated on-the-fly), which data generation algorithm fits best for which input parameter.

Therefore, the proposed language should be able to express attributes such as:

type the type of the value to be generated, e.g., integer, object, java.util.List, my.Object.

specification which specification language the algorithm understands, what subset of keywords of the specification language it supports, e.g., *Modern Jass* without the keywords @*Min* and @*Max* (which are only syntactic sugar).

resources a list of resources required, i.e., SMT solver, file system.

runtime/memory complexity the upper/lower bound of runtime and memory complexities.

The given list is an initial proposal and should be extended as needed through out the research process.

Bibliography

[Aguirre and Maibaum, 2003] Aguirre, N. and Maibaum, T. (2003). A logical basis for the specification of reconfigurable component-based systems. In *Proceedings of the 6th international conference on Fundamental approaches to software engineering*, FASE'03, pages 37–51, Berlin, Heidelberg. Springer-Verlag. (Cited on page 136.)

[Ammann and Offutt, 2008] Ammann, P. and Offutt, J. (2008). *Introduction to Software Testing*. Cambridge University Press. (Cited on pages 3, 14, and 63.)

[Arts et al., 2006] Arts, T., Hughes, J., Johansson, J., and Wiger, U. (2006). Testing telecoms software with quviq QuickCheck. In *Proceedings of the 2006 ACM SIGPLAN workshop on Erlang*, Portland, Oregon, USA. ACM. (Cited on page 50.)

[Atterer, 2000] Atterer, R. (2000). *Automatic Test Data Generation From VDM-SL Specifications*. Phd thesis, The Queen‘s University of Belfast. (Cited on page 25.)

[Barnett, 2010] Barnett, M. (2010). Code Contracts for .NET : Runtime Verification and So Much More. *Lecture Notes in Computer Science*, 6418/2010(Runtime Verification):16–17. (Cited on page 10.)

[Barnett et al., 2004] Barnett, M., DeLine, R., Fähndrich, M., Leino, K. R. M., and Schulte, W. (2004). Verification of Object-Oriented Programs with Invariants. *Object Technology*, 3:2004. (Cited on page 17.)

[Barnett et al., 2009] Barnett, M., Fähndrich, M., de Halleux, J., Logozzo, F., and Tillmann, N. (2009). Exploiting the synergy between automated-test-generation and programming-by-contract. In *Proceedings of 31th International Conference on Software Engineering*. Citeseer. (Cited on pages 18 and 48.)

[Barnett et al., 2005] Barnett, M., Leino, K., and Schulte, W. (2005). *The Spec# Programming System: An Overview*, pages 49–69. Springer Berlin. (Cited on pages 10, 16, 17, and 52.)

[Biere et al., 2003] Biere, A., Cimatti, A., Clarke, E., Strichman, O., and Zhu, Y. (2003). Bounded model checking. *Advances in Computers*, 58. (Cited on pages 118 and 119.)

[Board, 2007] Board, I. S. T. Q. (2007). Standard glossary of terms used in Software Testing. (Cited on page 4.)

[Bordeaux et al., 2006] Bordeaux, L., Hamadi, Y., and Zhang, L. (2006). Propositional Satisfiability and Constraint Programming: A comparative survey. *ACM Computing Surveys*, 38(4). (Cited on page 23.)

[Boshernitsan et al., 2006] Boshernitsan, M., Doong, R., and Savoia, A. (2006). From Daikon to Agitator: Lessons and Challenges in Building a Commercial Tool for Developer Testing. In *Proceedings of the 2006 international symposium on Software testing and analysis*, pages 169–180, New York, NY, USA. ACM Press. (Cited on pages 25 and 35.)

[Bourdonov et al., 2002] Bourdonov, I. B., Kossatchev, A., Kuliamin, V. V., and Petrenko, A. (2002). *UniTesK Test Suite Architecture*, volume 2391/2002 of *Lecture Notes in Computer Science*, pages 121–152. Springer. (Cited on page 52.)

[Boyapati et al., 2002] Boyapati, C., Khurshid, S., and Marinov, D. (2002). Korat: Automated testing based on Java predicates. In *Proceedings of the 2002 International Symposium on Software Testing and Analysis (ISSTA), Rome, Italy, July 22–24, 2002*, pages 123–133. ACM Press New York, NY, USA. (Cited on pages 25 and 51.)

[Brat et al., 2004] Brat, G., Drusinsky, D., Giannakopoulou, D., Goldberg, A., Havelund, K., Lowry, M., Pasareanu, C., Venet, A., Visser, W., and Washington, R. (2004). Experimental Evaluation of Verification and Validation Tools on Martian Rover Software. *Formal Methods in System Design*, 25(2/3):167–198. (Cited on page 51.)

[Cadar et al., 2008a] Cadar, C., Dunbar, D., and Engler, D. (2008a). KLEE Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *USENIX Symposium on Operating Systems Design and Implementation*. (Cited on pages 25 and 48.)

[Cadar et al., 2008b] Cadar, C., Ganesh, V., Pawlowski, P. M., Dill, D. L., and Engler, D. R. (2008b). EXE: Automatically Generating Inputs of Death. *ACM Transactions on Information and System Security*, 12(2):322–335. (Cited on pages 25, 51, and 52.)

[Campbell et al., 2008] Campbell, C., Grieskamp, W., Nachmanson, L., Schulte, W., Tillmann, N., and Veanes, M. (2008). *Model-Based Testing of Object-Oriented Reactive Systems with Spec Explorer*, pages 39–76. Springer Verlag. (Cited on pages 25 and 52.)

[Cheon, 2007] Cheon, Y. (2007). Automated Random Testing to Detect Specification-Code Inconsistencies. Technical report, Department of Computer Science The University of Texas at El Paso, 500 West University Avenue, El Paso, Texas, USA. (Cited on pages 81, 100, 114, and 126.)

[Cheon et al., 2005a] Cheon, Y., Kim, M., and Perumendla, A. (2005a). A complete automation of unit testing for java programs. In *Proceedings of the 2005 International Conference on Software Engineering Research and Practice*, pages 290–295. CSREA Press. (Cited on pages 7, 81, 100, 114, and 126.)

[Cheon et al., 2005b] Cheon, Y., Leavens, G., Sitaraman, M., and Edwards, S. (2005b). Model variables: cleanly supporting abstraction in design by contract: Research Articles. *Softw. Pract. Exper.*, 35(6):583–599. (Cited on pages 12 and 21.)

[Cheon and Rubio-Medrano, 2007] Cheon, Y. and Rubio-Medrano, C. E. (2007). Random Test Data Generation for Java Classes Annotated with JML Specifications. Technical report, Department of Computer Science The University of Texas at El Paso, 500 West University Avenue, El Paso, Texas, USA. (Cited on page 11.)

[Ciupa and Leitner, 2005] Ciupa, I. and Leitner, A. (2005). Automatic testing based on design by contract. *Proceedings of Net. ObjectDays*, pages 545–557. (Cited on page 38.)

[Ciupa et al., 2008] Ciupa, I., Leitner, A., Oriol, M., and Meyer, B. (2008). ARTOO: Adaptive Random Testing for Object-Oriented Software. In *Proceedings of the International Conference on Software Engineering 2008*. (Cited on page 39.)

[Cooperation, 2009] Cooperation, M. (2009). *Code Contracts User Manual*. Microsoft Cooperation. (Cited on pages 18 and 20.)

[Council, 2010] Council, I. E. (2010). ICAPS Conference. Available online at `http://ipc.icaps-conference.org`, last accessed: 04/2010. (Cited on pages 88 and 89.)

[de Halleux and Tillmann, 2010] de Halleux, J. and Tillmann, N. (2010). *Moles: Tool-Assisted Environment Isolation with Closures*, volume 6141 of *Lecture Notes in Computer Science*, pages 253–270. Springer Berlin Heidelberg, Berlin, Heidelberg. (Cited on page 48.)

[de Moura and Bjø rner, 2008] de Moura, L. and Bjø rner, N. (2008). *Z3: An Efficient SMT Solver*, volume 4963/2008 of *Lecture Notes in Computer Science*, chapter 24, pages 337–340. Springer Berlin, Berlin, Heidelberg. (Cited on pages 7, 24, 48, and 49.)

[de Moura and Bjø rner, 2009] de Moura, L. and Bjø rner, N. (2009). *Satisfiability Modulo Theories: An Appetizer*, volume 5902 of *Lecture Notes in Computer Science*, pages 23–36. Springer Berlin. (Cited on pages 23 and 24.)

[Dhara and Leavens, 1996] Dhara, K. and Leavens, G. T. (1996). Forcing behavioral subtyping through specification inheritance. In *Proceedings of the IEEE 18th International Conference on Software Engineering*, pages 258–267. IEEE Comput. Soc. Press. (Cited on page 11.)

[Dijkstra, 1972] Dijkstra, E. W. (1972). *Structured programming*. Academic Press Ltd., London, UK, UK. (Cited on page 3.)

[Doliner, 2006] Doliner, M. (2006). Cobertura. Available online at `http://cobertura.sourceforge.net`, last accessed: 02/2011. (Cited on page 81.)

[Dutertre and de Moura, 2006] Dutertre, B. and de Moura, L. (2006). The Yices SMT solver. Technical report, SRI International. (Cited on pages 24, 105, 109, 110, 119, 123, and 126.)

[Ernst et al., 1999] Ernst, M. D., Cockrell, J., Griswold, W. G., and Notkin, D. (1999). Dynamically Discovering Likely Program Invariants to Support Program Evolution. In *Proceedings of the International Conference on Software Engineering*, pages 213–224. (Cited on page 34.)

[Fikes and Nilsson, 1971] Fikes, R. E. and Nilsson, N. J. (1971). STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3-4):189–208. (Cited on page 85.)

[Flanagan et al., 2002] Flanagan, C., Leino, K., Lillibridge, M., Nelson, G., Saxe, J., and Stata, R. (2002). Extended static checking for Java. In *ACM Sigplan Notices*, volume 37, pages 234–245, New York, NY, USA. ACM Press. (Cited on page 49.)

[Freese, 2002] Freese, T. (2002). EasyMock: Dynamic Mock Objects for JUnit. *XP2002*. (Cited on pages 42, 71, and 81.)

[Galler et al., 2010a] Galler, S., Maller, A., and Wotawa, F. (2010a). Automatically Extracting Mock Object Behavior from Design-by-Contract Specification for Test Data Generation. In *Proceedings of the 5th Workshop on Automation of Software Test*, AST'10. (Cited on pages 8 and 71.)

[Galler and Aichernig, 2010] Galler, S. J. and Aichernig, B. K. (2010). State-of-the-art White- and Gray-Box Test Generation Tools. submitted to STTT Journal. (Cited on page 24.)

[Galler et al., 2008] Galler, S. J., Peischl, B., and Wotawa, F. (2008). Challenging Automatic Test Case Generation Tools with Real World Applications. In *Proceedings of the Software Engineering and Applications 2008*. (Cited on page 7.)

[Galler et al., 2011] Galler, S. J., Quaritsch, T., Weiglhofer, M., and Wotawa, F. (2011). The IntiSa approach: Test Input Data Generation for Non-Primitive Data Types by means of SMT solver based Bounded Model Checking. submitted to QSIC 2011. (Cited on pages 8, 61, and 117.)

[Galler et al., 2010b] Galler, S. J., Weiglhofer, M., and Wotawa, F. (2010b). Synthesize it: from Design by Contract to Meaningful Test Input Data. In *Proceedings of the 8th IEEE International Conference on Software Engineering and Formal Methods*. (Cited on pages 8, 61, 103, and 126.)

[Galler et al., 2010c] Galler, S. J., Zehentner, C., and Wotawa, F. (2010c). AIana: An AI Planning System for Test Data Generation. In *Proceedings of the 1st Workshop on Testing Object-Oriented Systems*, pages 30–37, Maribor, Slovenja. (Cited on pages 8, 61, and 85.)

[Gamma et al., 2002] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (2002). *Design Patterns*. Addison-Wesley Reading, MA. (Cited on page 53.)

[Ghallab et al., 1998] Ghallab, M., Nationale, E., Aeronautiques, C., Isi, C. K., Penberthy, S., Smith, D. E., Sun, Y., and Weld, D. (1998). PDDL - The Planning Domain Definition Language. online, last accessed 2010. (Cited on pages 87, 88, 89, and 90.)

[Girgis, 1993] Girgis, M. R. (1993). Corrigendum for 'Constraint-Based Automatic Test Data Generation' by R.A. DeMillo and A.J. Offutt. *IEEE Trans. Softw. Eng.*, 19:640.1–. (Cited on page 58.)

[Godefroid et al., 2005] Godefroid, P., Klarlund, N., and Sen, K. (2005). DART: directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, volume 40, pages 213–223, New York, NY, USA. ACM Press. (Cited on pages 25, 48, and 51.)

[Google, 2008] Google (2008). Friends you can depend on. Available online at `http://googletesting.blogspot.com/2008/06/tott-friends-you-can-depend-on.html`, last accessed: 02/2011. (Cited on pages 29 and 52.)

[Google, 2010] Google (2010). CodePro AnalytiX. Available online at `http://code.google.com/javadevtools/codepro/doc/index.html`, last accessed: 11/2010. (Cited on pages 25, 41, 42, and 43.)

[Gunnerson, 2000] Gunnerson, E. (2000). *A Programmer's Introduction to C#*. Apress Berkeley, CA. (Cited on page 16.)

[Harman et al., 2002] Harman, M., Hierons, R., Fox, C., Danicic, S., and Howroyd, J. (2002). Pre/-post conditioned slicing. In *Software Maintenance, 2001. Proceedings. IEEE International Conference on*, pages 138–147. IEEE. (Cited on page 136.)

[Hierons et al., 2009] Hierons, R., Bogdanov, K., Bowen, J., Cleaveland, R., Derrick, J., Dick, J., Gheorghe, M., Harman, M., Kapoor, K., Krause, P., et al. (2009). Using formal specifications to support testing. *ACM Computing Surveys (CSUR)*, 41(2):1–76. (Cited on page 5.)

[Hoare, 1969] Hoare, C. A. R. (1969). An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580. (Cited on pages 9 and 88.)

[Hoare and He, 1998] Hoare, C. A. R. and He, J. (1998). *Unifying Theories of Programming*. Prentice Hall, Upper Saddle River, NJ, USA. (Cited on page 64.)

[Hsu et al., 2006] Hsu, C. W., Wah, B. W., Huang, R., and Chen, Y. X. (2006). New features in sgplan for handling soft constraints and goal preferences in pddl3.0. In *Proceedings of the 5th International Planning Competition*. International Conf. on Automated Planning and Scheduling. (Cited on page 101.)

[Huima, 2007] Huima, A. (2007). *Implementing Conformiq Qtronic*, volume 4581 of *Lecture Notes in Computer Science*, pages 1–12–12. Springer Berlin Heidelberg, Berlin, Heidelberg. (Cited on page 50.)

[Jaffuel and Legeard, 2006] Jaffuel, E. and Legeard, B. (2006). *LEIRIOS Test Generator: Automated Test Generation from B Models*, volume 4355 of *Lecture Notes in Computer Science*, pages 277–280. Springer Berlin Heidelberg, Berlin, Heidelberg. (Cited on page 50.)

[Johnson and Kernighan, 1973] Johnson, S. and Kernighan, B. (1973). The Programming Language B. Technical report, Bell Labs. (Cited on page 9.)

[Jones, 1986] Jones, C. B. (1986). *Systematic software development using VDM*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK. (Cited on page 9.)

[Kuliamin et al., 2003] Kuliamin, V. V., Petrenko, A. K., Kossatchev, A. S., and Bourdonov, I. B. (2003). UniTesK: Model Based Testing in Industrial Practice. In *Proceedings of the 1st European Conference on Model-Driven Software Engineering*, pages 55–63, B. Communisticheskaya, 25, 109004, Moscow, Russia. (Cited on page 25.)

[Lardeux et al., 2008] Lardeux, F., Monfroy, E., and Saubion, F. (2008). *Interleaved Alldifferent Constraints: CSP vs. SAT Approaches*, volume 5253 of *Lecture Notes in Computer Science*, pages 380–384. Springer Berlin Heidelberg, Berlin, Heidelberg. (Cited on page 23.)

[Larsen et al., 2010] Larsen, P. G., Battle, N., Ferreira, M., Fitzgerald, J., Lausdahl, K., and Verhoef, M. (2010). The overture initiative integrating tools for VDM. *ACM SIGSOFT Software Engineering Notes*, 35(1):1–6. (Cited on page 25.)

[Leavens and Cheon, 2003] Leavens, G. and Cheon, Y. (2003). Design by Contract with JML. (Cited on pages 12 and 13.)

[Leavens et al., 2002] Leavens, G., Poll, E., Clifton, C., Cheon, Y., and Ruby, C. (2002). JML reference manual. (Cited on pages 20, 21, and 101.)

[Leavens et al., 1998] Leavens, G. T., Baker, A. L., and Ruby, C. (1998). Preliminary Design of JML. Technical report, Department of Computer Science, Iowa State University. (Cited on page 20.)

[Leavens et al., 1999] Leavens, G. T., Baker, A. L., and Ruby, C. (1999). JML: A Notation for Detailed Design. *Behavioral Specifications of Businesses and Systems*, pages 175–188. (Cited on page 10.)

[Legeard et al., 2002] Legeard, B., Peureux, F., and Utting, M. (2002). Automated Boundary Testing from Z and B. In Eriksson, L.-H. and Lindsay, P., editors, *Proceedings of the International Symposium of Formal Methods*, volume 2391 of *Lecture Notes in Computer Science*, pages 221–236. Springer Berlin / Heidelberg. (Cited on page 50.)

[Leitner and Bloem, 2005] Leitner, A. and Bloem, R. (2005). Automatic Testing through Planning. Technical report, Graz University of Technology. (Cited on page 85.)

[Liskov and Wing, 1994] Liskov, B. H. and Wing, J. M. (1994). A Behavioral Notion of Subtyping. *ACM Transactions on Programming Languages and Systems*, 16:1811–1841. (Cited on page 13.)

[Mackinnon et al., 2001] Mackinnon, T., Freeman, S., and Craig, P. (2001). *Endo-testing: unit testing with mock objects*, pages 287–301. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA. (Cited on page 52.)

[Maller et al., 2010] Maller, A., Galler, S. J., and Wotawa, F. (2010). A Survey on Design-by-Contract support of state-of-the art programming languages. Technical report, Institute for Software Technology, Graz University of Technology. (Cited on page 14.)

[Mccarthy and Hayes, 1969] Mccarthy, J. and Hayes, P. J. (1969). Some Philosophical Problems from the Standpoint of Artificial Intelligence. In *Machine Intelligence*, volume 4, pages 463–502. (Cited on page 86.)

[Metsker and Wake, 2006] Metsker, S. J. and Wake, W. C. (2006). *Design Patterns in Java*. Addison-Wesley Professional. (Cited on page 53.)

[Meyer, 1992a] Meyer, B. (1992a). Applying "Design by Contract". *Computer*, 25(10):40–51. (Cited on page 10.)

[Meyer, 1992b] Meyer, B. (1992b). *Eiffel The language*. Prentice Hall PTR. (Cited on page 15.)

[Meyer, 1997] Meyer, B. (1997). *Object Oriented Software Construction*. Prentice Hall PTR, first edition. (Cited on pages 9, 14, 15, and 16.)

[Meyer et al., 2007] Meyer, B., Ciupa, I., Leitner, A., and Liu, L. (2007). Automatic Testing of Object-Oriented Software. In *SOFSEM 2007: Theory and Practice of Computer Science*, pages 114–129. (Cited on pages 25, 38, and 39.)

[Meyers, 2005] Meyers, S. (2005). *Effecitve C++: 55 Specific Ways to Improve Your Programs and Design*. Addison-Wesley Professional, 3rd editio edition. (Cited on page 31.)

[Microsoft-Research, 2010] Microsoft-Research (2010). Advanced Concepts: Parameterized Unit Testing with Microsoft Pex. Technical report, Microsoft Research, Redmond. (Cited on page 48.)

[Müller, 2011] Müller, E. (2011). Wenn die Dinge sprechen lernen - Internet of Things. *Manager Magazine*, 2:84–88. (Cited on page 131.)

[Myers et al., 2004] Myers, G. J., Sandler, C., Badgett, T., and Thomas, T. M. (2004). *The Art of Software Testing*. Wiley, 2nd edition. (Cited on pages 27 and 28.)

[Nieuwenhuis et al., 2007] Nieuwenhuis, R., Oliveras, A., Rodrígue-Carbonell, E., and Rubio, A. (2007). Challenges in Satisfiability Modulo Theories. In Baader, F., editor, *Term Rewriting and Applications*, volume 4533 of *Lecture Notes in Computer Science*, chapter 2, pages 2–18–18. Springer Berlin / Heidelberg, Berlin, Heidelberg. (Cited on page 23.)

[Pacheco and Ernst, 2007] Pacheco, C. and Ernst, M. D. (2007). Randoop: Feedback-Directed Random Testing for Java. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, page 815. (Cited on pages 25 and 45.)

[Pacheco et al., 2008] Pacheco, C., Lahiri, S. K., and Ball, T. (2008). Finding Errors in .NET with Feedback-Directed Random Testing. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 87–96. ACM. (Cited on page 45.)

[Pacheco et al., 2007] Pacheco, C., Lahiri, S. K., Ernst, M. D., and Ball, T. (2007). Feedback-directed random test generation. In *Proceedings of the 29th International Conference on Software Engineering*, Minneapolis, MN, USA. (Cited on pages 45 and 46.)

[Paraosft, 2010] Paraosft (2010). Parasoft C++test User's Guide. (Cited on pages 25, 31, 32, and 33.)

[Parasoft, 2010a] Parasoft (2010a). Jtest 8.0 User Guide. (Cited on page 10.)

[Parasoft, 2010b] Parasoft (2010b). Parasoft. (Cited on page 28.)

[Parasoft, 2010c] Parasoft (2010c). Using Design by Contract to Automate Java Software and Component Testing. (Cited on page 25.)

[Penix et al., 2005] Penix, J., Visser, W., Park, S., Pasareanu, C., Engstrom, E., Larson, A., and Weininger, N. (2005). Verifying Time Partitioning in the DEOS Scheduling Kernel. *Formal Methods in System Design*, 26(2):103–135. (Cited on page 51.)

[Quaritsch, 2010] Quaritsch, T. (2010). jConTest: Describing Test Data Dependencies and Design by Contract Specifications for Automatic Test Data Generation. Technical report, Graz University of Technology. (Cited on page 126.)

[Quaritsch, 2011] Quaritsch, T. (2011). Test Data Generation using Static Call Sequence Analysis and Design by Contract Specifications. Master's thesis, Graz University of Technology. (Cited on pages 135 and 136.)

[Rieken, 2007] Rieken, J. (2007). Design By Contract for Java - Revised. Master's thesis, Department für Informatik, Universität Oldenburg. (Cited on pages 10, 13, 22, 62, and 100.)

[Russell and Norvig, 2002] Russell, S. J. and Norvig, P. (2002). *Artificial Intelligence: A Modern Approach*. Prentice Hall, second edition. (Cited on page 86.)

[Saff and Ernst, 2004] Saff, D. and Ernst, M. D. (2004). Mock object creation for test factoring. In *PASTE '04: Proceedings of the 5th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 49–51, New York, NY, USA. ACM. (Cited on page 71.)

[Sen and Agha, 2006] Sen, K. and Agha, G. (2006). CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools. In *Proceedings of the 18th International Conference on Computer Aided Verification*, pages 419–423, Seattle, Washington, USA. Springer. (Cited on pages 25 and 51.)

[Sen et al., 2005] Sen, K., Marinov, D., and Agha, G. (2005). CUTE: a concolic unit testing engine for C. In *Proceedings of the 10th European Software Engineering Conference*, volume 30, page 263, Lisbon, Portugal. ACM. (Cited on page 48.)

[SMTCOMP, 2010] SMTCOMP (2010). Call for Entrants. (Cited on page 27.)

[Spivey, 1989] Spivey, J. M. (1989). *The Z notation: a reference manual*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA. (Cited on page 9.)

[Tarski, 1941] Tarski, A. (1941). On the Calculus of Relations. *Journal of Symbolic Logic*, 6(3):73–89. (Cited on page 64.)

[Tillmann and de Halleux, 2008] Tillmann, N. and de Halleux, J. (2008). Pex - White Box Test Generation for .NET. In Beckert, B. and Hähnle, R., editors, *Proceedings of the 2nd International Conference on Tests and Proofs*, volume 4966 of *Lecture Notes in Computer Science*, pages 134–153, Berlin, Heidelberg. Springer Berlin Heidelberg. (Cited on pages 10, 25, 48, and 58.)

[Tillmann and Schulte, 2005] Tillmann, N. and Schulte, W. (2005). Parameterized unit tests. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 253–262, New York, NY, USA. ACM Press. (Cited on page 48.)

[Tillmann and Schulte, 2006] Tillmann, N. and Schulte, W. (2006). Mock-object generation with behavior. In *Proceedings of the 21st IEEE International Conference on Automated Software Engineering*, pages 365–368, Washington, DC, USA. IEEE Computer Society. (Cited on page 71.)

[Veanes et al., 2010] Veanes, M., de Halleux, J., and Tillmann, N. (2010). Rex: Symbolic regular expression explorer. In *Proceedings of the 3rd International Conference on Software Testing, Verification and Validation*, pages 498–507. IEEE. (Cited on pages 48, 49, and 113.)

[Visser et al., 2003] Visser, W., Havelund, K., Brat, G., Park, S., and Lerda, F. (2003). Model Checking Programs. *Automated Software Engg.*, 10(2):203–232. (Cited on page 51.)

[Visser et al., 2004] Visser, W., Pasareanu, C. S., and Khurshid, S. (2004). Test input generation with java PathFinder. In *Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, volume 29, pages 97–107, New York, NY, USA. ACM Press. (Cited on pages 25 and 51.)

[von Mayrhauser et al., 2000] von Mayrhauser, A., Scheetz, M., Dahlman, E., and Howe, A. E. (2000). Planner Based Error Recovery Testing. In *Proceedings of the 11th International Symposium on Software Reliability Engineering*, Washington, DC, USA. IEEE Computer Society. (Cited on page 85.)

[Wampler, 2006] Wampler, D. (2006). The Challenges of Writing Reusable and Portable Aspects in AspectJ: Lessons from Contract4J. In *Proceedings of International Conference on Aspect Oriented Software Development (AOSD 2006)*. (Cited on page 10.)

[Wei et al., 2010] Wei, Y., Gebhardt, S., Oriol, M., and Meyer, B. (2010). Satisfying Test Preconditions through Guided Object Selection. In *Proceedings of the 3rd International Conference on Software Testing, Verification and Validation*, pages 1–10, Paris, France. (Cited on page 39.)

[Weld, 1999] Weld, D. S. (1999). Recent Advances in AI Planning. *AI Magazine*, 20(2):93–123. (Cited on page 90.)

[Weyuker, 1982] Weyuker, E. J. (1982). On Testing Non-Testable Programs. *Computer Journal*, pages 465–470. (Cited on page 4.)

[Zehentner, 2010] Zehentner, C. (2010). Planning4ObjectCreation: An AI-Planning System for Test Data Generation. Master's thesis, Graz University of Technology. (Cited on pages 85, 86, and 88.)

Appendix A

List of Publications

A.1. Published

Galler, S., Peischl, B., and Wotawa, F. (2008). Challenging Automatic Test Case Generation Tools with Real World Applications. In *Software Engineering and Applications 2008*.

I developed the experiment setting and conducted the evaluation. Bernhard Peischl and Franz Wotawa helped in writing and proof-reading the paper.

Galler, S. J., Maller, A., and Wotawa, F. (2010a). Automatically Extracting Mock Object Behavior from Design-by-Contract Specification for Test Data Generation. In *AST 2010*.

The idea of replacing parameters with mock objects and configuring them according to their Design by Contract™ specification came from me. Most of the final implementation was conducted by myself as well. Andreas Maller provided a first prototype implementation. Franz Wotawa helped in writing the publication.

Galler, S. J., Zehentner, C., and Wotawa, F. (2010c). AIana: An AI Planning System for Test Data Generation. In *1st Workshop on Testing Object-Oriented Software Systems*, pages 30–37, Maribor, Slovenja.

The idea of mapping Design by Contract™ specifications to PDDL actions and using AI planner came from me. Christoph Zehentner was a master student who did most of the implementation. The paper was written by myself with input and proof-reading of my supervisor Franz Wotawa.

Galler, S. J., Weiglhofer, M., and Wotawa, F. (2010b). Synthesize it: from Design by Contract to Meaningful Test Input Data. In *The 8th IEEE International Conference on Software Engineering and Formal Methods*.

The idea of synthesizing mock objects from Design by Contract™ specifications and the prototype implementation came from me. Martin Weiglhofer helped in getting the formalizations correct. Franz Wotawa proof-read the publication.

A.2. Not yet published

Galler, S. J., Quaritsch, T., Weiglhofer, M., and Wotawa, F. (2011). The IntiSa approach: Test Input Data Generation for Non-Primitive Data Types by means of SMT solver based Bounded Model Checking. submitted to QSIC 2011.

The initial idea of formalizing an SMT problem to calculate the reachable initial state of a parameter object came from me. Input and discussions with Martin Weiglhofer lead to the final solution of using bounded-model checking on Kripke structures. My supervisor Franz Wotawa helped in proof-reading the paper.

Galler, S. J. and Aichernig, B. K. (2010). State-of-the-art White- and Gray-Box Test Generation Tools. submitted to STTT Journal.

I evaluated and wrote the survey under the supervision of Bernhard Aichernig.