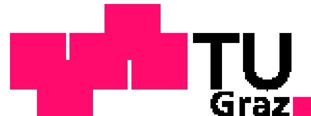
Master's Thesis

Planning4ObjectCreation An AI-Planning System for Test Data Generation

Christoph Zehentner

Institute for Softwaretechnology
Graz University of Technology



Supervisor: Univ.-Prof. Dipl.-Ing. Dr. techn. Franz Wotawa
Dipl.-Ing. Stefan J. Galler

Graz, April 2010

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, ____________________ ______________________________

(date) (signature)

Abstract

As object oriented software tends to be very complex, the automatic generation and execution of tests is very important. The focus of this thesis thus comprises the automatic generation of test data for jUnit tests. Additionally, Design by Contract™ annotations consisting of pre- and postconditions might be available. These specifications of the method under test state whether the method can be executed or a potential fault is detected respectively. Whereas a postcondition can simply be checked for compliance, fulfilling the precondition is a more complex task, as all paramteres must comply with the precondition as well.

If such a parameter is a complex type (a class or an interface), an instance has to be generated and its state has to be modified to comply with the precondition of the method under test. The jCAMEL tool, which focuses on the generation of these tests automatically currently uses a random strategy. An object is instantiated and methods are called at a random basis until the target state is reached. As such method invocation sequences tend to be complex, the random strategy performs poorly.

The proposed approach tackles this problem of low-perfomance by transforming the available Design by Contract™ specifications of the requested class into a planning domain. The precondition of the method under test is translated to a planning problem. A planner then generates a plan, a sequence of actions that transform an initial state the newly instantiated object is in, into the requested goal state. This way, the compliance with the necessary precondition can be ensured. The resulting plan can be retranslated into Java code and then be used as a parameter for the method under test that fulfills the precondition.

Our approach enables jCAMEL to generate specialized tests for methods that use parameters with complex requirements fast and goal oriented. As a result, the plan-based approach improves test coverage substantially over the random strategy.

Zusammenfassung

Objektorientierte Software ist in den meisten Fällen sehr komplex. Daher wird das Automatisieren von Testerzeugung und Beurteilung immer wichtiger. Diese Arbeit beschäftigt sich mit der Testdatenerzeugung für jUnit Tests unter Verwendung von Design by Contract™ Annotationen. Ein jUnit Test besteht aus einem Aufruf einer Methode einer Klasse und dem Vergleich des erwarteten und eingetroffenen Ereignisses. Ist diese Software mit Design by Contract™ Annotierungen spezifiziert, kann aus der Überprüfung der Nachbedingung der Methodenspezifikation entschieden werden, ob sich die Methode korrekt verhalten hat, oder ob die Spezifikation verletzt wurde, was auf eine fehlerhafte Implementierung schließen lässt.

Um eine Methode aber aufrufen zu können, ohne die Vorbedingung der Design by Contract™ Spezifikation zu verletzten, müssen alle Parameter so erstellt werden, dass sie der Vorbedingung entsprechen. Handelt es sich um einen nicht-primitiven Datentyp, eine Klasse oder ein Interface, so muss eine Instanz erzeugt werden, und darauf Methoden aufgerufen werden, bis diese Instanz in einem Zustand ist, die der Spezifikation der getesteten Methode genügt. Solche Sequenzen können sehr komplex werden, sodass der bereits bestehende Ansatz, das randomisierte Aufrufen von Methoden, schnell an seine Grenzen stößt.

Der in dieser Arbeit entwickelte Ansatz stellt eine Erweiterung des jCAMEL Testerzeugungswerkzeugs dar, welches basierend auf Design by Contract™ Spezifikation eben solche jUnit Tests erstellt. Hierfür wird die Spezifikation der zu erzeugenden Klasse in eine Planungsdomäne überführt. Die Vorbedingung der zu testenden Methode wird in ein Planungsziel konvertiert. Mit dieser Information wird nun ein bestehender Planer aufgerufen. Der entstehende Plan, der einer Aufrufsequenz von Methoden auf ein Objekt entspricht, wird nun in Java Code zurücktransformiert. Dieses Objekt ist nun in einem Zustand in dem es der Vorbedingung der zu testenden Methode entspricht und kann als Parameter verwendet werden.

Das dadurch erzeugte Objekt ermöglicht es, im Gegensatz zur den meisten Fällen der randomisierten Erzeugung von Objekten, auch Methoden mit nicht-primitiven Parametern zu testen, und führt daher zu stark erhöhter Testabdeckung.

Contents

I Introduction 9

1 Motivation 10

2 Preliminaries 13
2.1 Basics . . . 13
2.1.1 The Stack . . . 13
2.1.2 Definitions of a Class . . . 15
2.2 Design by Contract™ . . . 17
2.3 jCAMEL . . . 21
2.3.1 Denotable . . . 22
2.4 Planning . . . 24
2.4.1 Planning Languages . . . 26
2.4.2 PDDL4J . . . 28

II Approach 29

3 Approach 30
3.1 Generation of the Planning Domain . . . 31
3.2 Generation of the Planning Goal . . . 37
3.3 Running the Planner . . . 40
3.4 Generation of Java Code based on a Plan . . . 40
3.5 Details . . . 40
3.5.1 Handling of Constructors . . . 41
3.5.2 Translation of Specifications . . . 43
3.5.3 Optimistic Test Data Generation . . . 51
3.5.4 Pessimistic Test Data Generation . . . 54
3.6 Limitations . . . 56

4 Related Work and Alternative Approaches 57
4.1 Related Work . . . 57
4.1.1 System Testing with an AI-Planner . . . 58
4.1.2 The Planning Extension for TestStudio . . . 58

4.1.3 MEA-Graphplan 60
4.1.4 Dingels et al. 61
4.2 Comparison to related work 61
4.3 Alternative Approaches 64
4.3.1 KORAT 64
4.3.2 Random Test Data Generation 64
4.3.3 Test Data Generation using Evolutionary Algorithms 66

III Implementation 68

5 Implementation 69
5.1 Infix to Prefix 69
5.2 Design 71
5.2.1 Configuration 77
5.3 Implementation Details 78
5.3.1 Generation of the Planning Domain 78
5.3.2 Generation of the Planning Goal 80
5.3.3 Execution of the Planner 80
5.3.4 Generation of Java Code from a Plan 81
5.4 Limitations and Assumptions 84

IV Results 85

6 Empirical Evaluation 86
6.1 StackCalc 87
6.2 BillingSoftware 89

7 Conclusion 92
7.1 Further Research 93

List of Figures

1.1 Overview of the systems generation strategy 12

2.1 Denotables. 23

4.1 The planning flowchart of Leitners extension to TestStudio [Lei04] 59
4.2 The automated planning system (APS) as proposed by Gupta et al. [gup04]. 61

5.1 Tree representation of $(x + y) - ((2 * a) + b)$ 70
5.2 Overview of the system 72
5.3 Design of the application. 73
5.4 Overview of the generation process. 74
5.5 Generation of PDDL operators for one class 75
5.6 The Denotable generated from the plan given in Listing 5.4. 83

List of Tables

3.1 Default values for state initialization. 38
3.2 Incorrect values during evaluation of the PDDL action effect. 45
3.3 Correct values during evaluation of the PDDL action effect. 46
3.4 Comparison of generation strategies. 55

4.1 Comparison of our approach to Leitner, Howe and Dingels 63

5.1 Unsupported characters in PDDL and their translation from Java. 80

6.1 Amount of Modern Jass annotations in the StackCalc implementation. . 87
6.2 Number of generated tests within the *StackCalc* case study. 88
6.3 Line and branch coverage of the generated tests of the *StackCalc* case study. 88
6.4 Number of generated tests for operators only of the *StackCalc* case study. 89
6.5 Line and branch coverage of the generated tests for operators only of the *StackCalc* case study. 89
6.6 Number of generated tests of the *BillingSoftware* case study 91
6.7 Line and branch coverage of the generated tests of the *BillingSoftware* case study 91
6.8 Average test generation time in seconds over all 100 experiment iterations. 91

Part I

Introduction

1 Motivation

Modern software projects are complex. Testing this software is a time and resource consuming task. Almost 50% of development costs are spent for testing activities [Tas02]. One important factor when testing software is using the right data to find erros. Generating this data by hand is time-consuming and error-prone. Moreover, handwritten tests and test data often lacks of satisfactory test coverage.

The above mentioned problems can be handled better, if the data and the tests are generated automatically. One such approach of generating test data for Java programs is the jCAMEL [GPW08] tool developed at the Institute for Softwaretechnology, Graz University of Technology. It aims for generating jUnit-Tests for Design-by-Contract annotated Java classes.

One key feature is generating tests for given methods of a class. As methods may contain parameters, these paramters need to be generated to call the method. As the method is annotated with Design-by-Contract annotations, the parameters can be restricted. An integer parameter may need to be within a given range, or an object must be in a given state. Several value generators for different data types already exists. If the parameter of the method under test is an object-reference, the generation of the object becomes a complex task. The current idea is to generate an object by calling its constructor, and then randomly call methods until a before specified timeout is reached. If the state is reached, the object can be used as parameter for the method under test. This solution has some drawbacks. It is not guaranteed that the generation of an object in a desired target state succeeds in the given time.

Example 1: Consider an example method that needs to be tested, that takes a reference to a java.util.Stack as its single parameter as shown in Listing 1.1. The Design by Contract™ annotation forces this stack to have at least five elements.

```
@Pre("stack.size() >= 5")
public void useStack(Stack stack) { ... }
```

Listing 1.1: The sample method useStack takes one input parameter. The above Modern Jass annotation forces the parameter to contain at least five elements.

The Stack class contains 60 public methods, where 6 methods can be used to add content to the stack, and 10 will remove at least one element or clear the whole stack. Therefore we have a probability of $p(addElement) = 0.1$ to select a method that will lead into the desired direction. Using the above example, the generation process needs to call at least five push-operations without calling any remove method. This leads to the following probability to generate a valid object:

$$
\begin{aligned}
p(addElement) &= 0.1 \\
p(stackSize = 5) &= p(addElement)^5 \\
p(stackSize = 5) &= 0.00001
\end{aligned}
$$

Having a probability of **0.001%** is not acceptable. Using more steps than the minimum of five, the probability of generating a valid object may increase, but is still not satisfactory. Furthermore, the probability of accidently clearing the stacks content increases too. □

Therefore, a new generation strategy should be developed. As the jCAMEL tool needs annotated java classes to generate tests, this information can be used to implement the new strategy. This should be done using as many public available and well documented software packages as possible. The idea of using a planning system that is based upon the Design by Contract™ annotations to generate a valid object as parameter was developed. The system should translate the DbC-annotations into a format already implemented planners understand. Then the planner is invoked to generate a plan. Afterwards, the generated plan has to be converted to the internal data structure of jCAMEL, to be able to generate a jUnit test.

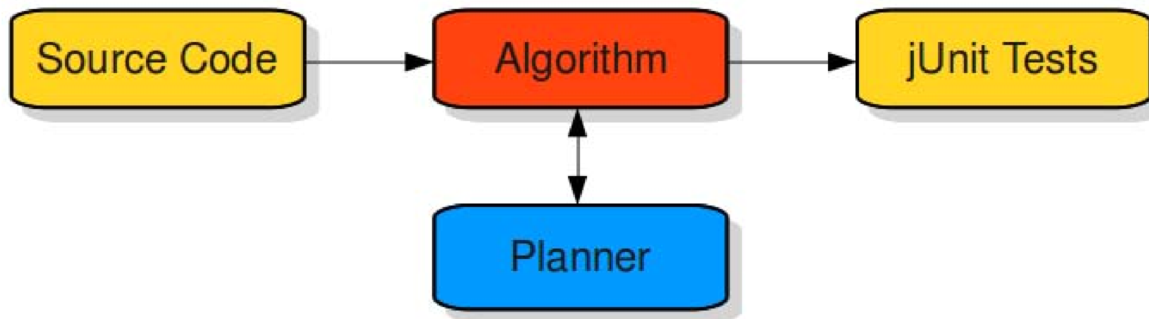


Figure 1.1: Overview of the systems generation strategy. The source code of the *software under test* is taken and transformed to a representation an already existing planner is able to understand. This planner is then used to generate a valid sequence of actions, that transform the object into the desired goal state. After this step is done, the algorithm transforms the planners output into a sequence of Java-code that can be used within a jUnit test case.

This thesis is structured as follows. Chapter 2 will give an overview on the used technologies. In Chapter 3 we will explain the approach we have chosen to solve the problems pointed out in this motivation. Furthermore, we will review related work and alternative approaches in Chapter 4.

Chapter 5 shows the design and functionality of the implementation. In Chapter 6 we show how the system is able to generate test data using some two studies. In Chapter 7 we will conclude about the work being done and give an outlook on what can be improved or extended.

2 Preliminaries

This section will introduce all technologies necessary to understand the presented approach. First of all we will introduce a running example used to point out our ideas and give formal definitions on concepts used within this thesis in Section 2.1. The ideas of Design by Contract™ will be discussed in Section 2.2. After that, jCAMEL, the test data generation tool developed at Graz University of Technology by Galler et al. [GPW08] wich will be extended within this work, is described in Section 2.3. Furthermore the ideas of AI-planning will be reviewed in Section 2.4. Within that section we will discuss what planning is and introduce some major planning description languages as one of them, PDDL [GIP+98], will be used within this thesis.

2.1 Basics

In this Section, we will give basic definitions we will use to point out our ideas throughout this work. First of all, we will introduce the example classes used to describe our ideas. Then we will give formal definitions on concepts used.

2.1.1 The Stack

We will use a stripped down version of the java.util.Stack class to demonstrate the basic ideas of our approach. The methods have been annotated using Modern Jass [Rie07] Design by Contract™ specifications.

```
1  @Invariant("size_ >= 0")
2  @Model(name="mSize", type="int")
3  public class Stack {
4    private int size_;
5
6    @Post("mSize == 0)
7    public Stack() { ... }
8
9    @Post("size() == @Old(size(), int) + 1")
10   public void push(int element) { ... }
11
12   @Pre("size() > 0")
13   @Post("size() == @Old(size(), int) - 1")
14   public void pop() { ... }
15
16   @Pre("size() > 0")
17   @Post("size() == @Old(size(), int)")
18   public int peek() { ... }
19
20   @Pure
21   @Also(
22     @SpecCase(pre="size() > 0", post = "@Return == false"),
23     @SpecCase(pre="size() == 0", post = "@Return == true")
24   )
25   public boolean isEmpty() { ... }
26
27   @Pure
28   @Post("@Return == mSize")
29   public int size() { ... }
30 }
```

Listing 2.1: The annotated *Stack* class used as running example within this thesis. The constructor creates an object of class *Stack* that is empty. *push*(*int*) adds one element to the stack, whereas *pop*() removes the top element. *peek*() returns the top element and does not changes the stacks content. The method *size*() is used to return the stacks current size, and is marked as pure, which means that it does not change anything within the object, and can be used within other method contracts. *isEmpty*() returns whether the *Stack* has elements or not. Its specification is split into two *SpecCase* annotations. The invariant of the private field *size_* ensures that the size of the stack can never become negative. There is one model variable *mSize* to demonstrate the usage of model variables and how our approach deals with them.

2.1.2 Definitions of a Class

Within this Section, we will give formal definitions of concepts used throughout this thesis. We will start by defining an object oriented class in a formal way. Upon that, we will build the concepts of State Variables, the public visible state of a class and State Manipulators, which will change this State Variables.

Definition 1: Class
Galler et al. [GWW10] define a class as a triple $C = \langle c, m, f \rangle$. Where c is the set of public available Constructors. f is the union of public, protected and private fields: $f = f^{pub} \cup f^{prot} \cup f^{priv}$, and m the set of all methods.

$$m = \bigcup_{vis \in \left\{ \begin{matrix} pub, \\ prot, \\ priv \end{matrix} \right\}} \bigcup_{ret \in \left\{ \begin{matrix} void, \\ nvoid \end{matrix} \right\}} \bigcup_{abs \in \left\{ \begin{matrix} abs, \\ con \end{matrix} \right\}} {}^{abs}m^{vis}_{ret}$$

Here vis denotes the visibility of the methods, which is either public, protected or private, ret states whether the method returns a value ($nvoid$) or not ($void$) and abs denotes whether the method is concrete (con) or abstract (abs). □

Example 2: Using the Definition of a class as triple $C = \langle c, m, f \rangle$ the set of constructors C of the Stack example given in Listing 2.1, is the only available public constructor: $c = \{Stack()\}$. There are no public or protected fields in this examples. Therefor $f^{pub} = \emptyset$ and $f^{prot} = \emptyset$. There is one private field: $f^{priv} = \{size_\}$. Thus we write f as $f = f^{pub} \cup f^{prot} \cup f^{priv} = \emptyset \cup \emptyset \cup \{size_\} = \{size_\}$. There are two concrete methods that do not return a value, namely push and pop. Thus ${}^{con}m^{pub}_{void} = \{push, pop\}$. The three other methods do return a value and are also concrete methods: ${}^{con}m^{pub}_{nvoid} = \{peek(), size(), isEmpty()\}$. As there are no more methods in this example, all other sets of methods are emtpy. We can now write the set of methods as:

$$m = {}^{con}m^{pub}_{void} \cup {}^{con}m^{pub}_{nvoid} \cup {}^{abs}m^{pub}_{void} \cup \ldots \cup {}^{abs}m^{priv}_{nvoid}$$
$$m = \{push(int), pop()\} \cup \{peek(), size(), isEmpty()\} \cup \emptyset \ldots \cup \emptyset$$
$$m = \{push(int), pop(), peek(), size(), isEmpty()\}$$

So we can write the class of *Stack* as:

$$C(Stack) = \langle \{Stack()\}, \{push(int), pop(), peek(), size(), isEmpty()\}, \{size_\} \rangle$$

□

Definition 2: External State.
For a class $C = \langle c, f^{pub} \cup f^{prot} \cup f^{priv}, {}^{con}m^{pub}_{void} \cup {}^{con}m^{pub}_{nvoid} \cup {}^{abs}m^{pub}_{void} \cup \ldots \cup {}^{abs}m^{priv}_{nvoid} \rangle$ the external state is the set $ES(C)$ of all public accessible fields and non-void methods:

$$ES(C) = {}^{con}m^{pub}_{nvoid} \cup f^{pub}$$

□

Example 3: Using the Stack from Listing 2.1 for Definition 2 we can write its external state as:

$$\begin{aligned} {}^{con}m^{pub}_{nvoid} &= \{size(), peek(), isEmpty()\} \\ f^{pub} &= \emptyset \\ ES(Srack) &= \{size(), peek(), isEmpty()\} \end{aligned}$$

□

Definition 3: State Manipulators.
We define the set of operations $SM(C)$ of a class $C = \langle c, f^{pub} \cup f^{prot} \cup f^{priv}, {}^{con}m^{pub}_{void} \cup {}^{con}m^{pub}_{nvoid} \cup {}^{abs}m^{pub}_{void} \cup \ldots \cup {}^{abs}m^{priv}_{nvoid} \rangle$ that might manipulate the object's external state $ES(C)$ as the union of public available constructors and concrete methods:

$$SM(C) = c \cup {}^{con}m^{pub}_{nvoid} \cup {}^{con}m^{pub}_{void}$$

□

Example 4: The Stacks State Manipulators
Having the Definition of Potential State Manipulators we write the set of State Manipulators of the Stack $SM(Stack)$, where we give the Stack as

$C(Stack) = \langle \{Stack()\}, \{push(int), pop(), isEmpty(), peek(), size()\}, \{size_\} \rangle$, as:

$$\begin{aligned} SM(Stack) &= c \cup {}^{con}m^{pub}_{nvoid} \cup {}^{con}m^{pub}_{void} \\ c &= \{Stack()\} \\ {}^{con}m^{pub}_{void} &= \{push(int), pop()\} \\ {}^{con}m^{pub}_{nvoid} &= \{size(), peek(), isEmpty()\} \\ SM(Stack) &= \{Stack()\} \cup \{push(int), pop(), \} \cup \{peek(), size(), isEmpty()\} \\ SM(Stack) &= \{Stack(), push(int), pop(), peek(), size(), isEmpty()\} \end{aligned}$$

□

2.2 Design by Contract™

The basic idea of Design by Contract™ is having a contract between the calling software module, and its callee. Such a contract between the caller and the callee consist of three parts [Mey92]:

Precondition A precondition states what the module expects when it gets called. Referring to the *Stack* example given in Listing 2.1, the precondition of calling a stacks pop method would be having a non-empty stack.

Postcondition The postcondition defines the result of the operation. Sticking to the stack example, the postcondition of the pop method would be: the element on top was returned and removed from the stack.

Invariant The invariant defines a state that has to be true before, during and after the call to the method occured. For example, a member value has not to be changed during the whole call of a classes method.

In terms of Hoare logic [Hoa69], we can now understand a contract as follows. Given a module S a contract consists of the set of preconditions P that hold before executing S. After executing S, the postcondition Q must hold. The invariant I holds before and after the execution. Thus, we can give the annotated method as the following Hoare-triple:

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

Example 5: An example Contract.
Using the *Stack*'s method *pop* shown in Listing 2.1 we give the precondition P as $P = size() > 0$ and the postcondition as $Q = size() == @Old(size(), int) - 1$. When executing the method, the Design by Contract™ system will check if the return value of $size()$ is greater than zero. If this precondition holds, the method is executed. If not, a precondition error is raised, which means that the method was called incorrectly. After the execution of *pop* the postcondition is checked. If the size was not decreased by one, a postcondition error is generated, stating that the implementation did not comply with its contract. □

In addition to the concepts of pre- and postconditions, Design by Contract™ systems such as Modern Jass [Rie07] offer several additional annotations we need for our approach: pure methods, multiple specifications, pre-state access of variables and functions, and model fields.

Pure methods are side effect free methods, that do not alter any member variable of the class. Pure methods are the only methods that are allowed to be used within contracts of other methods. As the methods are executed when the contract is checked, non-pure methods would change the softwares behaviour.

Example 6: Pure Methods.
The *Stack* from Listing 2.1 contains two methods that are flagged as pure, namely $isEmpty()$ in line 20 and $size()$ in line 27. Both methods return values but do not change anything within the class. Thus they can be used within other method contracts. □

To allow to specify multiple behaviour of one single method, it is possible to annotate a method with more than one pair of pre- and postcondition, where one precondition maps to one postcondition:

$$P_1 \rightarrow Q_1 \wedge \ldots \wedge P_n \rightarrow Q_n$$

If no precondition is satisfied when the annotated method is called, a precondition error is raised.

Example 7: Multiple Specifications.
Using the *Stack*'s method $isEmpty()$ from Listing 2.1, one can see that there is an $@Also$

annotation from line 21 to 24 containing two specifications *SpecCase*. One states that if the value of *size*() is greater than zero, the return value must be false. The other one that the return value is true if *size*() is zero. □

Design by Contract™ systems allow to accesss the value of a variable or function before the method was executed, within a postcondition. This is used when specifying how a value has changed or not. Within Modern Jass, this annotation is called @*Old*.

Example 8: Pre-state access.
The *Stack*'s methods *push*(*int*), *pop*() and *peek*() from Listing 2.1 contain postconditions that make use of the @*Old* keyword. The postcondition of *push*(*int*) states that the value of *size*() after the execution must be the value of *size*() before executing *push*(*int*) plus one. Thus it tells the system that the value of *size*() must have increased by one. The same applies to *pop*() and *peek*() where pop is specified to decrease the value of *size*() whereas *peek*() is not allowed to change it at all. □

The concept of model fields was introduced by Cheon et al. [CLSE05]. A model field is a special field that exists only in the context of the classes specifications and is used to describe the internal state of the object. It abstracts the type and name of real implementation. Thus, it is possible to write specifications that do not directly relate to one special implementation [GWW10].

Example 9: Model Fields.
Our *Stack* example from Listing 2.1 contains one model variable: *mSize* which represents the current size of the stack. *mSize* can be used in any specification dealing with the current stack size, regardless how the real implementation is named. Thus, it would be possible to have one implementation where the method to access the actual size is called *size*() and another implementation where this method might be called *depth*(), without changing the specifications. The specifications of *size*() and of the constructor *Stack*() make use of the model field. □

During execution of an annotated module, the contract will be checked. If the contract is violated by any of the two involved parties, the program is terminated immediately. Design by Contract™ thereby eases fault localization. If the precondition is violated, the bug is most likely within the callers code or the callee was misused. If the postcondition

was false, the bug will be within the called software module. Hereby we can identify three main advantages of using Design by Contract™. First of all, the contracts can be used as documentation of the software. Moreover, the contracts can be checked after each refactoring step to ensure the functionality of the refactored code. Therefore the contracts can be seen as quality assurement technique. At last, contracts assist in fault localization.

Definition 4: Contract of a Class.
We define a Contract of a class $C = \langle c, f^{pub} \cup f^{prot} \cup f^{priv}, {}^{con}m^{pub}_{void} \cup {}^{con}m^{pub}_{nvoid} \cup {}^{abs}m^{pub}_{void} \cup \ldots \cup {}^{abs}m^{priv}_{nvoid} \rangle$ as $DbC(C) = \langle mf, inv, mc \rangle$, where mf is the set of model fields in the classes constract and inv is the invariant of the class C.

Furthermore, mc is the set of method specifications $mc = \{ms_1 \ldots ms_m\}$, where one method specification belongs to one constructor $c_j \, \epsilon \, c$ or one method $m_k \, \epsilon \, m$ of class C.

One method specification ms is defined as the tupel of $ms = \langle specs, pure \rangle$ where $pure$ is a specification element that tells the system that no changes to the objects state happen when invoking the annotated method. $specs$ is a set of pre- and postcondition pairs $spec = \langle P, Q \rangle$. Thus we can write mc as:
$mc = \{\langle \{\langle P^1_1, Q^1_1 \rangle, \ldots, \langle P^1_n, Q^1_n \rangle\}, pure \rangle, \ \ldots, \{\langle P^m_1, Q^m_1 \rangle, \ldots, \langle P^m_k, Q^m_k \rangle\}, pure \rangle\}$.

If a pre- or postcondition is not given, we define it to be $true$. □

Example 10: The contract of the Stack
Having the *Stack* from Listing 2.1 according to Definition 1 as
$C(Stack) = \langle \{Stack()\}, \{push(int), pop(), isEmpty(), peek(), size()\}, \{size_\} \rangle$, the contract $DbC(C) = \langle mf, inv, mc \rangle$ is composed as follows:

The set of model fields mf is $\{mSize\}$ as it is the only model field used. The set of invariants inv contains one element, namely that the field $size_ \, \epsilon \, f^{priv}$ never gets

negative: $inv = \{size_ >= 0\}$. The complete Contract DbC(Stack) is now:

$$
\begin{aligned}
DbC(Stack) = \langle \{mSize\}, \{size_ >= 0\}, \\
\{ \\
mc_{Stack} = \ & \langle \{\langle true, mSize == 0\rangle\}, \emptyset\rangle, \\
mc_{pop} = \ & \langle \{\langle size > 0, size == @Old(size) - 1\rangle\}, \emptyset\rangle, \\
mc_{push} = \ & \langle \{\langle true, size == @Old(size) + 1\rangle\}, \emptyset\rangle, \\
mc_{peek} = \ & \langle \{\langle true, size == @Old(size)\rangle\}, \emptyset\rangle, \\
mc_{size} = \ & \langle \{\langle true, @Return == mSize\rangle\}, pure\rangle \\
mc_{isEmpty} = \ & \langle \{\langle size > 0, @Return == false\rangle, \langle size == 0, @Return == true\rangle\}, pure\rangle \\
\}\rangle
\end{aligned}
$$

□

2.3 jCAMEL

The jCAMEL test case generation system is developed at Graz University of Technology by Galler et al. [GPW08] as an extension of JET [CRM07] which was developed to automatically generate a sequence of test cases for java programs based on Design by Contract™ annotations given in JML, the Java Modelling Language [LC04]. Instead of JML, these Design by Contract™-annotations for use with jCAMEL are written in Modern Jass [Rie07]. A test case for one java class consists of a sequence of method-calls of the class under test. As many methods contain parameters, these parameters have to be generated by jCAMEL to be able to actually call it. Therefor some parameter generation sub-systems have been developed to generated the correct instance of parameters:

Random The random data generator can be used for any parameter type. If the parameter is a reference to an object, the random generator instantiates one object and randomly calls methods of this instance to generate a valid, thus contract satisfying, object to use as parameter for a method call. As already pointed out in Section 1, this strategy is not satisfying as it is not able to generate a valid object

in many cases. This generator is always used as fallback-approach, if any of the other generation strategies fail.

Z3 Constraint Solver For primitive data types as integers the Z3 SMT Solver [dMB08] of Microsoft is used.

Regular Expressions This data generator can be used to generate strings that need to satisfy a given precondition.

Manual Data As a test engineer might want to test the software with very special test values there is a possibilty to specify test data by hand.

Mock Objects The idea of mock objects was introduced by Mackinnon, Freeman and Craig in 2001 [MFC01]. Mock Objects simulate the behaviour of real objects and can be used as replacement. An example scenario of use would be using mock objects during testing, when real objects are nor available or to complicated to configure, such as network connections to a server or a complex structure of objects. To use such a mock object, the following information is required:

- What type to mock
- Which methods are called, and how often this is expected
- The allowed input values to these methods and their return values

Using this expected behaviour, mock objects can signal that they where miss used if some call happens, that was not specified before. Thus, by removing dependencies to real objects, tests become lightweight and independent from system components. Within jCAMEL, mock objects are used as an alternative to randomly generating a real object. Tests have shown, that this approach outperforms the random strategy by about 30% with respect to function coverage for the used case studies [GMW10]. The approach developed within this work is also compared using the same reference case studies.

2.3.1 Denotable

As already pointed out, a test case consists of a series of method calls with according parameters. To internally map this sequence, the concept of Denotables introduced by the JET toolkit is used. A Denotable "*uniformly represent various kinds of Java values and objects that can be part of a test case* [jet]." This structure is used within jCAMEL

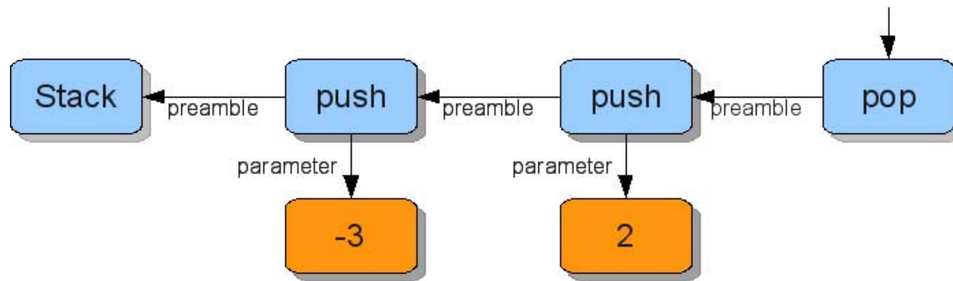


Figure 2.1: The Denotable for the example code given in Listing 2.2. The Denotable containing the method *pop* is the one returned by jCAMEL. It contains the previous method invocations as preamble. The list of parameters is again a Denotable.

```
Stack v0 = new Stack();
v0.push(-3);
v0.push(2);
v0.pop();
```

Listing 2.2: The serialized Denotable of the *Stack* example given in Figure 2.1. At first, the Stack is instantiated. The following method invocations operate on the previously instantiated object.

to represent a test case. To store a test case this structure is searilized to a Java class and saved to a file. An example Denotable of the code shown in Listing 2.2 is given in Figure 2.1.

2.4 Planning

> *"The task of coming up with a sequence of actions that will achieve a goal is called planning." [RN02]*

Planning can be described as finding a sequence of actions, that will bring a system from its initial state S to a desired goal state G. Such a system must provide descriptions of what an action may change in the world and when the execution of the action is valid. 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 [Hoa69] such an action can be written as:

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

Now a plan can be described as sequence of actions $a_1 \dots a_n$ that lead from the initial state S to the goalstate G. The plan is valid if the following sequence can be proven:

$$\begin{aligned} \{S\}\ plan\ \{G\} \Leftrightarrow & \{S\}\ a_1\ \{E_1\} \wedge \\ & \{P_2\}\ a_2\ \{E_2\} \wedge \\ & \wedge\ \dots\ \wedge \\ & \{P_n\}\ a_n\ \{G\} \end{aligned}$$

Definition 5: Planning Domain.
We define the planning domain being the set D of actions a_i: $D = \{a_1, ..., a_n\}$ whereas an action is the above given tuple of an action name, parameters, precondition and effect: $a = \langle name, parameters, P, E \rangle$. □

Example 11: An example domain.
We use a data structure as an example domain. Assume that there are two actions within the planning domain. One is called *push* and increases the size of structure by one. This action can be called at any time. Thus its definition is: $a_{push} = \langle push, \emptyset, true, size + 1 \rangle$. Moreover, there is an action *pop* that reduces the size by one, and is only allowed to be called when the size is greater than zero: $a_{pop} = \langle pop, \emptyset, size > 0, size - 1 \rangle$. The resulting planning domain now is:
$D = \{\langle push, \emptyset, true, size + 1 \rangle, \langle pop, \emptyset, size > 0, size - 1 \rangle\}$ □

Definition 6: Initial State.
We define the initial state S to be the set of all predicates that describe the initial state of the world: $S = \{p_i, ...p_k\}$ □

Example 12: An example initial state.
Using the domain from Example 11 we assume the data structure is intially empty. We can now give the initial state according to Definition 6 as: $S = \{size == 0\}$. □

Definition 7: Planning Goal.
We define the planning goal to be a set G containing all predicates p that describe the desired state of the world: $G = \{p_1, ..., p_n\}$ □

Example 13: An example goal.
Using the domain from Example 11 we want the data structure to have at least two elements. We can now give the planning goal according to Definition 7 as: $G = \{size >= 2\}$. □

Definition 8: Planning Problem.
A planning problem is defined as the tupel $PP = \langle D, I, G \rangle$ where D is the planning domain consisting of all available actions: $D = \{a_1, ..., a_n\}$, S is the initial state and G is the desired goal state, which is a set of facts about the world: $G = \{p_1, ..., p_n\}$ □

Example 14: An example problem.
Using Examples 11 and 13 we have a planning problem $PP = \langle D, G \rangle$ consisting of the domain: $D = \{\langle push, \emptyset, true, size + 1\rangle, \langle pop, \emptyset, size > 0, size - 1\rangle\}$. Initially the stack is empty, thus $S = \{size == 0\}$ and the goal is $G = \{size >= 2\}$. □

Definition 9: A Plan.
We define a plan P as a ordered set of actions a_i that transfer a given initial state S into goal state G, whereas an action a_i can occure multiple times within P: $P = \{a, ..., a\}$□

Example 15: The resulting plan.
Using the planning problem from Example 14, the plan to get a data structure that has at least two elements is calling the *push* method twice:

$$P = \{push, push\}$$

Two assumptions need to be made to make the above definitions being correct. Modern planners rely on both the Closed World Assumption and the Frame Problem:

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. [RN02]

The Frame Problem

The Frame Problem was introduced by McCarthy and Hayes [MH69] in 1969. It states that a machine is not able to decide which conditions will not have changed when invoking an action, unless it is explicitly stated what will not change. This problem affects the design of planners, as they only require action descriptions that state what actually changes by invoking the described action.

2.4.1 Planning Languages

This Section will give an overview of planning languages that have been developed. At first we will look at STRIPS, which PDDL is based on. We will use PDDL within this thesis as planning language.

STRIPS

STRIPS was introduced by R. E. Fikes and N. J. Nillson in 1971 [FN71]. It is important to distinguish between the STRIPS-language which is explained now, and the STRIPS-planner. STRIPS is one of the oldest and most simple representations of operators. It was originally invented to control a robot called “Shakey”. Shakey is able to drive through its world by the means of many connected rooms, push boxes and use light switches. Because of its simplicity and of its expressiveness the STRIPS-language became

on of the most used notions for planning problems and is used in many other planning systems.

The STRIPS-language describes object states and operators acting on these states. States are a conjunction of facts, which are ground literals. Only literals which are true are noted in STRIPS (see the Closed World Assumption). A goal is a conjunction of facts, which need to be reached by invoking the operations. Each operator has a precondition which needs to be completely fulfilled in order to execute the operator. Executing the operator changes the current world state, which is expressed by two sets of literals. At first there is an add lists, which contains literals which are true after invoking the operation. Literals that are false after executing the operation are denoted in the delete list, as removing facts from the current world state means that they become false.

PDDL - Planning Domain Definition Language

The Planning Domain Definition Language was developed by Drew McDermott and the IPC-1 committee [GIP+98]. 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 [ica]. PDDL is syntactically based on LISP and hence it is a prefix language. In PDDL, a planning task consists of:

Objects Objects that exist in the planning domain. PDDL supports typing of Objects.

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

Actions 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 2.3 shows an example PDDL action description. The List of action forms up the planning domain as defined in Definition 5.

```
(: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 2.3: 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 [ica] domains.

Initial states Predicates that are true when starting with the planning process, according to Definition 6.

Goal states The states that form up the goal to be achieved, as given in Definition 7.

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. The PDDL standard has evolved over the last years and got more and more features. As not every planner supports all features of PDDL, a domain file always contains the list of features used, like STRIPS, ADL or numeric variables.

2.4.2 PDDL4J

PDDL4J [Pel09] was initially developed by Damien Pellier, assistant professor at Université Paris Descartes, and is hosted at sourceforge. It was released under the french open source license CeCILL-B. It is intended as PDDL parsing library for use in Java-based planners. It parses PDDL and maps it to an internal object structure. PDDL4J is used within this work as part of the conversion step. We build the internal object structure within our tool and serialize the structure into files. During development of our approach, bugs have been found and corrected and the fixes were contributed back.

Part II

Approach

3 Approach

We have described the jCAMEL tool in Section 2.3. It aims on automatic unit test generation for a given software that was written in Java. It currently lacks of the ability to generate complex objects as test data, that fulfill the currently tested methods specification.

In Section 2.2 we have described the concepts of Design by Contract™. jCAMEL expects the *software under test* to be annotated with Modern Jass annotations, an Design by Contract™ specification language. In Design by Contract™ a specification, a contract, of a software is provided to enhance fault localization. This contracts consist of a precondition P and a postcondition Q, where P must hold before execution of an annotated module, and Q must hold afterwards.

We then described the idea of planning, where a sequence of actions is searched within a planning domain $D = \{a_1, ..., a_n\}$, that transform an initial state S into a goal state G. One action description consists of a precondition, telling the planner when it is allowed to be used. Furthermore, it has an effect on the world, being listed in the effect statement.

We propose the idea of using the information given in the contracts and transform them into a planning problem. That means, that for an object, that has to be in a specific state, we use the contracts information about how the methods of the class change the objects state. This information can be converted, as both planning and Design by Contract™ rely on the concepts of having pre- and postconditions. It can than be passed to the planner, which will generate a sequence of actions, that relate to the invocation of methods of the class. The goal state can be extracted from the specification of the currently tested method we are generating parameters for, the method under test.

Summing up, the generation process can be given as the algorithm shown in Listing 3.1. The contract $DbC(C)$ is converted in the planning domain D as described in Section 3.1. Then the Goal G is generated from the contract of the method under test: $DbC(mut)$ which is introduced in Section 3.2. The domain and goal form up the planning problem according to Definition 8. Then the planner is executed and the returned plan P is converted into the internal data structure of jCAMEL, which is covered in Section 3.4.

```
Denotable AITypeValueGenerator(DbC(C),Pre(mut))
{
  D := generatePlanningDomain(DbC(C));
  G := generateGoalFile(Pre(mut));
  P := runPlanner(D, G);
  denotable := convertPlanToDenotable(P);

  return denotable;
}
```

Listing 3.1: The generation process as abstract algorithm. First of all, the planning domain D as defined in Definition 5 is generated using the contract $DbC(C)$ of the requested type C. Then the corresponding goal G is build from the precondition of the method under test, Pre(mut). These two files are then passed to the planner, which returns the sequence of actions that transform the object from an initial state into the specified goal state. This plan is then converted into a sequence of Java code, internally represented as a Denotable.

3.1 Generation of the Planning Domain

To generate the planning domain it is necessary to know which method invocation changes the state of the object and what exactly gets changed. This information is available as pre- and postcondition pairs in the method specification. As not the whole contract of a class will be usable and not all methods of the class will have influence on its internal state, we will introduce the concepts of State Variables and Action Methods, that are based on the definition of a class and of a contract we gave in Section 2. We will show that having these two concepts, we are able to give an interpretation function, that will translate from a contract to a planning domain.

Definition 10: State Variables

We define $SV(C)$ as the set of State Variables of a class $C = \langle c, f^{pub} \cup f^{prot} \cup f^{priv}, {}^{con}m^{pub}_{void} \cup {}^{con}m^{pub}_{nvoid} \cup {}^{abs}m^{pub}_{void} \cup \ldots \cup {}^{abs}m^{priv}_{nvoid} \rangle$. Methods within these state variables will not change the state itself as they are required to be flagged as pure within their method specification $ms \in mc$ of the classes contract $DbC(C) = \langle mf, inv, mc \rangle$. Thus the set of state variables is the intersection of ${}^{con}m^{pub}_{nvoid}$ and the set of pure methods p, where $p = \{\forall\ ms_m \in mc \mid ms_m = pure\}$ and the public fields f^{pub}.

Thus we can write the set of State Variables as:

$$f^{pub} \cup ({}^{con}m^{pub}_{nvoid} \cap \{\forall\ ms \in mc \mid ms = pure\})$$

□

Example 16: The Stack's State Variables

Having the Stack from Listing 2.1 as $C(Stack) = \langle \{Stack()\}, \{push(int), pop(), isEmpty(), peek(), size($
and the two specified method *size()* and *isEmpty()* within the Stacks contract DbC(Stack),
we can write the State Variables of the Class Stack as:

$$\begin{aligned}
f^{pub} &= \emptyset \\
{}^{con}m^{pub}_{nvoid} &= \{peek(), size(9, isEmpty()\} \\
\{\forall\ ms_m \in mc \mid ms_m = pure\} &= size(), isEmpty() \\
SV(Stack) &= \emptyset \cup (\{peek(), size(), isEmpty()\} \cap \{size(), isEmpty()\}) \\
SV(Stack) &= \{size(), isEmpty()\}
\end{aligned}$$

The Stacks State Variables that are relevant for generating the planning domain are *size()* and *isEmpty()*. □

Definition 11: Action Methods

We define the set of Action Methods of a class C, $AM(C)$ as the set of State Manipulators, that contains at least one State Variable $sv \in SV(C)$ and is not flagged to be pure.

$$AM(C) = SM(C) \cap \{\forall spec \in ms | sv \in SV(C) \subset Q_{spec} \wedge pure = \emptyset\}$$

Thus all constructors and methods, that have a postcondition Q telling the system that it will change one State Variable form up the set of Action Methods $AM(C)$, where changing a State Variable reffers to having the State Variable in a statement of Q. □

Example 17: The Stack's Action Methods

The Stack is given as $C(Stack) = \langle\{Stack\}, \{push, pop, peek, size\}, \{size_\}\rangle$ and its set of State Manipulators as $SM(Stack)$ $\{Stack(), push(int), pop(), peek(), size(), isEmpty()\}$. The contract is

$$
\begin{aligned}
DbC(Stack) = \langle&\{mSize\}, \{size_ >= 0\},\\
&\{\\
&\quad mc_{Stack} = \ \langle\{\langle true, mSize == 0\rangle\}, \emptyset\rangle,\\
&\quad mc_{pop} = \ \langle\{\langle size > 0, size == @Old(size) - 1\rangle\}, \emptyset\rangle,\\
&\quad mc_{push} = \ \langle\{\langle true, size == @Old(size) + 1\rangle\}, \emptyset\rangle,\\
&\quad mc_{peek} = \ \langle\{\langle true, size == @Old(size)\rangle\}, \emptyset\rangle,\\
&\quad mc_{size} = \ \langle\{\langle true, @Return == mSize\rangle\}, pure\rangle\\
&\quad mc_{isEmpty} = \ \langle\{\langle size > 0, @Return == false\rangle, \langle size == 0, @Return == true\rangle\}, pure\rangle\\
&\}\rangle
\end{aligned}
$$

Therefore, the set of Action Methods $AM(Stack)$ is calculated as follows:

$$
\begin{aligned}
SV(Stack) &= \{\textbf{size(), isEmpty()}\}\\
SM(Stack) &= \{Stack(), push(int), pop(), peek(), size(), isEmpty()\}\\
mc_{Stack} &= \langle\{\langle true, \textbf{size()} == 0\}\rangle, \emptyset\rangle\\
mc_{pop} &= \langle\{\langle size() > 0, \textbf{size()} == @Old(\textbf{size()}, int) - 1\}\rangle, \emptyset\rangle\\
mc_{push} &= \langle\{\langle true, \textbf{size()} == @Old(\textbf{size()}, int) + 1\}\rangle, \emptyset\rangle\\
mc_{peek} &= \langle\{\langle true, \textbf{size()} == @Old(\textbf{size()}, int)\}\rangle, \emptyset\rangle\\
mc_{size} &= \langle\{\langle true, @Return == mSize\}\rangle, \textbf{pure}\rangle\\
mc_{isEmpty} &= \langle\{\langle size > 0, @Return == false\rangle, \langle size == 0, @Return == true\rangle\}, \textbf{pure}\rangle\\
AM(Stack) &= AM(Stack) \cap \{\forall spec \in ms | sv \in SV(C) \subset Q_{spec}\}\\
AM(Stack) &= \{Stack(), push(int), pop(), peek()\}
\end{aligned}
$$

□

A planner expects a description of the planning domain as a set of actions $D = \{a_1 ... a_n\}$ as defined in Definition 5. These action descriptions contain a precondition a_i^{pre} that must be satisfied when the specific action is executed. Furthermore, the actions have

effects on the current state which are given in the effect statement a_i^{effect}. An effect description in PDDL is a calculation directive how to calculate the new state out of the previous state.

As defined in Definition 4, Design by Contract™ specifications consist of a precondition P that must hold, when the specified method is invoked and a postcondition Q that must hold after execution. The Design by Contract™ postcondition Q is a logic description of the state after execution of the specified method.

Since a Design by Contract™ postcondition can contain a calculation directive as logic description how the state changes, we can assume that PDDL effects are a subset of Design by Contract™ postconditions. We interpret a Design by Contract™ precondition as an action's precondition and a Design by Contract™ postcondition that consists of calculation directives as a PDDL action's effect. Conditions that do not contain calculation directives can not be translated to PDDL and are ignored.

Definition 12: Interpretation Functions.
We define an interpretation function $\Phi(spec)$ that translates from one specification $spec = \langle P, Q \rangle$ that is part of a method specification $ms = \langle specs, pure \rangle \, \epsilon mc$ of a contract $DbC(C) = \langle mf, inv, mc \rangle$. This function contains two parts. One for translating the precondition P, and one for translating the postcondition Q.

For transforming a contract's precondition P and postcondition Q to a PDDL action description with precondition a_i^{pre} and effect a_i^{effect} we define the Interpretation Functions as follows:

$$\Phi_{pre}(P) \rightarrow a_i^{pre}$$
$$\Phi_{post}(Q) \rightarrow a_i^{effect}$$

□

Example 18: Interpretation Functions.
We use the Stack from Listing 2.1. Having the class as $C(Stack) = \langle \{Stack\}, \{push, pop, peek, size\}, \{size_\} \rangle$, its set of State Manipulators is $SM(Stack) = \{Stack, push, pop\}$. Having the specification of pop as $spec_{pop} = \langle \{ \langle size() > 0, size() == @Old(size(), int) - 1 \rangle \}, \emptyset \rangle$, the example interpretation of its State Manipulator pop is now:

$$\begin{aligned}
\Phi(pop) \rightarrow & \Phi_{pre}(spec_{pop}(P_1)), \\
& \Phi_{post}(spec_{pop}(Q_1)) \\
\Phi(pop) \rightarrow & \Phi_{pre}(size() > 0), \\
& \Phi_{post}(size() == @Old(size(), int) - 1)
\end{aligned}$$

Which then becomes: $a_{pop} = \langle pop, \emptyset, size > 0, size - 1 \rangle$ □

Thus the planning domain for a class C contains at least one action description for every element of the set of State Manipulators $SM(C)$ the class offers, as it is possible that a method contains multiple specifications.

In Design by Contract™ it is valid that an action method $am \in AM(C)$ contains more than one Design by Contract™ specification ($spec_j$) as shown in Example 7. Thus the approach must create an action description for each specification of the Action Methods $am \in AM(C)$, where $spec^{pre}$ is the according precondition and $spec^{post}$ the postcondition of the specification *spec*.

As Java and Modern Jass allow a wider range of valid characters as part of method names than PDDL, names must be mangled. This is achieved by a mangling function $\mu^{in}_{out}(name)$ which maps between an input and an output language.

Definition 13: Name Mangling.
We define the name mangling functions $\mu^{in}_{out}(name)$ as functions that translate from valid names in the input language *in* to a valid name in the output language *out*:

$$\begin{aligned}
\mu^{java}_{pddl}(java_typename) &\rightarrow pddl_typename \\
\mu^{pddl}_{java}(pddl_typename) &\rightarrow java_typename \\
\mu^{pddl}_{java}(\mu^{java}_{pddl}(typename)) &\rightarrow typename
\end{aligned}$$

□

Not available Characters in PDDL

PDDL supports a more narrow range of valid input characters. Table 5.3.1 gives a collection of unsupported characters and their occurences in Java and how we translate them to valid input.

Mangling of State Manipulators

Fully qualified method names in Java consist of the name itself, and a list of their fully qualified parameter type names. This fully qualified name is translated by concatenating the name and parameter type names. See Example 19.

Constructors are handled equaly. Their fully qualified type name additionaly includes the class' package and name. This information is also used when translating from Java to PDDL. All translated names are lower cased.

Example 19: Name Mangling of a Java Function.
Consider the Stack being in package *java.util* and using integers as containing data type. The translation of its paramterless constructor and the push method that takes one argument will look as follows:

$$constructor = public\ java.util.Stack()$$
$$\mu^{java}_{pddl}(constructor) \rightarrow java_util_stack$$

$$method = public\ void\ push(int\ element)$$
$$\mu^{java}_{pddl}(method) \rightarrow push_int$$

□

Definition 14: We can now define the transformation of a class $C = \langle c, f^{pub} \cup f^{prot} \cup f^{priv}, {}^{con}m^{pub}_{void} \cup {}^{con}m^{pub}_{nvoid} \cup {}^{abs}m^{pub}_{void} \cup \ldots \cup {}^{abs}m^{priv}_{nvoid} \rangle$ and a contract $DbC(C) = \langle mf, inv, mc \rangle$ into a planning domain $D = \{a_1, ..., a_n\}$ as:

$$D = \{a \mid \forall_{am \,\epsilon\, AM(C)} : \forall_{spec \,\epsilon\, am} : a_{am,spec} = \langle \mu^{java}_{pddl}(am), \emptyset, \Phi_{pre}(spec^{pre}), \Phi_{post}(spec^{post}) \rangle \}$$

So for every specification of every action method of C a PDDL action specification is generated using the mangled name and the interpretations of the pre- and postcondition. □

Example 20: Interpretation of the Stack.
Using the Stack's class definition of
$C(Stack) = \langle\{Stack()\}, \{push(int), pop(), isEmpty(), peek(), size()\}, \{size_\}\rangle$ and its contract as

$$
\begin{aligned}
& DbC(Stack) = \langle\{mSize\}, \{size_ >= 0\}, \\
& \quad \{ \\
& \quad\quad mc_{Stack} = \langle\{\langle true, mSize == 0\rangle\}, \emptyset\rangle, \\
& \quad\quad mc_{pop} = \langle\{\langle size > 0, size == @Old(size) - 1\rangle\}, \emptyset\rangle, \\
& \quad\quad mc_{push} = \langle\{\langle true, size == @Old(size) + 1\rangle\}, \emptyset\rangle, \\
& \quad\quad mc_{peek} = \langle\{\langle true, size == @Old(size)\rangle\}, \emptyset\rangle, \\
& \quad\quad mc_{size} = \langle\{\langle true, @Return == mSize\rangle\}, pure\rangle \\
& \quad\quad mc_{isEmpty} = \langle\{\langle size > 0, @Return == false\rangle, \langle size == 0, @Return == true\rangle\}, pure\rangle \\
& \quad \}\rangle
\end{aligned}
$$

the interpretation according to the interpretation functions given in Definition 12 is now: $D = \{\langle Stack, \emptyset, \neg instantiated, instantiated \wedge size() = 0\rangle, \langle push_int, \emptyset, instantiated \wedge size() > 0, size()+1\rangle, \langle pop, \emptyset, instantiated \wedge size() > 0, size()-1, \rangle\langle peek, \emptyset, instantiated \wedge size() > 0, size() = size()\rangle\}$. The according serialization as PDDL is given in Listing 3.2. □

3.2 Generation of the Planning Goal

The planning goal definition in PDDL contains two main parts. One is the initial state, where the planner should start planning from. The other one is the desired goal state, the planner should achieve.

The initial state description needs to contain all state variables $sv \in SV(C)$ used in

```
(: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 3.2: The planning domain generated from the *Stack* example given in Listing 2.1.

Type	used PDDL Type	Default-Value
boolean	fact	false (being not defined)
numeric values	fluent	0.0
references	fact	null (fact is not defined)

Table 3.1: Default values for state initialization

the planning domain. These state variables need to have initial values assigned. If a postcondition of a constructor does not state all state the values of all state values, we use default values which are given in Table 3.2.

To generate the goal state, the Design by Contract™-precondition of the method under test must be analyzed. Every part of the precondition P that contains at least one state variable $sv_i \, \epsilon \, SV(C)$ of the current generated parameters class need to be taken into account. All other parts can be ignored as they don't include relevant information for the current generation request.

Definition 15: Relevant parts of the specification.

We define R as the set of relevant parts of a specification $R = \{p_1, ..., p_n\}$. A part p_i of a specification is defined as one logic expression. Thus a specification can be seen as the conjunction of parts p using the logic operators $\wedge$, and $\vee$. A part is relevant if it

```
@Pre("stack != null && stack.size() >= 2 && number < 0")
void useStack(Stack stack, int number)
{ ... }
```

Listing 3.3: An example method that needs a Stack with size 2 as one parameter and a primitive as second parameter. For generation of the Stacks instance, jCAMEL will select our planning approach.

contains at least one state variable $sv_i \epsilon SV(C)$. □

Example 21: Relevant parts of the specification.
We use the method *useStack(Stack stack, int number)* from Listing 3.3. It's precondition consists of three parts. Part one forces the parameter *stack* to be not null. Part two requires the method *size()* of the parameter *stack* to return a value greater than or equal to two. The third and last part requires the second parameter number to be smaller than zero.

After generation of the planning domain according to Section 3.1 the set of state variables is known to be $SV(Stack) = \{size(), isEmpty()\}$. Analyzing the precondition shows that only one part contains one of these state variables. All others can be ignored when generating the planning goal. The remaining parts of the precondition can now be translated into PDDL using the same conversion rules as when transforming the planning domains preconditions.

One more part contains information about the *Stack's* state: **stack != null** tells the planning system that it needs to return an instance. If the specification would be **stack == null** the planning system does not need to be invoked and a **NullDenotable** can be returned immediately. If no specification is given that states wether the reference must not be null or not the system generates an instance by default. □

```
(:init (= (size) 0.0))
(:goal (and (stack_instantiated) (>= (size) 2)))
```

Listing 3.4: The according planning goal and initial state for the example given in Listing 3.3. The initial state is created using the types default values as shown in Table 3.2.

3.3 Running the Planner

After the two input files for the planning process, the domain D and the goal description G, have been generated, the planner needs to be started. As PDDL is used as planning language within the bi-annual planning competitions [ica], a wide range of planners exist that will understand the input, and will produce a standardized output.

3.4 Generation of Java Code based on a Plan

When the planner successfully returned a plan P, it needs to be converted back into a sequence of java code. The returned plan consists only of a sequence of actions. These actions need to be mapped back to concrete methods of the class using the name mangling function μ^{pddl}_{java}. As the planner will have no information about parameters of the method these parameters need to be created during translation of the action. They will be generated by recursively calling the whole test data generation process for the current action, as shown in Listing 3.5. We call this strategy Optimistic Test Data Generation, which we will cover in detail in Section 3.5.3.

3.5 Details

This Section covers details of our approach. In Section 3.5.1 we will describe details on the special handling of constructors. Then we will point out our ideas of translating specific Design by Contract™ specifications into their according PDDL representation

```
Denotable convertPlanToDenotable(Plan P)
{
  Denotable result := empty
  foreach (action in P) {
    result := addMethodInvocation(action)
    foreach (paramter in action) {
      callTypeValueGenerator(paramter, method_specification)
    }
  }
  return result
}
```

Listing 3.5: Pseudocode algorithm for retranslating the plan to a Denotable. For each action a in the plan P a method invocation is added to the denotable. If this method invocation needs parameters, they are generated by calling the according type value generator.

in Section 3.5.2. At the end, we will compare two different concepts of dealing with unknown information in Section 3.5.3, where we describe our idea of Optimistic Test Data Generation and Section 3.5.4, which deals with Pessimistic Test Data Generation. We will close with a review, why we have chosen Optimistic Test Data Generation as our approach.

3.5.1 Handling of Constructors

As the set of State Manipulators of a class $C = \langle c, f^{pub} \cup f^{prot} \cup f^{priv}, {}^{con}m^{pub}_{void} \cup {}^{con}m^{pub}_{nvoid} \cup {}^{abs}m^{pub}_{void} \cup \ldots \cup {}^{abs}m^{priv}_{nvoid} \rangle$ contains both constructors $c \in C$ and public concrete methods it is necessary to invoke a constructor call before any method. Furthermore, a constructor must be called only once. Therefore, we introduce a predicate $p_{instantiated}$ that states whether a concrete object of the class C was instantiated using a constructor $c \in C$.

To ensure that a constructor is used only once, we add the negated predicate $p_{instantiated}$ to every precondition of a constructors action a^{pre}_c. Furthermore, $p_{instantiated}$ is added to the actions effect a^{effect}_c. Thus a constructor's action description will always look like shown in Example 22.

Moreover, we add the predicate $p_{instantiated}$ to every method's action precondition a_m^{pre} to ensure a planner will use actions that relate to concrete instance methods only, after constructing an instance of class C. The name of the predicate is composed of the lower cased class name and the term "instantiated".

Definition 16: Special Interpretation of Constructors.
We define $p_{instantiated}$ to be a predicate that is true after a constructor has been called once. The precondition for calling a constructor is extended by not having called a constructor before.

We have defined a class as $C = \langle c, f^{pub} \cup f^{prot} \cup f^{priv}, {}^{con}m_{void}^{pub} \cup {}^{con}m_{nvoid}^{pub} \cup {}^{abs}m_{void}^{pub} \cup \ldots \cup {}^{abs}m_{nvoid}^{priv} \rangle$. As a constructor needs to be called before any call to a concrete instance method who form up our State Manipulators together with constructors, we have to ensure that one constructor call is an action, the planner chooses first. This is done using a special interpretation function $\Phi_{pre}^{con}(c)$ and $\Phi_{pre}^{con}(c)$:

$$\Phi_{pre}^{con}(c) \rightarrow \neg p_{instantiated} \wedge \Phi_{pre}(c)$$
$$\Phi_{post}^{con}(c) \rightarrow p_{instantiated} \wedge \Phi_{post}(c)$$

□

Example 22: Adding the Instantiated Predicate to a constructors action.
Assuming that the Stacks parameterless constructor would not have any specification at all, the resulting action description would look like in Listing 3.6. The name is mangled using the mangling function μ_{pddl}^{java} given in Definition 13:

```
(:action java_util_stack
        (:precondition (not (stack_instantiated)))
        (:effect (stack_instantiated))
)
```

Listing 3.6: The translation of the Stacks parameterless constructor to a PDDL action. Assuming the constructor had no Design by Contract™-specification the precondition would be having not yet instantiated the stack. Using this action within a plan results in an instantiated object of the class Stack.

3.5.2 Translation of Specifications

In this Section we will go into detail how the translation from the Design by Contract™ pre- and postcondition of a methods specification is done. For each translatable part of a specification, we will give the according interpretation function Φ.

We use only those parts of the specification that contain at least one State Variable of C as defined in Definition 10 for the translation of the pre- and postcondition. If a part does not contain any State Manipulator it is ignored. A part of a specification is defined as one logic statement. The concatenation using logic operators form up a pre- or postcondition. For details on what we define to be a relevant part of a specification see Definition 15.

Arithmetical Operations

Arithmetical operations can be translated to PDDL using fluents. Fluents are the PDDL representation of numbers.

```
@Post("a() == b() + 17")
public void calc(){ ... }
```

Listing 3.7: An example annotation stating that the execution of *calc* leads to *a()* having the value of *b()* plus 17.

The interpretation function of an arithmetical operation are:

$$\Phi(id_1 == id_2\ +\ id_3) \rightarrow (and\ (assign\ id_1\ id_2)\ (increase\ id_1\ id_3))$$
$$\Phi(id_1 == id_2\ -\ id_3) \rightarrow (and\ (assign\ id_1\ id_2)\ (decrease\ id_1\ id_3))$$
$$\Phi(id_1 == id_2\ *\ id_3) \rightarrow (and\ (assign\ id_1\ id_2)\ (scale{-}up\ \ id_1\ id_3))$$
$$\Phi(id_1 == id_2\ /\ id_3) \rightarrow (and\ (assign\ id_1\ id_2)\ (scale{-}down\ \ id_1\ id_3))$$

```
(:action calc
          :effect (and (assign (a) (b)) (increase (a) 17))
)
```

```
@Pre("true || (false && true)")
public void do(){ ... }
```

Listing 3.9: Example of logic conjunctions within a precondition.

Listing 3.8: The translation of the example annotation given in Listing 3.7 into PDDL.

Logic Conjunctions

Logic conjunctions within preconditions can be translated directly to an actions precondition description, as shown in Listing 3.10.

$$\Phi(exp_1 \,\&\&\, exp_2) \rightarrow (and\ \Phi(exp_1)\ \Phi(exp_2))$$
$$\Phi(exp_1 \,||\, exp_2) \rightarrow (or\ \Phi(exp_1)\ \Phi(exp_2))$$

```
(:action do
        :precondition (or (true) (and (not(true)) (true)))
)
```

Listing 3.10: Translation of logic conjunctions into PDDL

OLD-values

Design by Contract™ systems support accessing the value of a variable before the invocation of a method in its postcondition, which is explained in Section 2.2. This *pre-state* value can be accessed in any part of a postcondition. As PDDL is evaluated from left to right, the ordering of condition parts is decisive for the evaluation of arithmetical operations. If we would translate the postcondition as a simple arithmetical expression, the effect would not be correct when an old value is accessed after the original value has changed.

```
@Pre("true")
@Post("x() == @Old(x(),int) + 1 && y == @Old(x(),int)")
public void do() { ... }
```

Listing 3.11: An example postcondition with two value assignments. If the pre-state access, the @*Old* annotation is not handled correctly we can see undefined behaviour depending in the order of the specification's parts. An example of invalid value assignment based on this specification is given in Table 23

```
(:action do
  :precondition (true)
  :effect (and (increase x 1) (assign y x))
)
```

Listing 3.12: An incorrect translation of the postcondition given in Listing 3.11. The ordering of the condition's parts effects the evaluation as the old value is ignored, as shown in Table 23.

Example 23: Ignoring the old value leads to incorrect behaviour.

As described, the ordering of statements might have influence on the evaluation if we introduce no mechanism to access old values. Listing 3.11 shows a postcondition where the ordering has influence on the evaluation when the pre-state values will not be handled correctly.

statement	x	y
initial	1	1
(increase x 1)	2	
(assign y x)		2
after evaluation	2	**2** ✗
correct assignment	2	1

Table 3.2: Incorrect values during evaluation of the PDDL action effect, because no pre-state helper variables are used. See Listing 3.12 for the according planning domain action.

statement	x	y	_pre_x
initial	1	1	-
(assign _pre_x x)			1
(increase x 1)	2		
(assign y _pre_x)		1	
after evaluation	2	**1** ✓	
correct assignment	2	1	

Table 3.3: Correct values during evaluation of the PDDL action effect due to the use of old value helpers.

If the parts of the postcondition would have different ordering, the evaluation would be correct. To overcome this issue we assign the old value to a helper variable before every other statement in the effect list to access it within the rest of the effect statement without any side-effect. The name of such a variable is composed of the string **_pre_** and the variables name. This limits the range of valid State Variable names. Thus, State Variable names that start with **_pre_** should not be used within Design by Contract™ contracts as they might interfere with these helper variables.

Example 24: The implicit framing condition.
Using the postcondition from Listing 3.11 a correct translation looks like in Listing 3.13. The value of x is assigned to an helper variable and then the values can be changed as stated in the postcondition.

```
(:action do
  :precondition (true)
  :effect (and (assign _pre_x x)
     (increase x 1)
     (assign y _pre_x))
)
```

Listing 3.13: The implicit framing condition achieved by assigning values before changing them. This ensures order-independent, correct evaluation of the statements as shown in Table 24.

□

Boolean values: Handling true and false

Due to the nature of PDDL, the boolean values **true** and **false** do not exist by default. They have to be abstracted by predicates. Each generated planning domain will contain a predicate **true** that is valid during the whole planning process. **False** can now been translated to $not(true)$.

If a state variable $sv \epsilon SV$ is a boolean state variable, it is translated to a PDDL predicate p_{sv}.

$$\Phi(sv == true) \rightarrow (sv)$$
$$\Phi(sv == false) \rightarrow (not(sv))$$

```
@Pre("state_variable() == false")
@Post("state_variable() == true")
public void do() { ... }
```

Listing 3.14: A boolean state variable used in a method specification.

Thus, the action definition will look as in Listing 3.15. The actions precondition is the negated predicate p_{sv} and the postcondition is p_{sv} according to the name mangling rules given in Definition 13.

```
(:action do
          :precondition (not (state_variable))
          :effect (state_variable)
)
```

Listing 3.15: Translation of the boolean state variable from Listing 3.14 into PDDL

```
@Post("x() == @Old(x(), int) - 1 && x() == y() + 1")
public void do { ... }
```

Listing 3.16: An example method with a postcondition that contains two parts giving expectations for the same State Variable.

Having more definitions of the same State Variable

If a method specification *spec* contains more parts that define the result of the same State Variable it is sufficient to use only one part as effect of the corresponding action, as the other parts contain different calculation rules for the same State Variable. All parts must define the same value calculation. An example postcondition is given in Listing 3.16

Here it is sufficient to use either the information that x is decreased by one or that x is calculated by adding one to the value of y, as both calculations must yield the same result.

Having more Specifications of one Method

If a state manipulator $sm \in SM(C)$ of a class $C = \langle c, f^{pub} \cup f^{prot} \cup f^{priv}, {}^{con}m^{pub}_{void} \cup {}^{con}m^{pub}_{nvoid} \cup {}^{abs}m^{pub}_{void} \cup \ldots \cup {}^{abs}m^{priv}_{nvoid} \rangle$ contains more method specifications $spec \in ms_{method} \in mc$ within the classes contract $DbC(C) = \langle mf, inv, mc \rangle$, we have to create one planning domain action for each specification *spec*.

Example 25: A Method with more than one specification.
We use an example class C with one State Variable $SV(C) = \{state\}$ and one State Manipulator $SM(C) = \{set\}$. The contract $DbC(C) = \langle mf, inv, mc \rangle$ of class C is $DbC(C) = \langle \emptyset, \emptyset, \{ms_{set} = \langle \{\langle x > 0, state() == false \rangle, \langle x <= 0, state() == true \rangle\}, \emptyset \rangle\} \rangle$ as given in Listing 3.17. Thus we have two specifications *spec* in the set of method specifications for method *set*. Using a parameter greater than zero results in setting the State Variable *state* to false. Using a paramter less than or equal to zero sets *state* to true. Therefore, the result of invoking the according action a_i within the planning domain $D = \{a_1, ..., a_n\}$ depends on which precondition holds.

```
@Also(
   @SpecCase(pre="x > 0", post="state() == false"),
   @SpecCase(pre="x <= 0", post="state() == true") )
public void set(int x) { ... }
```

Listing 3.17: An example method with two specifications. Using a parameter greater than zero results in setting the State Variable *state* to false. Using a paramter less than or equal to zero sets *state* to true. The generated PDDL actions are given in Listing 3.18.

When such an action is used within a plan the retranslation step needs to take care of correct parameter creation. Only the part of the specification that relates to the action is passed to the parameter generation of jCAMEL. This ensures, that the precondition of the correct specification holds when invoking the method, and the assumptions made by the planner are correct afterwards. If we would pass the whole set of specification the generated parameter might fulfill the uninteded part of the specification and the planned object would not reach the desired state. This depends on the implementation of the used value generator.

```
(:action set_1
  (:precondition (> x 0))
  (:effect (not (state)))
)

(:action set_2
  (:precondition (<= x 0))
  (:effect (state))
)
```

Listing 3.18: The two generated PDDL actions for the State Manipulator given in Listing 3.17

Getter and Setter-Methods

Getter and Setter-Methods are methods that set a private or protected fields value. We expect a setter methods pre- and postcondition to specify the state of the reference of the parameter used to set the field. This means, it has to be stated, that a reference will

be null or not. Furthermore, the postcondition has to state if the field was set to be null or not by using either the associated getter method, or a model variable as described in the next Section.

If a field was set to be not null, we will generate a predicate that states that the reference is not null. Otherwise, the predicate will be removed from the domain, by negotiation. Thus, we interpret specifications containing getter methods that compare references as:

$$\Phi(getX() == null) \rightarrow \neg getX$$
$$\Phi(getX()! = null) \rightarrow getX$$

Model-Variables

Model variables can be used within Modern Jass as substitute for real implementations within specifications as described in Section 2.2. These model variables can substitute a State Variable $sv \epsilon SV(C)$ of a class $C = \langle c, f^{pub} \cup f^{prot} \cup f^{priv}, {}^{con}m^{pub}_{void} \cup {}^{con}m^{pub}_{nvoid} \cup {}^{abs}m^{pub}_{void} \cup \ldots \cup {}^{abs}m^{priv}_{nvoid} \rangle$. Thus a method specification can include a model variable instead of a State Variable. We now have to map between a state variable and its representing model variable. This is done by checking the postcondition of the State Variables Design by Contract™-specifications. If the return statement contains a model variable, all occurrences of the model variable within the contract can be replaced by the State Variable.

Example 26: Model Variables and State Variables
We use a the class *Stack* from Listing 2.1 with one model variable *mSize*. According to Definition 10, the set of State Variables of the example *Stack* class is $SV(Stack) = \{size()\}$. The contracts of the methods given in Listing 2.1 either contain a State Variable $size()$ or the corresponding model variable *mSize*. Checking the postcondition of $size()$ tells the system that the method returns the value of the model variable. Thus we can substitute all occurences of *mSize* by $size()$ without changing the contracts meaning. □

```
@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 3.19: An exemplary method under test. The specification forces the input parameter to have at least 2 elements. Pushing elements to the Stack is allowed only if both methods of the complex type have been called with the right parameter.

3.5.3 Optimistic Test Data Generation

As already pointed out, two main strategies for the generation of the planning domain can be identified. We call the first one Optimistic Test Data Generation. Consider the example given in Listing 3.19. A test for the *methodUnderTest* should be generated. Therefore, a Stack needs to be instantiated as parameter. Furthermore, this Stack needs to have at least two elements of some complex type. This instance of the complex type needs to be in a very specific state.

To achieve this state, jCAMEL will choose the AITypeValueGenerator as its first gener-

```
(:action Stack
  (:precondition (true))
  (:effect (and (= (size) 0) (stack_instantiated) )
)

(:action push
  (:precondition (stack_instantiated))
  (:effect (increase (size) 1))
)
```

Listing 3.20: The resulting planning domain of the Stacks push method and constructor from Listing 3.19. So the action can called at any time after construction of the class, resulting in an increasing size.

ation strategy to generate an instance of Stack. The AITypeValueGenerator will receive the specifiactions of $methodUnderTest$ as well as all specifications of the Stack as its input data. As already pointed out, the specifications of the Stack will become the planning domain, and the specifications of the $methodUnderTest$ will become the planning goal or problem. When generating the planning domain, only those parts of the methods annotation will be translated to an action specification, that deal with the Stacks internal state. The complete precondition of the Method *push* deals with the input parameter, and is thus not of interest in the current conversion step. The postcondition deals with the internal state of the object and therefore is of interest. It will be converted. The only precondition for calling this method is that the class is instantiated, and the postcondition tells the system, that it will raise the size by one. The expected result is shown in Listing 3.20.

After converting the Stacks specifications the the planning domain, the precondition of the $methodUnderTest$ will be converted to the planning problem. As Optimistic Test Data Generation is defined, only those parts of the specifications that directly relate to the current parameter will be taken into account. In this example it is the whole specification, telling the system that the parameter has to be instantiated and contain at least two elements. The expected problem definition is given in Listing 3.21. The planner is now called with this information, and is expected to return a sequence that leads to the desired state. This is: construction of the class, and calling the push method

twice. On retranslation from the planners solution to Java code, the translator will map the actions to methods of the class Stack.

```
(:goal (and (stack_instantiated) (>= (size) 2) ) )
```

Listing 3.21: A goal defintion forcing the two facts, having an instantiated class containing at least two elements, to become true.

First of all, the constructor is called. Then the first push method is called on the resulting object. The translator now needs to check if there are any parameters, that are not known by now. As the parameter of type *ComplexType* is currently not known to the system, it needs to call the jCAMEL generation algorithm to receive one. jCAMEL will choose an appropriate generation strategy and return an object of *ComplexType* that fullfils the push methods specifiction. In this case, it will also call the AITypeValueGenerator, passing the precondition of $push(int)$ and the whole list of specifications of *ComplexType*. AITypeValueGenerator will invoke the same steps as already described and tries to return the object. When calling the setA/B Methods of *ComplexType* it will again call jCAMEL to receive corresponding values. After that, the instance of *ComplexType* is ready and passed to the calling instance of the AITypeValueGenerator which now tries to get parameters for the second call to the push method, and finally returns the complete sequence of Java code.

Through recursively calling of the whole jCAMEL-process the AITypeValueGenerator only needs those parts of information that are currently related to the problem and can ignore the rest. This strategy gives the system the ability to change the generation strategy during the generation process even for the same type of requested parameter. If one step fails it can be replaced by another generation strategy. This gives better failover capabilities.

Thus the optmistic part of the generation process is delegating creation of data to subsequent instances and expect that these instances will know how to create these data.

```
Stack s0 = new Stack();

ComplexType c0 = new ComplexType();
c0.setA(1);
c0.setB(-1);

s0.push(c0);

ComplexType c1 = new ComplexType();
c1.setA(2);
c1.setB(-5);

s0.push(c1);
```

Listing 3.22: The expected sequence of Java code, that generates a stack containing two valid instances of ComplexType. For the specifications of the classes see Listing 3.19. The integer parameters used here are generated by recursively calling jCAMEL with the methods specification as parameter.

3.5.4 Pessimistic Test Data Generation

Pessimistic Test Data Generation can be seen as direct opposite to Optimistic Test Data Generation. Here the planning system tries to generate as much data as possible within one step, as it assumes that data can only be generated when everything is known to do so. Using the example from Listing 3.19, the generation approach would work as follows:

First of all, the system needs to check which parameters of the *methodUnderTest* are requested. This is a *Stack* containing valid objects of type *ComplexType*. For each of the methods of *ComplexType* this check must be done also. This results in having Objects of type *ComplexType* in a specific state and having an instance of *Stack* in a specific state. All information of these two types are incorporated into a single planning domain. The desired state of these Objects can now be expressed within one single planning problem. For each possible different instance of the same type, different planning goal facts and planning variables need to be generated. This changes the layout of planning domain and planning problem and yields a more complex output of the conversion algorithms. Also retranslation will become more complex, as instances need to be clearly seperated. As

advantage, one can see that the complete example can be planned within one planning step. If an error occures during one of the translation steps or the planner can not return a result, as it is simply not possible, the whole generation process will fail.

For generation of data other than instances of classes, so called primitive data, the Pessimistic Test Data Generation still needs to call jCAMEL. We have chosen to implement the Optimistic Test Data Generation strategy, because of its better failover capabilities. The two strategies are compared in Table 3.4.

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 3.4: Comparison of the two previously describes generation strategies.

3.6 Limitations

As all fluents and predicates that are used within a planning domain $D = \{a_1, ..., a_n\}$ need to be declared and initialized in the planning goal definition, we assign a default value to every variable. If no specification of the class' contract $DbC(C) = \langle mf, inv, mc \rangle$ contains an initialization of the variable the default value will be used throughout the planning process. This might lead to a not solvable planning problem. Thus, if a constructor of the class changes the value of one State Variable the specification must contain this fact.

Furthermore, two valid Modern Jass constructs can not be translated to PDDL as there is no corresponding expression. These are the forall and exists keywords, where it is possible to specificy expectations on the content of collections, as lists or arrays. PDDL does not support the idea of collections. Therefore, a translation is not possible.

The main limitation of our approach is, that effects within PDDL describe how the state changes, and postconditions within Design by Contract™ give an definition on how the state must look like after executing. As already pointed out, the description of how the state changes can be seen as a subset of the possibilities Design by Contract™ offers when specifying postconditions. If a specification contains such a description we are able to generate the according PDDL action. Otherwise, a translation is not possible. Thus we have to limit the valid set of Design by Contract™-annotations our approach can handle.

4 Related Work and Alternative Approaches

Generation of test data is a wide field of research. There exist many approaches to generate test data. Some of them are directly related to our approach, and will be discussed in Section 4.1. Then we give an overview of different approaches, such as random generation or using evolutionary algorithm in Section 4.3.

4.1 Related Work

In this Section, we will give an overview of work related to this thesis. First of all, we will discuss the approach introduced by Andreas Leitner [Lei04] that is intendet to automatically generate tests for Eiffel™ [Mey87] programs based on their contract. This approach is related to the one presented in this work. After that, we will review the ideas of Gupta et al [gup04, GFB+07], who extended the Graphplan planning algorithm to generate test data. After dealing with related approaches, we will give an overview of alternative ways to generate test data. We will take a look at KORAT [BKM02], which also uses Design by Contract™-annotations to support the generation of software tests. We then give an overview of Random approaches in Section 4.3.2 which overall tends to generate a high number of tests. Within this Section, we will discuss two extensions that try to overcome the limitations of generating data randomly, namely *Directed Automated Random Testing* [GKS05] in Section 4.3.2 and *Feedback Directed Random Test Generation* [PLEB07] in Section 4.3.2.

4.1.1 System Testing with an AI-Planner

One of the first work done in the area of AI-planning systems used to generate test data, was carried out by Adele Howe et al. [vMSDH00, MHvML]. Their approach is an extension of *Sleuth*, an automatic test generation system. They have created a planning domain representing the command language of a tape store machine. Based on this planning domain, a test engineer had to specify the goal of the testcase by defining the planning goal and initial state. The planner (UCPOP [PW92]) then generates the plan, that is converted back to the command language automatically. The set of generated command is then used as a test series.

Their basic intention was to prove whether a planner would come up with reasonable and understandable test cases. They compared the automatically generated tests with the ones a test engineer would generate by hand.

They conclude, that using planners to come up with test sequences has some advantages over doing it by hand. First of all the generated sequence is correct, thus the set of commands will not violate any syntactic rules. Furthermore, the test sequences generated by the planner differ from the ones test engineers generated. They argue, that this would give another view of the *system under test*, as test engineers might be to close to the system.

4.1.2 The Planning Extension for TestStudio

In his master thesis, "Strategies to Automatically Test Eiffel Programs" [Lei04], Andreas Leitner introduced an extension to TestStudio [Ciu04, Gre04]. TestStudio was developed at ETH Zurich to automatically generate tests for Eiffel programs. As Eiffel™ natively supports the concept of Design by Contract™, the system uses this specifications to generate planning problems as in our approach.

As illustrated in Figure 4.1, the test data generation process of the system is designed as follows. First of all, the Problem Generator creates a planning problem based on the Design by Contract™-specifications of the Eiffel™ source code. This planning problem is then passed to the planning system. They use bifrost [Jen03], a planner using bi-

nary decision diagrams and extNADL as planning problem description language. After successfully planning, the plan is retranslated to Eiffel™ code and executed. If the precondition of the currently tested method is violated, the planning problem is enriched with information gathered during execution of the planned piece of code. Using the enriched planning problem, these steps are repeated until either the requirements are fulfilled or a timeout has been reached.

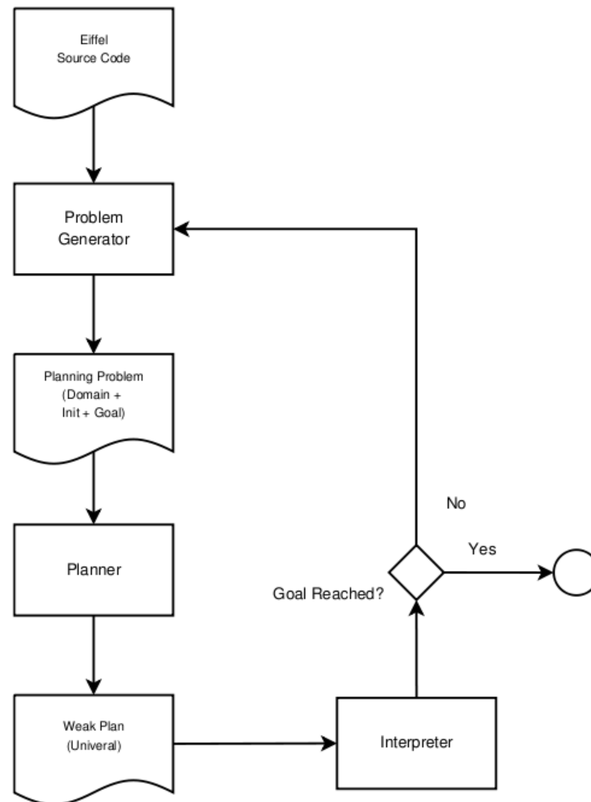


Figure 4.1: The planning flowchart of Leitners extension to TestStudio [Lei04]. First of all, the Eiffel™ sourcecode, namly the Design by Contract™ specifications are translated into a planning problem. This planning problem is then executed by a planner to generate a weak plan. This plan is retranslated to Eiffel™ code and executed. If the execution does not satisfy the goal, data gathered during execution is used to refine the plan. This loop is done until the generated plan fulfills the requirements.

Their approach lacks one main feature. They are not able to generate tests for methods that need parameters. This is one of the main differences to our approach. By using our so called Optimistic Test Data Generation approach, we are able to do so, by delegating the generation of parameters to a second planning instance. As the planner operates on binary decision diagrams, Leitner had to limit his approach to use only boolean facts

within planning domains. Furthermore, he adds integers with a precision of two bits, by abstracting them to two boolean values.

4.1.3 MEA-Graphplan

The MEA-Graphplan technique was developed by Gupta et al. [gup04, GFB+07] as an extension of the Graphplan algorithm, a planning technique. One of the problems that Graphplan suffers is the possibility of state space explosion. During the graph expansion phase, Graphplan has no information about whether executing an action leads towards reaching a goal or not. If the *program under test* is complex, and real world problems tend to be complex, the possibility of state space explosion is high. The Idea behind MEA-Graphplan is to inform the graph expansion algorithm whether an action leads towards reaching a goal or not. This is done by generating a regression matching graph, which then is used to direct the graph expansion. The regression matching graph is generated from each goal condition by regressing backwards until any of the given start conditions is reached. Now the standard Graphplan algorithm is adapted to only consider actions that are in the same level of the regression matching graph.

Furthermore, they proposed a general system to apply AI-planning techniques to software testing. This framework consists of two modules. The Planning Domain Generator and the Planner itself. The Planning Domain Generator is used to create the input parameters for the planner, based on the software under test (like out of UML State Chart Diagrams). They introduce the following set of input parameters to the Planning Domain Generator {S, O_f, V_f, T, C} , whereas S is the already mentioned state specification, O_f is a set of operations defined by the *software under test* which alter the state space, V_f is a set of function which return information about the state space. T represents the testing requirements and C some global constraints. The Planning Domain Generator now generates planning parameters {O, s_0, g} and passes it to the AI planner which tries to find a sequence of actions a that lead from the initial state s_0 to the goal state g by invoking operations O.

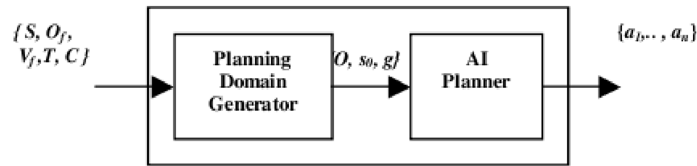


Figure 4.2: The automated planning system (APS) as proposed by Gupta et al. [gup04].

4.1.4 Dingels et al.

Dingels et al. [DFQ07] compared three different planning techniques (namely GraphPlan, UCPOP, ForwardSearch) about their abilities to generate test data. To do this, methods of a Java program are annotated with pre & post conditions using javadoc comments. Then test cases are written and a result is specified as a test goal (i.e. goal: collection_under_test.isEmpty() == true). The pre and postconditions of the methods and the test goal are compiled into the compared planners languages. Then the planners try to find valid method call sequences to reach the desired object state. The sequences are then translated into real JUnit test cases. They conclude that GraphPlan is the most usefull planner of the three compared ones, because of his immense performance. They where capable of generating planning domains containing boolean facts only.

4.2 Comparison to related work

We have shown, that our approach, Planning4ObjectCreation, was able to generate test data for the case studies, the random approach wasn't able to generate. As Howe et al. [vMSDH00, MHvML] and Leitner [Lei04] have also used planners for the generation of test data we will give an overview of the main differences. Table 4.2 gives a direct comparison. Both approaches, Planning4ObjectCreation and the one of Leitner use Design by Contract™-specifications to generate the planning domain D and the planning goal G, whereas Howe et al. build these by hand. Leitner is not able to generate parameters for generated sequence of methods, thus the approach is not able to generate a test sequence that needs parameters. If the planned sequence violates a contract because of too unprecise specifications, Leitner enriches the specification with information gathered during the first execution and restarts the generation process. If our approach fails to generate it either returns an empty object or uses a fall back strategy such as random

generation. Both strategies are not as sophisticated as the one Leitner uses, and therefore constitute an area of further improvement of our system. Dingels et al. [DFQ07] also use a planner for generating test sequences. They are able to do so for boolean predicates only, which are added to the code for the purpose of planning only. We use already existing Design by Contract™ annotations that where originally not designated for test data generation.

	Our Approach	**PlanningExtension** (Leitner)	**System Testing w. AI Planner** (Howe)	**Dingels et al.**
Programming Language	Java with Modern Jass annotations	Eiffel	command line language of a tape device	Java + proprietary annotations
Automatic extraction of the planning domain	✓, using the classes contract	✓, using the classes contract	✗, generated by hand.	✓
Automatic extraction of the planning goal	✓, using the method under test precondition	✓, using the method under test precondition	✗, specified by test engineers.	✓
Automatic translation of the plan to a test	✓	✓	✓	✓
Generation of parameters for generated sequence of methods	✓, by recursively calling the jCAMEL tool	✗	✗	✗
Enrichment of planning domain from test execution	✗	✓	✗	✗
Usable data types	boolean, floating points, integers, object references	boolean, integers with a resultion of 2 bit	¤	boolean

Table 4.1: Comparison of our approach (Planning4ObjectCreation) to those of Leitner [Lei04], Howe et al. [vMSDH00] and Dingels et al [DFQ07]. If a feature is supported by the approach, it is marked with ✓, else with ✗. ¤ marks that no evidence was found in literature.

4.3 Alternative Approaches

This sections cover alternative approaches to generate test data. First of all, we will discuss KORAT [BKM02]. Then we will give an overview of random test data generation strategies in Section 4.3.2. We will discuss *DART* [GKS05], *Directed Automated Random Testing* a system design for randomly generating test data for C programs and *Feedback-directed random test generation* [PLEB07]. Furthermore we will review test data generation using evolutionary approaches in Section 4.3.3.

4.3.1 KORAT

Boyapati et al. introduced KORAT [BKM02]. KORAT is a specification based test data generation framework for Java programs. The authors use method specifications, for instance JML [LC04] annotations (any method specification can be used, as long as a converter is provided) to generate a predicate in terms of a boolean method, which maps to the methods preconditions. Furthermore, a list of boundary values for the tested methods input parameters is needed. Korat generates all non-isomorphic test values between these boundary values, which fulfill the previously generated java predicates. For each isomorph class of input values, the lexically smallest is chosen. To prevent state space explosion when generating new input values, the execution of predicates is being observed. Any fields, that are not read during the execution of the predicate may not be considered any more when generating different input for these predicates. This Method can be referred as *Specification Based Test Data Generation*.

4.3.2 Random Test Data Generation

Generating test data using a random method is quite simple to accomplish and according to Forrester and Miller [FM] has a high chance of finding real software bugs. A problem of generating test data using random generators only is the potential low path-coverage. Many interesting tests may not be generated [Edv99]. To solve that kind of problems, recent approaches tend to use coverage information to guide the generation process. These directed random test data generation methods will be covered in the next section.

Directed Random Test Data Generation

Directed Automated Radmon Testing, *DART* was introduced by Godefroid et al. [GKS05]. *DART* consists of three steps:

1. automatic extraction of the program interface
2. automatic generation of a test driver
3. automatic generation of program test data

We will focus on Step 3 of their approach, the automatic generation of program test data. *DART* was built to generate tests for C-programs. Its generation process uses both symbolic execution and real execution of the *software under test* . First of all, a function to be tested is called with a randomly generated input vector v. During the real execution using this vector, each branching statement is recorded. These branching statements are transformed into symbolic constraints. For each branching statement, a new vector is generated using symbolic execution, so that it will force the branch to take the other, not yet tested, direction. This is done by altering the collected contraints and trying to fulfill them. Consider the following example:

> After executing some C-code, the predicate sequence collected looks as following, where x_0 and y_0 are symbolic variables: $< x_0 \neq y_0, 2*x_0 \neq x_0+10 >$. To let the program execute the path which will take the alternative second branch, the constraint needs to be altered as following: $< x_0 \neq y_0, 2*x_0 = x_0+10 >$. [GKS05]

After solving these constraints, the values are stored and used during the next test run. This leads to higher branch coverage as simple random testing would do.

Feedback Directed Random Test Generation

The approach of Pachecho et al. is called “Feedback-directed random test generation” [PLEB07] and is implemented in the tool RANDOOP. In their approach, they do not only generate test data, they generate complete test-suits for object-oriented programs. A test of an object-oriented program can be seen as a sequence of method calls

with arbitrary inputs, which can either be a primitive value, or an object reference.

To generated such a sequence, a random public method of given classes is selected. Some random chosen methods and values are generated and added to the sequence of calls *seqs*. This sequence is executed with the randomly generated values, and checked against some given contracts. If the sequence does not violate any contract, and has not been generated before, it is saved for further use.

Such contracts may be [PLEB07]:

- A method may not throw a NullPointerException if no input was null
- A method must not throw AssertionErrors

An important part is the generation of the random values needed as parameters:

- If it is a primitive type, it is randomly selected from a previously specified pool of values. These pool is generated by the user.
- If the parameter is a reference type, an object needs to be created. This is done by using either one of the previously generated sequences out of *seqs*, generating a new sequence, or using *null*. Selecting one of these three alternatives is done randomly.

Generating these sequences is done until a specified time limit is reached. After this, a test case is generated out of each sequence.

4.3.3 Test Data Generation using Evolutionary Algorithms

Michael et al. introduced *GADGET* [MMS01], a test data generation tool based on genetic algorithms. *GADGET* uses condition-decision coverage as its testing objectives. Before using *Genetic Algorithms* to generate specific test data, the *software under test* is executed with randomly generated input. After this execution, a so called coverage table is generated, with the state of each condition. This coverage table is used to determine whether a branch was already tested or not. For each requirement the *Genetic Algorithm* is initialized and tries to find a solution. Whenever the *Genetic Algorithm* generates a possible solution which fulfills any of the given requirements the solution is saved for

further use and the coverage table is updated accordingly. The test data generator keeps trying to fulfill any requirement until all requirements are fulfilled or no further solutions can be extracted. By trying to fulfill a given criterion, *GADGET* finds solutions for many other requirements. Michael et al. call this phenomenon "serendipitous satisfaction". For each of the not yet fulfilled testing requirements, a fitness function is generated and the problem of finding a possible solution is mapped to a minimization problem.

Evolutionary algorithms depend on two operators, inspired by biology:

Crossover Crossover takes two data strings, so called genomes. It selectes a random pivot point and concantenates the front part of the first string and the tail of the second string around that pivot point. For example, the two strings **110**01100 and 010**11001** can be crossed by randomly selecting the third position as pivot. The result would be: 11011001. [Mit98]

Mutation Mutation randomly flips one bit inside the data. Taking the above result string as example, 1101100**1**, may be flipped at the last position to result in 1101100**0**. Flipping random bits occurs on each position with a low but equal probability. [Mit98]

When generating test data using evolutionary algorithms, the set of input values can be represented by a genome sequence. Generating a new set of test data now is based on altering the current set of data using the two genetic operators previously descriped, namely crossover and mutation. Selecting a set of test data from a pool of already generated data is done by their fitness, calculated using a specific fitness functions, i.e. the branch coverage achieved by the data set.

Miller et al.[MRZ06] extended the approach introduced by Michael [MMS01] by using Program Dependence Graph Analysis to generate better constraints for the Genetic Algorithm. One problem the authors see at the approach of Michael et al. is that *GADGET* depends on the serendipitous satisfaction phenomenon. It is possible, that *GADGET* may not find input for a specific test requirement. Using fitness functions that rely on the coverage information only, is that sets of boolean values might all have the same fitness, as they all lead to the same branching behaviour. The *Genetic Algorithm* can not prefer one set of values over the other, if they have the same fitness function which makes selecting the potentially better input for mutation undecidable.

Part III

Implementation

5 Implementation

First of all, the theoretical background of infix to prefix conversion will be illustrated in Section 5.1, as it is fundamental for the design of the application. Then, the system design will be shown in Section 5.2. In Section 5.3 we will go into detail how we have implemented the approach introduced in Section 3. We will point out the limitations that have emerged during implementation of the approach in Section 5.4.

5.1 Infix to Prefix

Translating from Modern Jass specifications to PDDL can be seen as translating from an infix to a prefix language.

Converting from infix to prefix notion can be achieved by traversing a syntax tree in depth-first order [GY05]. The Sun Java Compiler API[vdA06] offers an easy and flexible way of accessing the syntax tree of java code, which Modern Jass-annotations are. This can be done using the abstract syntax tree and the corresponding tree visitor, which is explained in detail in Section 5.2.

Conversion works through traversing the tree in depth-first manner and is shown in the example from Listing 5.1. While traversing the tree, the symbol of the current node is written to the output, as shown in Figure 5.1.

```
(x + y) - ((2 * a) + b)
```

Listing 5.1: Example infix expression

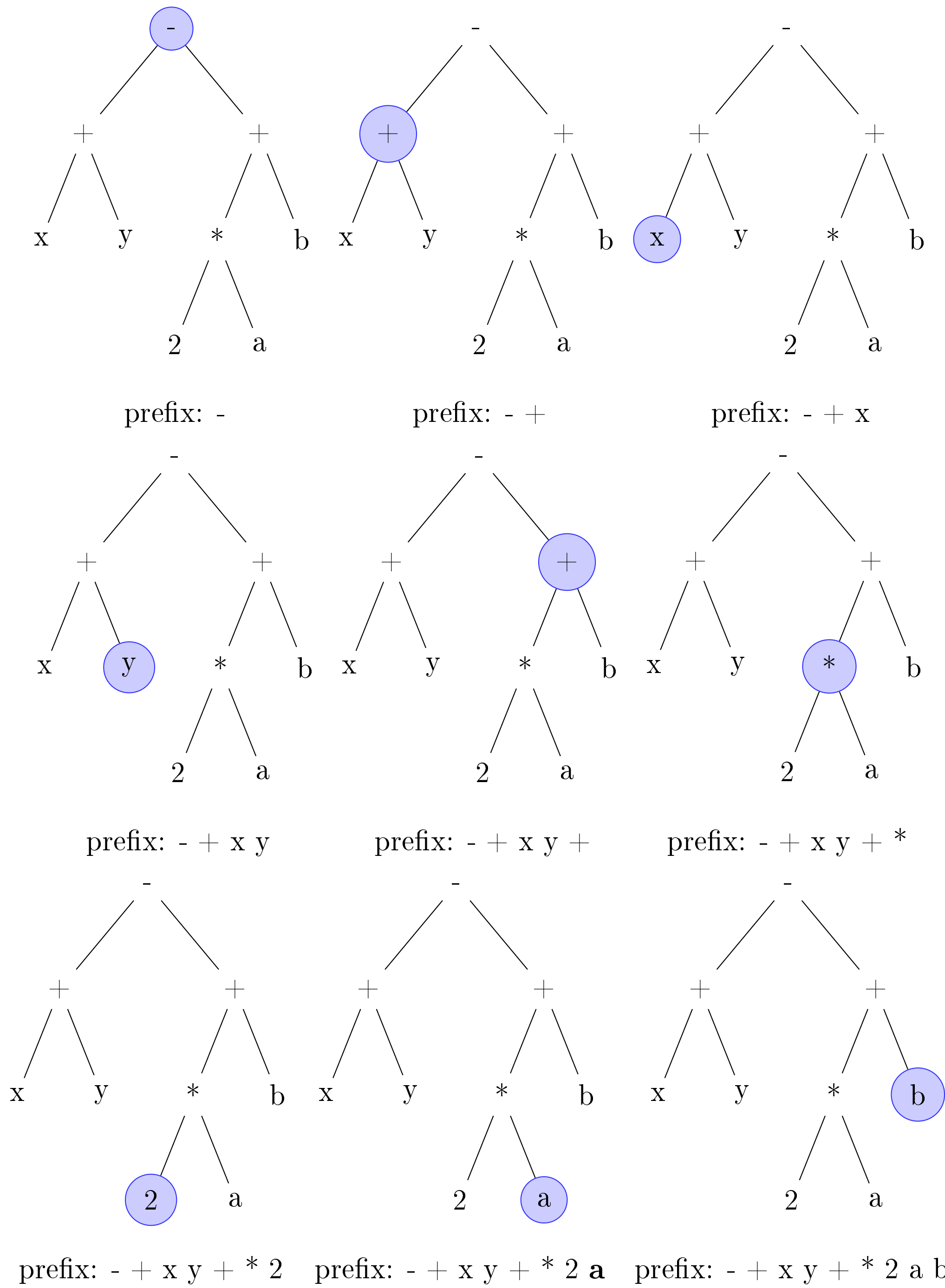


Figure 5.1: This figure illustrates the traversion of the tree representation of the expression: $(x + y) - ((2 * a) + b)$. The tree is traversed in depth first order.

```
- + x y + * 2 a b
```

Listing 5.2: Resulting prefix expression of the infix expression given in Listing 5.1

5.2 Design

This Section contains a description of the implemented classes and their purpose. The system was designed with adaptability in mind. The main advantage is the easy exchangeability and extendability of the used subsystems. Through the three main interfaces, it is possible to completely exchange the underlying planning system. It is possible to use different planners, that even use different planning problem definition languages, such as PDDL or ADL. Figure 5.2 gives an overview of the two main components and their interfaces.

AITypeValueGenerator The AITypeValueGenerator is the main class of this project. It will be called by jCAMEL whenever a parameter is requested that is an instance of a class.

AITypeValueGeneratorConfig The class AITypeValueGeneratorConfig is in charge of all configuration data management needed by the application. Based on external configuration files, it is possible to change the used planning subsystem and the related components.

PlannerFactory The PlannerFactory is responsible for construction the correct Planner and Plan2Denotable classes and filling them with configuration values. For more details on configuration of the system see Section 5.2.1.

Contract2PlanningLanguage This interface is used for the Design by Contract™ to planning language conversion. Different planners may use different planning problem languages, such as PDDL. The implementing class of this interface is responsible for converting the method specifications correctly. The correct Contract2PlanningLanguage implementation is declared in the planners configuration file.

Contract2PDDL This class is in charge of starting the generation process. For each method of the class under test the PDDL-actions will be generated. Figure 5.5 gives an idea how this works.

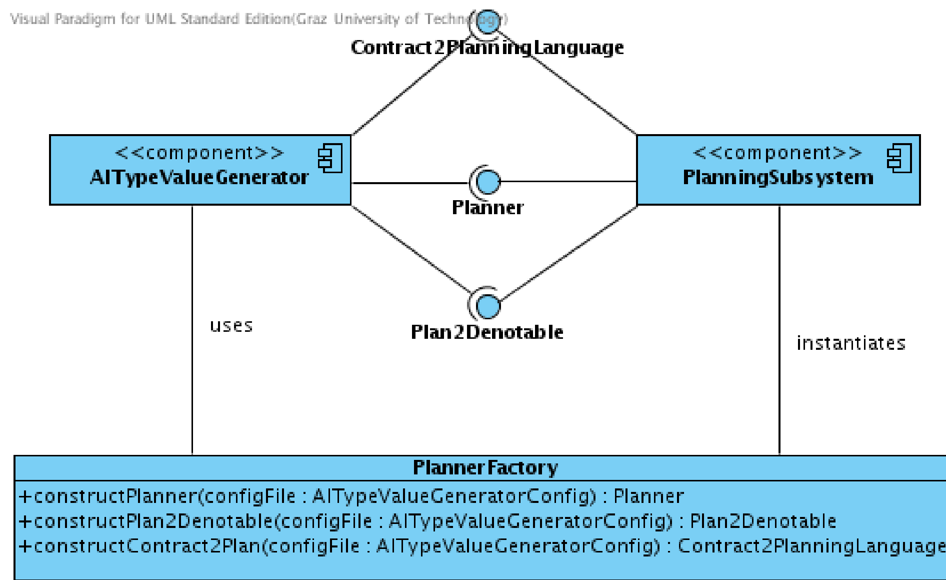


Figure 5.2: Overview of the system. There are three main interfaces: *Contract2PlanningLanguage* is responsible for generation of the planning domain D and the planning goal G that form up the planning problem PP as defined in Definitions 5, 7 and 8. It can be seen as the implementation of the interpretation function of a contract $DbC(C)$ of a class C as given in Definition 14 in Section 3.1 and Section 3.2. The interface *Planner* is in charge of starting the planner and handling its in and output as explained in Section 3.3. *Plan2Denotable* is then used to translate the planners output, the plan P, into the internal code representation of jCAMEL, namely *Denotables*, as it was introduced in Section 3.4.

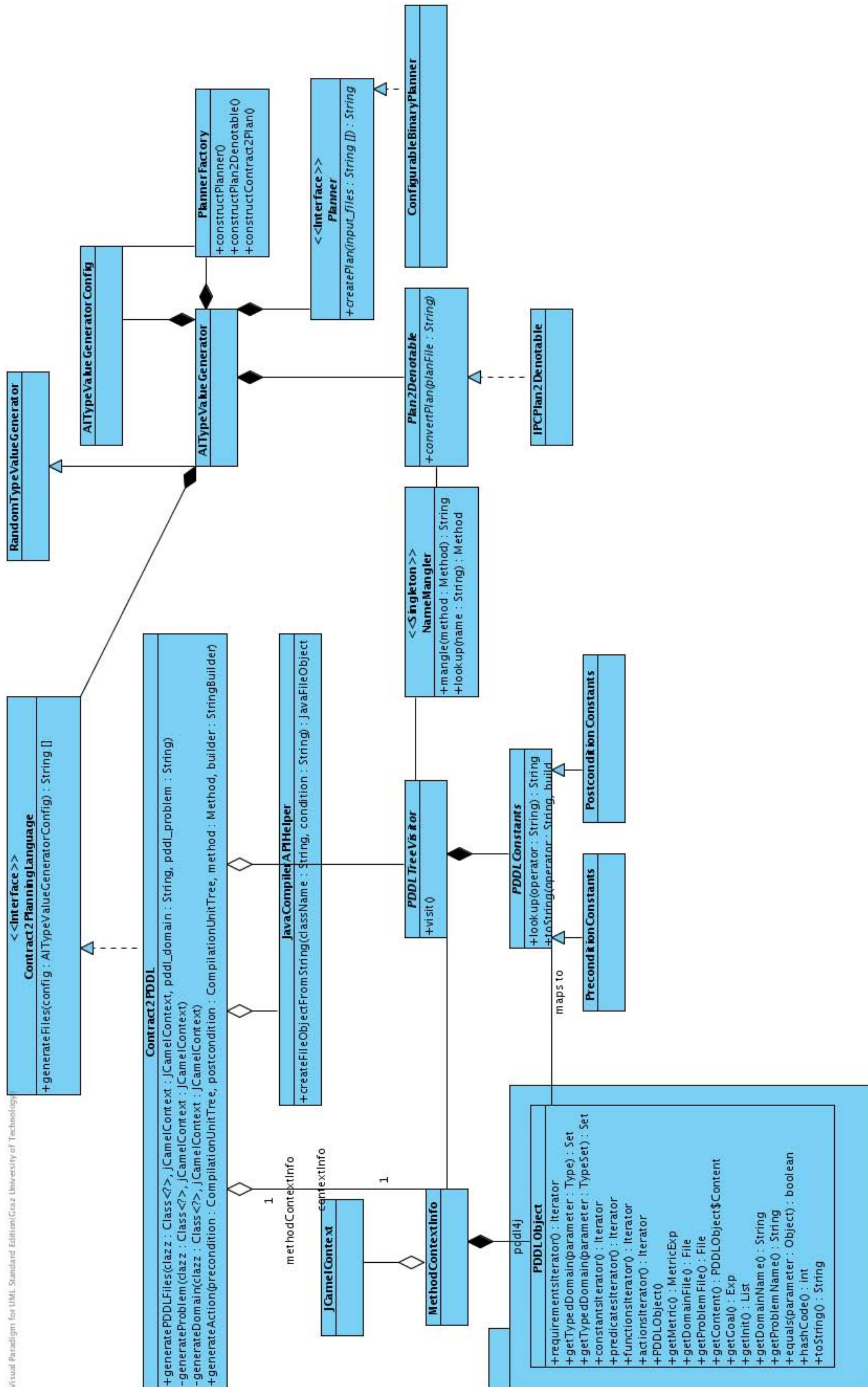


Figure 5.3: Design of the application.

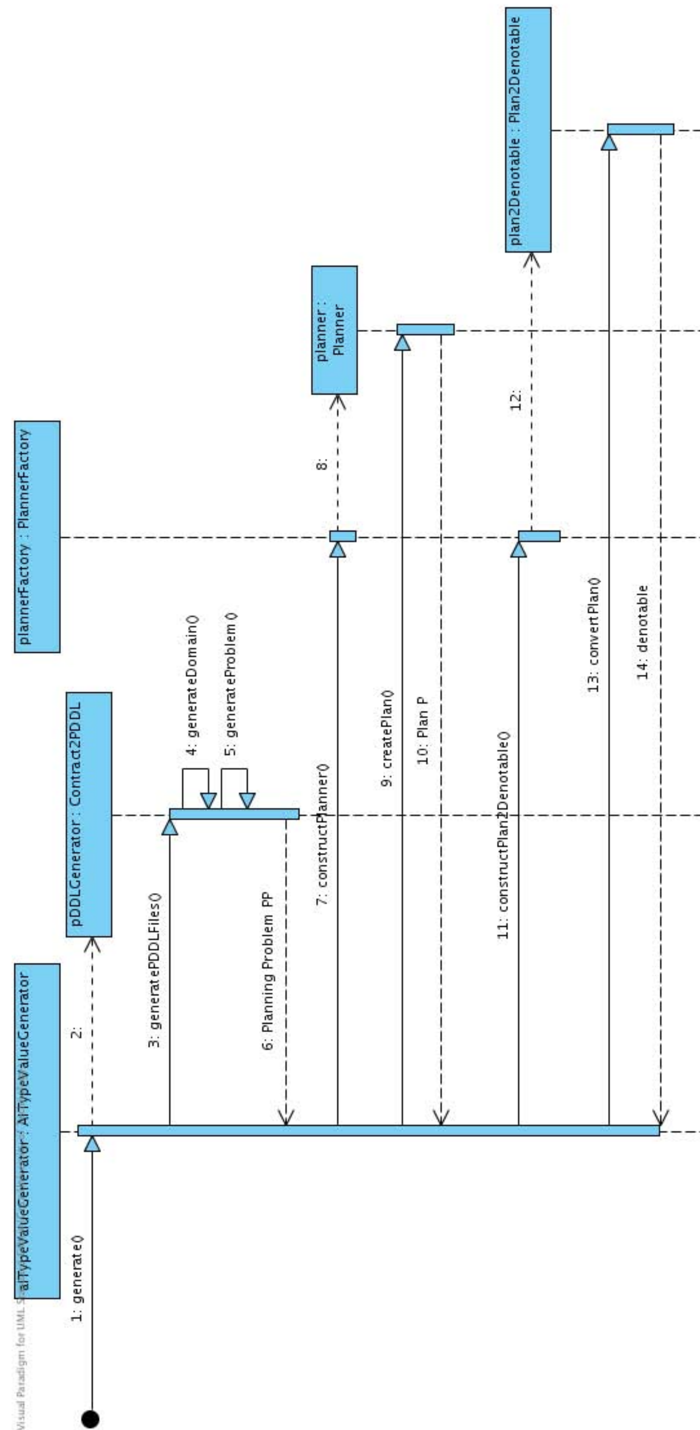


Figure 5.4: Overview of the generation process. At first, an instance of the *Contract2PlanningLanguage*, in this case *Contract2PDDL* is called to generate both the planning domain D and the planning goal G that form up the planning problem PP. Then this planning problem PP is passed to an instance of *Planner*, that will invoke an external planner. The generated plan P is then passed to an instance of the *Plan2Denotable* interface, which translates the plan back to a *Denotable*, the internal data structure of jCAMEL.

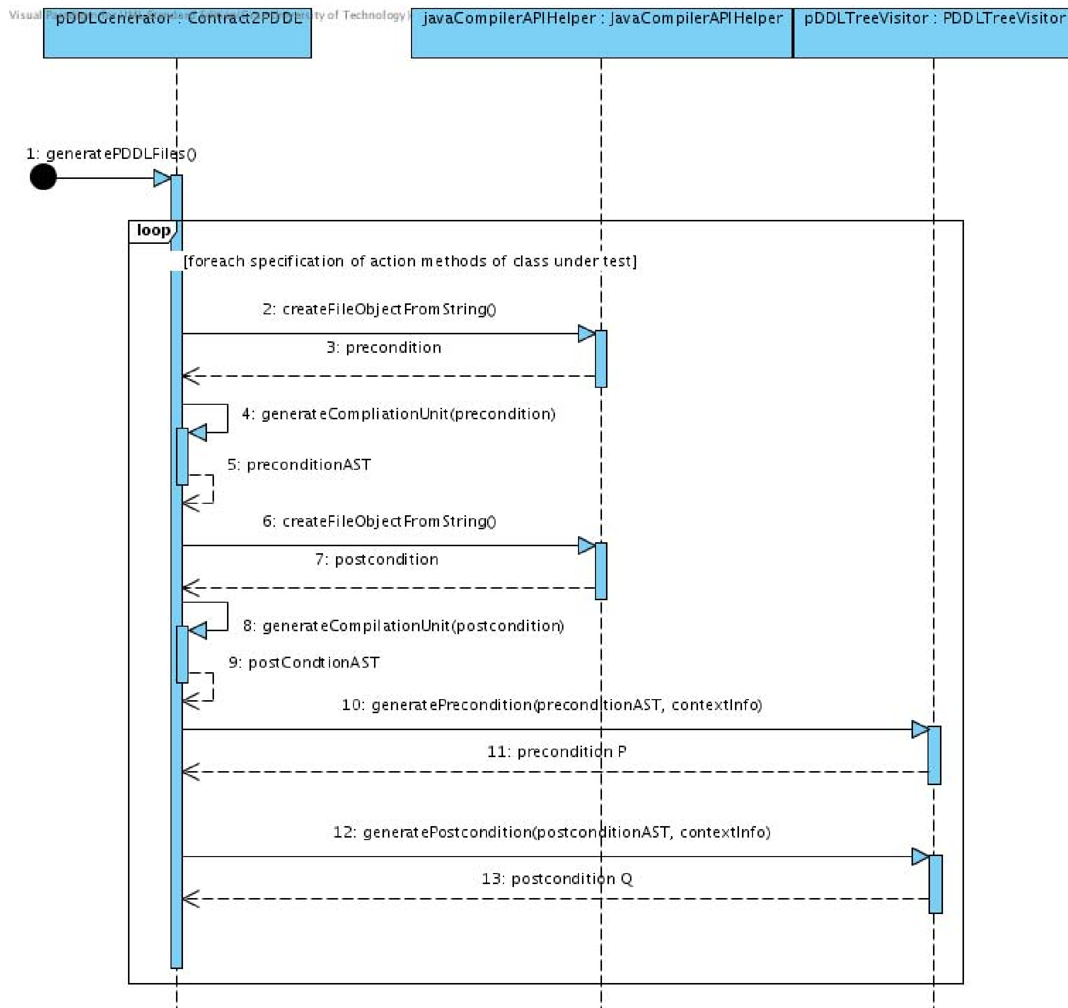


Figure 5.5: Generation of PDDL operators for one class. For each specification *spec* of a classes contract $DbC(C) = \langle mf, inv, mc \rangle$ as defined in 4 an *Abstract Syntax Tree (AST)* is generated from the pre- and postcondition using the Java Compiler API [vdA06] and is then passed to a *TreeVisitor* using the *generatePrecondition(preconditionAST, contextInfo)* and *generatePrecondition(preconditionAST, contextInfo)* methods. The resulting PDDL objects are then serialized to the domain and goal file, to be passed to the planner.

Plan2Denotable This is the abstract base class for parsing the generated plans. The implementation of this class hierarchy is based on the template pattern[FFBS04]. Parsing some elements of the plan file is delegated to the implementing classes, like the *IPCPlan2Denotable* class, which is in charge of handling plans that are compatible with the requirements of the International Planning Competitions [ica].

Planner The planner class is a wrapper for calling a specific planner. One main implementation is provided. The ConfigurableBinaryPlanner can be used to execute any third-party-planner which is available on the system. For details on how to configure the used planner see Section 5.2.1.

PDDLTreeVisitor The PDDLTreeVisitor is responsible for traversing the abstract syntax tree of the methods specification. The AST is generated using the Sun Java Compiler API[vdA06] and is represented in the **CompilationUnitTree** class. While traversing this tree, it will generate the according PDDL-object structure provided by PDDL4J [Pel09] which is then serialized to a file. Together with the PDDL-Constants class, the TreeVisitor acts as the Interpretation Function $\Phi(DbC)$ of the classes contract, as given in Definition 14.

PDDLConstants The PDDLConstants class is a dictionary of PDDL-operator names and their Java equivalents. As PDDL operators differ in pre- and postconditions, there are two implementations of the abstract base, which are in charge of handling the operator conversion correctly. The PDDLConstants class delegate the correct PDDL serialization to the external library PDDL4J [Pel09].

NameMangler The correct mangling of names is essential when generating PDDL-files, as PDDL has a more restricted set of valid characters. This class is used for mangling names of methods and variables into valid PDDL names and can be used as dictionary when reconstructing the Denotable from the generated plans. This class implements the name mangling function $\mu_{out}^{in}(name)$ given in Definition 13. For details on what denotables are, see Section 2.3.1.

MethodContextInfo This class is responsible for holding context information on what is currently being converted. It holds lists of currently seen functions and predicates. This information is used to reduce the number of used functions and predicates. If a defined function is never used within a pre- or postconditions it is not used when generating the PDDL-header. This reduces planning time and file size and helps keeping the files readable.

5.2.1 Configuration

Configuration files are used to make the data generation system more flexible. The following properties can be adapted using the config files. These files are XML-files, as XML is used throughout the rest of the test data generation system.

Global options. The global options are valid for the whole test data generation system. Two main settings can be changed here. The location of the intermediate PDDL-files produced by our system and the used planner. Based on this information the planner specific options can be loaded.

Planner specific options. All information needed for a specific planner are specified here. This are:

- The type of the planner. At most cases this will be the ConfigurableBinary-Planner.
- The type of input the planner is capable to handle. This work focuses on PDDL-based planners. Extending the system to handle alternative planners is possible by implementing the according interfaces.
- Path to the planner binary.
- Name of the planner binary.
- Command line parameters to pass to the planner.
- Type of the output file generated by the planner. This is important, as different planners produce different output. According to this information, the *PlannerFactory* can load the proper *Plan2Denotable* implementation.

5.3 Implementation Details

Im this Section, we will give details on the implementation of the approach as described in Section 3.

5.3.1 Generation of the Planning Domain

Generating the planning domain $D = \{a_1, ..., a_n\}$ means according to Definition 14 of Section 3.1, that we have to generate a planning action a_i for each specification $spec \in ms \in mc$ of a classes Action Methods contract. This is done by using the *PDDLTreeVisitor* class. This class behaves according to the *TreeVisitor Design Pattern* [FFBS04]. Thus, it visits all nodes within an *Abstract Syntax Tree (AST)* in depth first order, which is exactly what we need, as we translate from an infix to a prefix language, as shown in Section 5.1.

When determining the *relevant* parts of a specification according to Definition 15 we compare the occurrence of a State Variable within a part of a specification with the set of currently known State Variables. If it is in the set of known State Variables and does not belong to a parameter of the currently translated specification, the part being examined is considered as relevant. Listing 5.3 shows an example method with a specification consisting of two parts, where only the first one refers to a State Variable, whereas the other part contains a reference to the parameter's State Variables which are not relevant for the generation of the planning domain. Thus, the selection of a method like in Listing 5.3 or a member field that starts either with nothing or the keyword **this** is considered to be relevant, and is added to the set of currently known State Variables. Thus, we dynamically expand the set of known State Variables during the translation.

Name Mangling

In Section 3.1 we have pointed out that the subset of available characters in PDDL is significantly smaller then the one available to Java. Thus we have introduced a name mangling function $\mu_{out}^{in}(name)$ in Definition 13, that maps from a method's name in Java to a valid name in PDDL. This functionality is implented within the class *NameMan-*

```
@Pre("this.size() > 0 && other.size() > 0")
public void switch(Stack other)
```

Listing 5.3: An example method with a specification consisting of two parts, where only the first one refers to a State Variable, whereas the other part contains a reference to the parameter's State Variable. We assume that the class containing these method has a State Variable called $size()$. The parameter used here is the *Stack* from Listing 2.1.

gler. Table 5.3.1 gives an overview of unsupported characters in PDDL an how we map them to valid names. The *NameMangler* class offers two method, *mangleMember(String memberName, int spec_counter)* and *lookup(String actionName)*.

The method *mangleMember(Member member, int spec_counter)* is the implementation of the mangling function $\mu_{pddl}^{java}(name)$. The first parameter is an instance of the an Action Method $am \in AM(C)$ of the current class C, which is used to generate the planning domain. The second parameter is used to determine which specification $spec \in ms \in mc$ of the classes contract $DbC(C) = \langle mf, inv, mc\rangle$ of a method is translated. As an Action Method might have more than one specification, this reference is necessary. Details on that are given in Section 2.2, where we describe the possibility of Design by Contract™ systems to have more then one specification for a method. At first the members name is translated according to Definition13. The name consists of the name of the member and the names of all parameters. If it is a constructor it also includes the classes name and package. Within this name, all occurring invalid characters are replaced by their translation according to Table 5.3.1. Then the specification counter is appended to the resulting name. If the method has no parameters, the brackets are not translated, as they don't carry any information.

When mangling a members name, the member, the used specification *spec* and its assigned name is stored within the *NameMangler* class for later retrieval by the function *lookup(String actionName)*.

Java	Occurrence	Translation
Opening and Closing Brackes: (,)	Used in Method and Constructor names to give the list of arguments.	Replaced by an underscore: _
The dollar sign: $	Used as generic delimiter in method names	Replaced by three underscores: ___
Dots: .	In method and constructor names it is used as package seperator.	Replaced by an underscore: _

Table 5.1: Unsupported characters in PDDL and their translation from Java.

5.3.2 Generation of the Planning Goal

The determination of *relevant* parts of specifications for the generation of the planning goal G is done in opposite of the determination when generating the domain D. Thus, we consider only those parts of specifications to be relevant, that contain State Variables of parameters. Using Listing 5.3 as example, the second part of the specification: $\mathbf{other.size()} > \mathbf{0}$ would be relevant when generating in instance of the parameter named **other**.

5.3.3 Execution of the Planner

Both the planning domain D and the planning goal G that form up the planning problem PP are held within the object structure of PDDL4J. They are then serialized to two files, the domain file containing the planning domain, and the goal file. These files, and the desired plan output file are passed to the planner binary. The exact call to the planner binary is configured through the configuration interface given in Section 5.2.1.

If the planner returns a valid plan, which is determined by its return value, the plan file is passed to an instance of the *Plan2Denotable* interface, which will now generate a *Denotable*.

If no valid plan was returned, the generation process is stopped and the configured fallback strategy is used. There are two possibilities: returning a Denotable containing null, or using a different generation strategy.

5.3.4 Generation of Java Code from a Plan

As described in Section 5.3.1 we use the class *NameMangler* as the implementation of the mangling function $\mu_{out}^{in}(name)$ to do the mapping between PDDL action names and Java method names and vice versa, according to Definition 13. If the invocation of the planner was successful, a valid plan is returned.

An implementation of the interface *Plan2Denotable*, in our case *IPCPlan2Denotable*, is now in charge of opening the plan file and translating it into a *Denotable* as described in Section 3.4. Therefore, we parse the file for actions. Every action is looked up within the name mangling class, using the method *lookup(String actionName)*. The lookup method returns the associated class' member and the specification *spec* that was used to generate the according planning domain.

A Denotable is instantiated with the looked up member. If there is already a sequence of Denotables, this sequence is added as preamble to the current Denotable. If the method or constructor invocation we have retranslated from the plan needs a parameter to be called with, our system calls the jCAMEL tool to retrieve a set of arguments, that fulfill the current members specification *spec*. If one of the parameters needed to call the member is a non-primitive type, jCAMEL will choose our approach to generate a valid object. This leads to recursive invocation of the AITypeValueGenerator. If a maximum number of recursive invocations has occurred, the system stops generating parameters in order to prevent endless recursion. The thereby generated parameters are added to the Denotable. Then the next action is retrieved from the plan.

After the last action is retranslated, the Denotable is returned to the calling jCAMEL instance, which now generates the test for the method under test with the currently created object as parameter.

Example 27: A returned plan and its Denotable.
Listing 5.4 shows a returned plan of getting our *Stack* example to have 2 elements. The plans first action is associated with the constructor call, and then there are two actions that belong to the method *push(int)*. When reading the first line, *IPCPlan2Denotable* will call *lookup("STACK")* of the *NameMangler* class which will return the according constructor of the class *Stack*, and it's specification. The class *Stack* was defined as $C(Stack) = \langle\{Stack()\}, \{push(int), pop(), isEmpty(), peek(), size()\}, \{size_\}\rangle$ in Ex-

ample 2 with it's contract being

$$
\begin{aligned}
&DbC(Stack) = \langle \{mSize\}, \{size_ >= 0\},\\
&\quad \{\\
&\quad\ mc_{Stack} = \ \langle \{\langle true, mSize == 0\rangle\}, \emptyset\rangle,\\
&\quad\ mc_{pop} = \ \langle \{\langle size > 0, size == @Old(size) - 1\rangle\}, \emptyset\rangle,\\
&\quad\ mc_{push} = \ \langle \{\langle true, size == @Old(size) + 1\rangle\}, \emptyset\rangle,\\
&\quad\ mc_{peek} = \ \langle \{\langle true, size == @Old(size)\rangle\}, \emptyset\rangle,\\
&\quad\ mc_{size} = \ \langle \{\langle true, @Return == mSize\rangle\}, pure\rangle\\
&\quad\ mc_{isEmpty} = \ \langle \{\langle size > 0, @Return == false\rangle, \langle size == 0, @Return == true\rangle\}, pure\rangle\\
&\quad \}\rangle
\end{aligned}
$$

As the constructors precondition is only *true* no parameter needs to be generated to be able to call the constructor. Thus the current created Denotable D_1 contains only the constructor invocation. The next call is: *lookup("PUSH_INT")*, which returns the according Action Method $push(int)$ and it's contract $mc_{push} = \langle \{\langle true, size == @Old(size, int) + 1\rangle\}, \emptyset\}\rangle$. A new Denotable D_2 is generated with the invocation if push as it's parameter. Furthermore, jCAMEL is called to retrieve an integer that fulfills the specification of $push(int)$. As no specification limits the parameter, an integer within all possible integer values is returned. The previous Denotable D_1 is added as preamble to the current Denotable. The same applies to the second invocation of $push(int)$. Figure 5.6 shows the generated Denotable of this example. We assume, that jCAMEL has returned -3 and 2 as requested integers. The last generated Denotable, D_3 is returned to jCAMEL. □

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

Listing 5.4: An IPC-conform plan that calls three actions. The first relates to the constructor of the Stack example class given in Listing 2.1. The following two are associated with the method *push*(*int*). The first number is the time step the planner will invoke the action, the number at the end the costs the planner estimates for calling this action. Both features are currently not used, but we will give an overview of potential usage of them in Section 7. The name surrounded by the brackets is name of the used action.

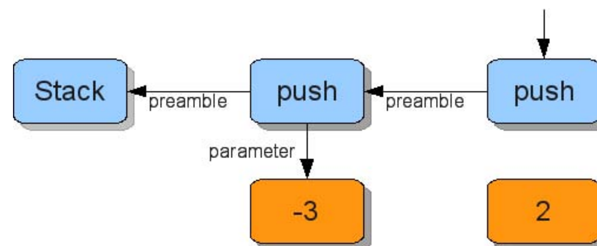


Figure 5.6: The Denotable generated from the plan given in Listing 5.4. We assume, that jCAMEL has returned -3 and 2 as requested integers.

5.4 Limitations and Assumptions

During development of the tool presented in this thesis, some limitations have emerged.

First of all, no planner that was submitted to the last years planning competition was able to handle domains that contained effects generated from multiplications and divisions. This is due to the fact, that using multiplications and divisions let the search-space become non-linear, which these planners where not developed for. Thus, we are not able to create objects that need either a multiplication or a division within one of their Action Methods postcondition to reach a requested target state. Actually none of our case studies presented in Section 6 contained such a postcondition.

Additionally, we have made some assumptions on the used Modern Jass annotations. In the translation step, our implementation does not support the use of boolean return values as part of a specification unless it is compared with a boolean value. Thus a specification telling that a parameter of the class Stack from Listing 2.1 named stack can not be empty is not allowed to be written as: **!stack.isEmpty()**. It has either to be written as **stack.isEmpty() != true** or as **stack.isEmpty() == false**.

Moreover, when specifying an arithmetic operation, the State Variable beeing assigned needs to be on the left side of the comparison. Thus it is possible to specify: **x() == @Old(x(),int) + 1** but not **@Old(x(),int) + 1 == x()**. Also this sort of statements did not occur in the used case studies.

Part IV

Results

6 Empirical Evaluation

In this section the implemented system is tested against chosen case studies. To show the differences of the implemented planner based approach to the already existing random data generator the generated tests are compared to those generated by the random approach which is described in Section 2.3.

All experiments where carried out on the same machine (Intel®Core™2 Quad CPU Q9400, 2.66GHz and 4GB RAM) and using the same software configuration. Each approach was executed using the same parameter configuration:

Mutating probability The mutating probability defines whether or not, the random approach should continue calling random methods on an object. The value used was **0.5**.

Attempts This parameter defines how many tests for a single method should be generated. We have used one attempt during our experiments.

Planner This parameter is used to set the planner the AITypeValueGenerator uses. During the test series we have used **sgplan6** [HWHC06].

Fallback The used fallback approach, when the planning step fails. We have chosen to return a Denotable containing *null*, thus stating that no instance was generated.

After execution of the test generator, the resulting tests are classified into the following categories:

succeeding If a testcase succeeds, both the precondition and the postcondition of the method under test are satisfied.

failing Failing tests do satisfy the precondition of the method under test, but violate the postcondition. These tests might reveak ab implementation fault.

meaningless Meaningless tests do not satisfy the precondition of the method under test. Thus the input parameters for the method under test are not correctly generated.

First of all, we will compare our approach using the StackCalc case study, a stack based calculator in Section 6.1. In addition we will take a look at a real world application in Section 6.2.

We will compare our results both using the number of successful generated tests (those tests that are not meaningless), and the achieved line and branch coverage. We will examine how may lines and branches of the *software under test* where covered by the generated tests. We both give the average and the standard deviation of the according metrics. Both approaches are given one chance to generate parameters for a method under test. Thus, the number of generated tests reflects the amount of methods an approach is able to generate parameters for.

6.1 StackCalc

The first case study used to compare our results was an implementation of a stack based calculator. The case study contains 43 classes, most of them beeing operator implementations that need a stack in some specific configuration to operate on. Table 6.1 contains the amount of Modern Jass specifications occuring in the case study.

Specification	Amount
@Pre	40
@Post	21
@SpecCase	44
@Pure	4
@Also	15
@Helper	3

Table 6.1: Amount of Modern Jass annotations in the StackCalc implementation [GMW10].

This case study contains 34 methods within 28 classes that contain non-primitive parameters. 15 of these parameters refer to a stack beeing in a specific state. The whole case study contains 109 methods. We have been able to generate tests for each of these methods. Table 6.2 gives the amount and classification of the generated test, whereas

Metric	AI	Random
Avg. Succ.	83.40	56.20
Dev. Succ	5.24	0.98
Avg. Fail.	7.40	4.00
Dev. Fail.	5.24	1.26
Avg. Meaningless	17.80	49.60
Dev. Meaningless	2.40	1.02

Table 6.2: Number of generated tests within the *StackCalc* case study. The amount of meaningless tests drops significantly, when the planner based approach is used.

Table 6.3 shows the achieved coverage for the whole case study. The achieved function coverage, thus tests that were not meaningless, increases by about **50%**, leading to an increased line coverage of **25%**.

The numbers fluctuate as some primitive parameters are generated with different approaches, such as random, that might not always return valid data. We thus ran 100 iterations of each generation algorithm, to balance the results.

For a second series of experiments, we have limited the generated tests to the implementation of operators to be able to directly compare the two approaches. The results show, that our approach is able to generate tests for each operator, which the random strategy can not. An operators specification requires a stack having some elements to operate on. Table 6.4 shows the resulting test classification, and Table 6.5 gives the achieved coverage.

Metric	AI	Random
Avg. Line Coverage	0.71	0.57
Dev. Line Coverage	0.02	0.01
Avg. Branch Coverage	0.34	0.24
Dev. Branch Coverage	0.06	0.03

Table 6.3: Line and branch coverage of the generated tests of the *StackCalc* case study. The achieved line coverage increases by about **25%** by using our proposed planner based strategy.

Metric	AI	Random
Avg. Succ.	13.47	1.43
Dev. Succ	0.50	0.78
Avg. Fail.	2.16	0.07
Dev. Fail.	0.50	0.25
Avg. Meaningless	0.00	14.11
Dev. Meaningless	0.00	0.89

Table 6.4: Number of generated tests for operators only of the *StackCalc* case study. The number of meaningfull tests shows that the random approach is not able to generate the required parameters sufficiently, as the planner based approach does. 15 methods have been tested.

Metric	AI	Random
Avg. Line Coverage	0.21	0.04
Dev. Line Coverage	0.00	0.01
Avg. Branch Coverage	0.06	0.00
Dev. Branch Coverage	0.00	0.01

Table 6.5: Line and branch coverage of the generated tests for operators only of the *StackCalc* case study.

6.2 BillingSoftware

The *BillingSoftware* application is part of a mobile phone billing application, developed by an international telecommunications infrastructure company.

The case study consists of 41 classes, with a total of 249 methods (methods of the classes and their base classes). Additionaly, it uses six libraries, where no specifications are available. The case study contains 71 methods that need an object as parameter.

The jCAMEL tool is designed to generate tests within one thread. We had to exchange one class (an implementation of a blocking queue) of the case study, as it caused a deadlock, when used within a single-threaded environment. We have therefor created a new implementation of the used queue interface, that does not block a single threaded application.

Furthermore, we had to exchange a parser class that was inside of one external library. It is in charge of parsing the incoming billing messages, that are received over the network.

It then returns a message object. As these messages are received as byte arrays and there is no specification available how they have to look like, we have chosen to provide an implementation that will return valid message objects. If there is a specification for the parsers byte array input, the according input value generator of jCAMEL would provide valid data.

Both of these changes are available to both test data generation strategies, and thus do not distort the results.

Table 6.6 shows the number of generated tests. Our approach is able to generate 59.07 more tests on average, that are not meaningless, compared to the random strategy. Thus, the generated test data satisfies the precondition of the method under test. This leads to significantly higher line coverage, as shown in Table 6.7. The line coverage doubled, by using our planning based strategy. The branch coverage does not increase significantly, as the additionally tested code contains only branches, that do input checking.

The most complex parameter requested, needed a total of **30** method invocations to achieve the target state. Each method invocation need a complex parameter as well. The random approach on average fails after the first generated method invocation. Thus, our approach is able to generate parameters with high configuration complexity.

Table 6.8 shows the average generation time for both case studies in seconds. Row “per succ.” gives the average time required to generate a non-meaningless test. Planning domains and the resulting plans within the *StackCalc* case study are relatively small. Hence the generation time does not differ significantly from the random strategy. For the *BillingSoftware* case study, the generated planning domains are complex. The long sequence of method invocations our approach is able to handle leads to increased planning time. With respect to successfully generated tests, the overhead caused by using a planner decreases.

Metric	AI	Random
Avg. Succ.	185.53	126.97
Dev. Succ	2.97	2.21
Avg. Fail.	6.13	5.62
Dev. Fail.	2.97	0.66
Avg. Meaningless	51.34	110.41
Dev. Meaningless	2.98	1.98
Exceptional	6	6

Table 6.6: Number of generated tests of the *BillingSoftware* case study. Using the planner based strategy, the amount of meaningless test drops significantly.

Metric	AI	Random
Avg. Line Coverage	0.40	0.20
Dev. Line Coverage	0.02	0.01
Avg. Branch Coverage	0.18	0.16
Dev. Branch Coverage	0.03	0.01

Table 6.7: Line and branch coverage of the generated tests of the *BillingSoftware* case study. Because of the higher amount of succeeding tests generated, the average line coverage over all 100 test series doubles. The branch coverage does not increase significantly. This is due the fact, that the additionally tested code contains only branches, that check whether the input was correct, which it always will, if the precondition holds.

		AI	Random
StackCalc	overall	162.07	107.02
	per succ.	1.78	1.77
StreamingFeeder	overall	495.96	166.93
	per succ.	2.59	1.25

Table 6.8: Average test generation time in seconds over all 100 experiment iterations.

7 Conclusion

Our presented test data generation strategy transforms Design by Contract™ annotations into a planning domain. We have shown that this technique is qualified for automatic test data generation. Testing a method requires instantiating objects as parameters, that have to comply with the methods precondition. If that specification forces the parameter to be in a very specific state, a random generation will likely fail. Thus using the contract of the parameters type, and the precondition of the method under test to generate a planning problem has exposed to be suitable for generating these parameters. The sequence generated by a planner reflects the series of method invocations that will bring the object into the required state.

The results show, that our approach is highly applicable to the problem of generating parameters that must satisfy strong preconditions. It clearly outperforms the random strategy in both function and line coverage. For the *StackCalc* case study, our approach was able to improve the function coverage by about **50%** leading to an improved line coverage of about **25%**. For the *BillingSoftware* case study, the function coverage improved about **45%**, leading to a line coverage improvement of **100%**, as methods with long bodies became testable.

Analysis of the generated test has shown, that the planner based strategy performs very good for specific categories of classes, and poorly for others. Our approach is applicable for all kinds of data structures, such as lists or stacks, as their specifications fit perfecty to planning. Furthermore, classes that have a high amount of getter and setter methods, and are therefore highly customizationable, are good targets for planning.

Our approach does not fit to classes that deal with high level calculations, such as math libraries. Through its dependence on the presence of Design by Contract™ annotations, methods that use parameters from external libraries where no specifications are available

can cause troubles. Our approach will instantiate such an external object, but will not invoke any actions. If the method under test has a precondition containing expectations over that parameter, the generation likely will fail. This also applies to external objects that have no specifications but highly depend on their parameters state. Thus, an automatic reasoning over the non-available specifications, as Leitner [Lei04] does, might be a promising field of further research.

7.1 Further Research

In this thesis, we have shown that the developed approach works. There are further issues one can focus on.

A possible field of further research, is using the possibility of adding time and cost constraints to the planning domain. When adding costs to actions, they could be assigned proportional to the amount of parameters of the method, that the action was generated from. Adding execution time information to the action could be done using heuristics determining how long a method potentially needs to execute. Doing this, would enable user of the planning system to decide whether to generate simple, cost effective instances of a requested object, or preferring sequences that would speed up the test execution. Using cost effective action sequences would speed up the data generation time, as less parameters would need to be generated.

If an error occured during generation of the planning domain, or the planner is not able to come up with a valid sequence of actions, better failover capabilities could be implemented. Currently we return a Denotable containing null as the generated value. A strategy like Leitner [Lei04] has chosen, using runtime information to enrich the specifications, would be promising.

Another point that could be looked at, is the support of Modern Jass annotations we currently can not parse, due to the assumptions made in Section 5.4.

Furthermore a comparison of different planning paradigms, according to Dingels et al. [DFQ07] would be promising.

The most challenging task we will focus on, is to support all Design by Contract™ postconditions, not alone those with a calculation description. Therefore, we have to develop a new planning system since this limitation is based on the usage of PDDL. A SMT Solver could be used for evaluation of the postconditions. This would enable the planning system to convert postconditions as “@Return == x() % 2”, which is no direct calculation rule, but describes the state after executing the annotated method.

Bibliography

[BKM02] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. Korat: automated testing based on java predicates. In *ISSTA '02: Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, volume 27, pages 123–133, New York, NY, USA, July 2002. ACM Press.

[Ciu04] Ilinca Ciupa. Teststudio: An environment for automatic test generation based on design by contract. Master's thesis, ETH Zurich, July 2004.

[CLSE05] Yoonsik Cheon, Gary Leavens, Murali Sitaraman, and Stephen Edwards. Model variables: cleanly supporting abstraction in design by contract: Research articles. *Softw. Pract. Exper.*, 35(6):583–599, May 2005.

[CRM07] Yoonsik Cheon and Carlos E. Rubio-Medrano. 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, 2007.

[DFQ07] Eddie Dingels, Timothy Fraser, and Alexander Quinn. Generating java unit tests with ai planning. In *WEASELTech '07: Proceedings of the 1st ACM international workshop on Empirical assessment of software engineering languages and technologies*, pages 2–6, New York, NY, USA, 2007. ACM.

[dMB08] Leonardo de Moura and Nikolaj Bjørner. *Z3: An Efficient SMT Solver*, volume 4963/2008 of *Lecture Notes in Computer Science*, chapter 24, pages 337–340. Springer Berlin, April 2008.

[Edv99] Jon Edvardsson. A survey on automatic test data generation. In *Proceedings of the Second Conference on Computer Science and Engineering in Linköping*, pages 21–28. ECSEL, October 1999.

[FFBS04] Elisabeth Freeman, Eric Freeman, Bert Bates, and Kathy Sierra. *Head First Design Patterns*. O'Reilly Media, 1 edition, October 2004.

[FM] Justin E. Forrester and Barton P. Miller. An empirical study of the robustness of windows nt applications using random testing. pages 59–68.

[FN71] Richard E. Fikes and Nils J. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3-4):189–208, 1971.

[GFB+07] Manish Gupta, Jicheng Fu, Farokh Bastani, Latifur Khan, and Yen. Rapid goal-oriented automated software testing using mea-graph planning. *Software Quality Journal*, 15(3):241–263, 2007.

[GIP+98] Malik Ghallab, Craig K. Isi, Scott Penberthy, David E. Smith, Ying Sun, and Daniel Weld. Pddl - the planning domain definition language. Technical report, CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control, 1998.

[GKS05] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: directed automated random testing. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, volume 40, pages 213–223, New York, NY, USA, June 2005. ACM Press.

[GMW10] Stefan Galler, Andreas Maller, and Franz Wotawa. Automatically extracting mock object behavior from design-by-contract specification for test data generation. In *AST 2010*, May 2010.

[GPW08] Stefan J. Galler, Bernhard Peischl, and Franz Wotawa. Challenging automatic test case generation tools with real world applications. In *Software Engineering and Applications 2008*, November 2008.

[Gre04] Nicole Greber. Test wizard: Automatic test generation based on design by contract. Master's thesis, ETH Zurich, 2004.

[gup04] *Automated test data generation using MEA-graph planning*, 2004.

[GWW10] Stefan J. Galler, Martin Weiglhofer, and Franz Wotawa. Synthesize it: from design by contract to meaningful test input data. 2010.

[GY05] Jonathan L. Gross and Jay Yellen. *Graph Theory and Its Applications, Second Edition (Discrete Mathematics and Its Applications)*. Chapman & Hall/CRC, September 2005.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, October 1969.

[HWHC06] C. W. Hsu, B. W. Wah, R. Huang, and Y. X. Chen. New features in sgplan for handling soft constraints and goal preferences in pddl3.0. In *Proc. Fifth International Planning Competition*. International Conf. on Automated Planning and Scheduling, June 2006.

[ica] Icaps conference. Available online at `http://ipc.icaps-conference.org`, last accessed: 10/04/10.

[Jen03] Rune M. Jensen. Efficient bdd-based planning for non-deterministic fault-tolerant, and adversarial domains, 2003.

[jet] Jet implementation documentation. Available online at: `http://opuntia.cs.utep.edu/ssv/et/javadocs/overview-summary.html`.

[LC04] Gary T. Leavens and Yoonsik Cheon. Design by contract with jml, 2004.

[Lei04] Andreas Leitner. Strategies to automatically test eiffel programs. Master's thesis, Graz University of Technology, Institute for Softwaretechnology, December 2004.

[Mey87] B. Meyer. Eiffel: programming for reusability and extendibility. *SIGPLAN Not.*, 22(2):85–94, 1987.

[Mey92] B. Meyer. Applying 'design by contract'. *Computer*, 25(10):40–51, 1992.

[MFC01] Tim Mackinnon, Steve Freeman, and Philip Craig. Endo-testing: unit testing with mock objects. pages 287–301, 2001.

[MH69] John Mccarthy and Patrick J. Hayes. Some philosophical problems from the standpoint of artificial intelligence. In *Machine Intelligence*, volume 4, pages 463–502, 1969.

[MHvML] Richard T. Mraz, Adele Howe, Anneliese von Mayrhauser, and Li Li. System testing with an ai planner.

[Mit98] Melanie Mitchell. *An Introduction to Genetic Algorithms (Complex Adaptive Systems)*. The MIT Press, February 1998.

[MMS01] C. C. Michael, G. Mcgraw, and M. A. Schatz. Generating software test data by evolution. *Software Engineering, IEEE Transactions on*, 27(12):1085–1110, 2001.

[MRZ06] James Miller, Marek Reformat, and Howard Zhang. Automatic test data generation using genetic algorithm and program dependence graphs. *Information and Software Technology*, 48(7):586–605, July 2006.

[Pel09] Damien Pellier. Pddl4j. `https://sourceforge.net/projects/pdd4j`, May 2009.

[PLEB07] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. Feedback-directed random test generation. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, pages 75–84, Washington, DC, USA, 2007. IEEE Computer Society.

[PW92] Scott J. Penberthy and Daniel S. Weld. Ucpop: A sound, complete, partial order planner for ADL. In Bernhard Nebel, Charles Rich, and William Swartout, editors, *KR'92. Principles of Knowledge Representation and Reasoning: Proceedings of the Third International Conference*, pages 103–114. Morgan Kaufmann, San Mateo, California, 1992.

[Rie07] Johannes Rieken. Design by contract for java - revised. Master's thesis, Department für Informatik, Universität Oldenburg, April 2007.

[RN02] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach.* Prentice Hall, second edition, December 2002.

[Tas02] G. Tassey. The economic impacts of inadequate infrastructure for software testing. Technical report, National Institute of Standards and Technology, 2002.

[vdA06] Peter von der Ahe. Jsr 199: Java compiler api. `http://jcp.org/en/jsr/detail?id=199`, December 2006.

[vMSDH00] Anneliese von Mayrhauser, Michael Scheetz, Eric Dahlman, and Adele E. Howe. Planner based error recovery testing. In *ISSRE '00: Proceedings of the 11th International Symposium on Software Reliability Engineering (ISSRE'00)*, Washington, DC, USA, 2000. IEEE Computer Society.