JOHANNES KEPLER
UNIVERSITÄT LINZ | JKU

Technisch-Naturwissenschaftliche
Fakultät

Instant Incremental Class Invariant Checking for Java Virtual Machines

MASTERARBEIT

zur Erlangung des akademischen Grades

Diplomingenieur

im Masterstudium

Software Engineering

Eingereicht von:
Sebastian Wilms

Angefertigt am:
Institute for Systems Engineering and Automation

Beurteilung:
Univ.-Prof. Dr. Alexander Egyed, M.Sc.

Mitwirkung:
DI(FH) Dr. Alexander Nöhrer

Linz, July 2013

Abstract

Software systems fail. Although software developers mostly try their very best to avoid errors in software systems, failures are unavoidable. Sometimes the failures are catastrophic, resulting in financial loss or even danger to human safety. Often the faulty behavior leads to the system being in an invalid state, which may cause follow-up errors later on. Yet, invalid states of a system can be detected using class invariants. Class invariants express the valid states of a software system and must not be violated. Through exhaustive testing, these invariants can then be tested to ensure that they hold. Invariant checking may thus avoid failure.

Unfortunately, existing approaches for invariant checking require extensive manual intervention in where/how to place the invariants in the code. Invariants only need to be validated in situations where its validity is affected. This requires careful placement of invariants checks in the code – a placement that needs to be reconsidered every time the code changes or the invariant changes. The only alternative is to validate the invariants continuously at every step of the software's execution. However, doing so is unscalable.

This thesis presents a novel, tool-supported approach for invariant checking that automatically decides when to validate invariants in a manner that is significantly more scalable compared to placing invariants everywhere and in a manner that is guaranteed correct.

Apart from defining the invariant, no additional user interaction is required. The approach avoids the invariant placement problem by observing the software system's execution and triggering an invariant check only if the software system's state changed in a relevant manner. By observing the execution and recognizing state changes of a software system during runtime, we are able to drastically reduce the number of performed invariant checks. The proposed approach has been implemented for the Java programming language but is, in principle, applicable to other languages also.

We evaluated its correctness, performance and scalability on several open source case studies.

Kurzfassung

Softwaresysteme schlagen fehl. Obwohl Softwareentwickler ihr möglichstes versuchen Fehler in Softwaresystemen zu vermeiden sind Fehler unvermeidbar. In gewissen Fällen sind die Auswirkungen der Fehler katastrophal, resultieren in hohen finanziellen Verlust und können sogar Menschenleben gefährden. Oft führt das fehlerhafte Verhalten zu einem ungültigem Systemzustand, welcher zu weiteren Folgefehlern führen kann. Fehlerhafte Systemzustände können durch das Verwenden von Klasseninvarianten entdeckt werden. Klasseninvarianten beschreiben die Menge der gültigen Zustände eines Softwaresystems und dürfen nicht verletzt werden. Durch intensives Testen kann sichergestellt werden, dass diese Invarianten nicht verletzt werden. Dadurch kann das Überprüfen von Invarianten Fehler vermeiden.

Existierende Ansätze zum überprüfen von Invarianten erfordern hohen manuellen Aufwand, im speziellen wo diese Überprüfungen im Quellcode direkt platziert werden. Invarianten müssen lediglich überprüft werden falls ihre Gültigkeit betroffen sein könnte. Dies erfordert sorgfältige Platzierung der Überprüfungen der Invarianten im Quellcode. Diese Platzierungen müssen jedes mal berücksichtigt werden sobald sich der Quellcode oder die Definition der Invarianten ändert. Die einzige Alternative ist das durchgängige Überprüfen aller Invarianten nach jedem Schritt der Ausführung des Systems. Allerdings ist dies nicht skalierbar.

Diese Masterarbeit präsentiert einen neuartigen Ansatz zum überprüfen von Invarianten welcher automatisch entscheidet zu welcher Zeit Überprüfungen stattfinden müssen. Dieser Ansatz skaliert erheblich besser als überall Überprüfungen durchzuführen und ist garantiert korrekt.

Außer dem definieren von der Invarianten ist keine weitere Benutzerinteraktion notwendig. Der Ansatz vermeidet dass genannte Platzierungsproblem durch überwachen der Ausführung eines Softwaresystems und löst das Überprüfen von Invarianten nur aus wenn sich der Zustand in einer relevanten Weise geändert hat. Durch das überwachen und feststellen von Änderungen des Systemzustandes zur Laufzeit sind wir in der Lage die Anzahl der Überprüfungen drastisch zu reduzieren. Der Ansatz wurde vollständig für die Programmiersprache Java implementiert, könnte aber prinzipiell an anderen Sprachen angewendet werden.

Zum Auswerten des Ansatzes bezüglich Korrektheit, Performance und Skalierbarkeit wurden diverse Open Source Fallstudien verwendet.

Contents

Abstract iii

List of Figures xi

List of Tables xiii

List of Algorithms xv

List of Listings xvii

1 Introduction 1
- 1.1 Motivation . . . 1
- 1.2 Scope . . . 2
- 1.3 Problem Illustration . . . 3
- 1.4 Vision and Goals . . . 7
- 1.5 Solution . . . 8
- 1.6 Structure of this Thesis . . . 9

2 Background 11
- 2.1 Constraint Language . . . 11
  - 2.1.1 OCL Types . . . 12
    - 2.1.1.1 Collections . . . 14
    - 2.1.1.2 OCL to Java Type Mapping . . . 15
  - 2.1.2 OCL Expressions . . . 17
    - 2.1.2.1 Context of OCL Expressions . . . 17
    - 2.1.2.2 Literals . . . 18
    - 2.1.2.3 Variables . . . 18
    - 2.1.2.4 Type Expressions . . . 19
    - 2.1.2.5 Model Access . . . 20
    - 2.1.2.6 Conditional Expression . . . 21
    - 2.1.2.7 Loops . . . 23

2.1.3 OCL translation examples . . . 24
2.1.3.1 Using a List Index as Part of Constraint . . . 24
2.1.3.2 Create a Collection of Attributes . . . 25
2.2 Java Platform Debugger Architecture . . . 26
2.2.1 Connecting to Target Application using JDI . . . 27
2.2.2 Notifications . . . 29
2.2.3 Accessing Values in Target Application . . . 31

3 Approach **33**
3.1 Basic Definitions . . . 33
3.2 Architecture . . . 39
3.2.1 Observer / Wrapper . . . 39
3.2.2 Constraint Checker . . . 40
3.2.3 Central Data Storage . . . 40
3.2.4 Invariant Definition Storage . . . 41
3.2.5 User Interface . . . 41
3.2.6 Invariant Analyzer Core . . . 41
3.3 Observing the Runtime Environment . . . 42
3.3.1 Creation of New Objects . . . 42
3.3.2 Scope Element Changes . . . 42
3.3.3 Destruction of Objects . . . 43
3.3.4 Observing Operation States . . . 43
3.4 Invariant Checking Mechanism . . . 44
3.4.1 Creation of New Invariant Instances . . . 44
3.4.2 Triggering Revalidations . . . 45
3.4.3 Scope Clean-up . . . 46
3.4.4 Constraint Checker . . . 47
3.4.4.1 Top-Down Validation . . . 48
3.4.4.2 Bottom-Up Validation . . . 48

4 Implementation Details **51**
4.1 OCL Constraint Checker . . . 51
4.2 Observing Component . . . 52
4.2.1 Observing the Creation of new Objects . . . 52
4.2.2 Observing Changes of Collection Contents . . . 53
4.2.3 Accessing / Converting Values . . . 53
4.2.3.1 Querying Collection Contents . . . 54
4.2.3.2 Value Caching . . . 55
4.2.4 Connection options . . . 55
4.2.5 Synchronization with Target Application . . . 56

4.3 Invariant Definition Storage . . . 56
4.4 Graphical User Interface . . . 56
4.4.1 Setting Preferences . . . 61
4.5 Core Invariant Analyzer Component . . . 61
4.5.1 Notification Filtering . . . 63
4.6 Central Data Storage . . . 65

5 Evaluation **67**
5.1 Correctness . . . 67
5.2 Performance . . . 69
5.3 Scalability . . . 71
5.4 Complexity . . . 73
5.5 Threats to Validity . . . 75

6 Related Work **77**
6.1 Static checking . . . 77
6.2 Dynamic checking . . . 79

7 Conclusion and Future Work **81**
7.1 Increasing Performance . . . 81
7.2 Fixing Errors . . . 82
7.3 Supporting Additional Design by Contract Paradigms . . . 83
7.4 Invariant Checking Triggering Options . . . 84
7.5 Seamless Integration with IDE . . . 84
7.6 Refactoring Implementation . . . 85

Bibliography **87**

A Evaluation Tools / Applications **91**
A.1 ATM . . . 91
A.2 jPacMan . . . 92
A.3 GanttProject . . . 93

Glossary **97**

Curriculum Vitae **101**

List of Figures

1.1 Double linked list structure . . . 3
1.2 Instantiation of a double linked list . . . 5

2.1 Simple OCL Types . . . 13
2.2 OCL Collection Types . . . 14
2.3 Abstract Syntax of OCL Expression . . . 17
2.4 Abstract Syntax of LetExpressions . . . 19
2.5 FeatureCallExpressions . . . 21
2.6 IfExpressions . . . 22
2.7 JDI event overview . . . 30
2.8 Locatable Events . . . 30
2.9 JDI Value Types . . . 32

3.1 Simplified object orientated structure . . . 34
3.2 Structure of used data elements . . . 34
3.3 Syntax Tree of invariant CorrectlyLinked . . . 37
3.4 Possible changes . . . 39
3.5 Architecture Overview . . . 40
3.6 Validation Tree of $CL[first]$ . . . 48
3.7 Validation Tree of $CL[third]$ . . . 49
3.8 Revalidation of $CL[first]$ . . . 50

4.1 Invariant Editor . . . 58
4.2 Code Completion . . . 58
4.3 Invariant View . . . 59
4.4 Validation Tree Navigation . . . 60
4.5 Structure Navigator . . . 61
4.6 InvariantAnalyzer Preferences . . . 62
4.7 Connector Preferences . . . 62

5.1 Amount of Invariant Instances during Evaluation . . . 68

LIST OF FIGURES

5.2 SpeedUps of incremental approach 72

A.1 GanttProject Screenshot 94

List of Tables

2.1 Equivalent OCL and Java Types 16
2.2 OCL to Java standard type mappings 16
2.3 Java to OCL standard type mappings 16
2.4 Literal examples 18
2.5 Notification Filtering Options 30

3.1 Invariant example 37

4.1 Stored invariant information 57

5.1 Evaluation Setup 69
5.2 Execution Times with invariant checking disabled and enabled 69
5.3 JDI field access overhead 70
5.4 JDI method call overhead 70
5.5 JDI method call with argument overhead 70
5.6 Number of invariant validations 72

List of Algorithms

1 Analyzing Changes and dispatching Invariant Instance validations 45
2 Updating global Scope 47
3 Generating change notifications for collection changes 53

List of Listings

1.1 Node implementation . . . 4
1.2 `size()` method implementation . . . 6
2.1 OCL constraint with context . . . 18
2.2 LetExpression example . . . 19
2.3 Rewritten CorrectlyLinked using if . . . 22
2.4 IfExpression with different subtypes . . . 22
2.5 IfExpressions for error handling . . . 23
2.6 `forAll` Example . . . 24
2.7 `forAll` Example with multiple iterators . . . 24
2.8 `select` Example . . . 24
2.10 List Index example in Java . . . 24
2.9 `iterate` Examples . . . 25
2.11 List Index example in OCL . . . 25
2.12 Collection of Attributes example in Java . . . 25
2.13 Collection of Attributes example in OCL . . . 25
2.14 LaunchingConnector code snippet . . . 27
2.15 SocketAttachingConnector code snippet . . . 28
3.1 Method contracts for accessing field values and invoking methods . . . 35
3.2 Link method . . . 43
4.1 Example Invariant Definition File . . . 57
A.1 ATM Invariants . . . 91
A.2 jPacMan Invariants . . . 92
A.3 GanttProject Invariants . . . 94

Chapter 1

Introduction

1.1 Motivation

Software system failures can be disastrous, especially if life is at risk or the failure causes physical damage of any kind. To minimize the critical failures, we have to ensure that a software system behaves according to its specification. The concept of design by contract is no new idea at all [24], but past events and the increasing complexity of software systems show that this concept becomes more and more indispensable nowadays [18]. Although the core emphasis is to ensure correct functional behavior, one may also use those concepts to describe the valid states of a system. A system transitioning into an invalid state, either by internal or external events, may cause incorrect behavior further on. This means that correct behavior and valid states correlate directly in most cases. The valid states of a software system modeled or implemented using an object orientated language can be defined using class invariants. Usually those invariants are expressed as constraints written in a language as least as powerful as first order logic. Certain modeling languages like the UML [30] even provide native support to document such constraints.

One way of reducing the risk of software failure is to check the behavior of a program statically by means of formal proving. While formal proving works great for verifying the correct implementation of algorithms, with the increasing complexity of state-of-the-art software systems it does not. In case of loosely coupled components, that may even be exchanged at runtime (and therefore change the behavior), statical analysis is mostly not applicable since the exact behavior is unknown prior to execution. It is simply not feasible to take all possibilities into account, especially if new / exchanged components are by a third party. Complex software systems may even be self-adaptive, making dynamic verification a requirement [2]. Furthermore formal proving always requires user interaction. There are tools aiding a user in the proving process, but it is impossible to automate it altogether (see Chapter 6).

When statical analysis reaches its limits, developers check those invariants dynamically

during runtime. However, testing class invariants in an implementation is a rather hard task and rarely done, even less in a productive environment. This is caused by the complexity of state-of-the-art software systems, where a developer has to decide on proper places for invariant checks. They are either added manually or automatically to almost every method of a class. Furthermore, a system is not just delivered to a customer or sold off the shelf. It is continuously maintained and evolves over time, making it even harder harder to keep the locations of invariant checks up-to-date (see Section 1.3).

Hence, there is clearly a need for an approach that performs runtime checks and automates as much as possible. Meaning that developers do not have to decide the appropriate time for an invariant check to occur. Additionally, the approach should perform as many checks as necessary and as few as possible. This paper contributes a new novel fully automated tool-supported incremental scalable approach, aiming at reducing both required user interactions, as well as, required computation time, as much as possible. We evaluated this on various openly available software systems, ensuring that approach is both correct and scales up to larger scale systems.

1.2 Scope

This thesis does not deal with all aspects associated with design by contract. We are not exactly interested in correct operational behavior, the focus is on correct or valid system states from a global perspective. Hence, we do not support checking operations' pre- or postconditions, since it suffices to check them locally (i. e. at the beginning or end of the corresponding operation respectively). Furthermore, we only care about publicly visible states of classes and therefore do not support loop invariants. Only class invariants for object orientated systems are considered at the moment.

Generally our approach would be applicable to any object oriented language, but we chose Java as a reference language for the implementation due to its high popularity. Although we specifically implemented our approach to fully support the Java language it may also be used for other languages that compile to Java byte code and are executed in a Java Virtual Machine, e. g. Scala. Another deciding factor for using Java was the fact that it provides sophisticated tool support for observing a target application, which is already commonly used by debugging, profiling, and monitoring tools, etc. Switching to another language could be done easily by providing similar means to access the required information.

Further we restricted the constraint language used for defining invariants to the Object Constraint Language (OCL). For a detailed description of OCL and the reasoning behind this choice, see Section 2.1. Although our language of choice is OCL, the actual constraint checker used to validate the invariants is generic and does not strictly depend on OCL. If someone would provide a parser for a different language that would map its concepts to our own, exchanging the constraint language would be quite easy.

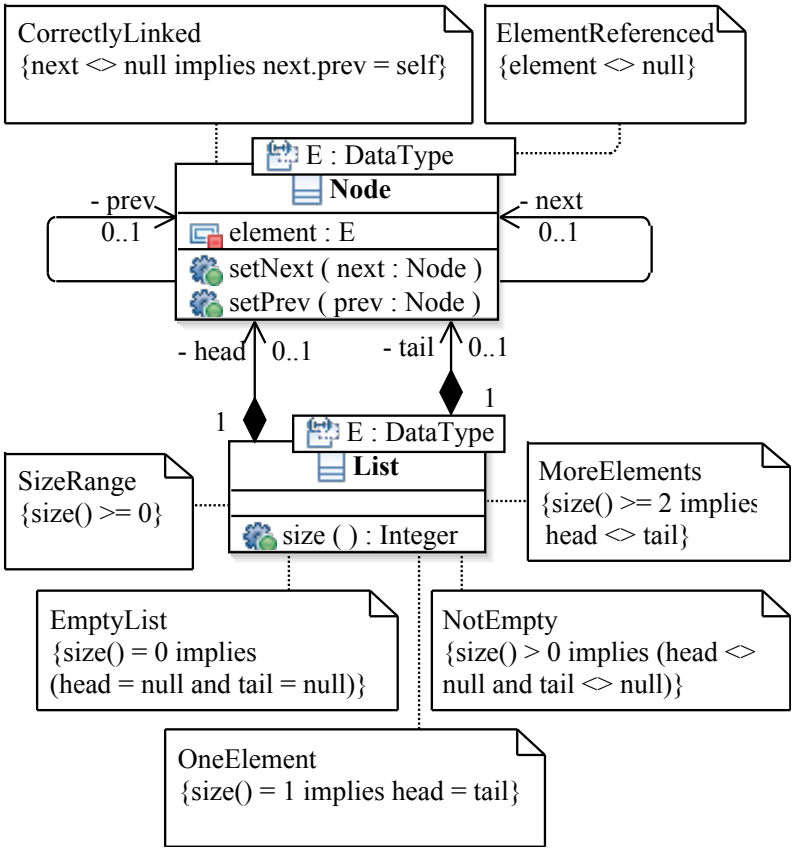


Figure 1.1: Double linked list structure

1.3 Problem Illustration

We illustrate the problem of dealing with invariants using a simple example, namely a double linked list. Figure 1.1 shows the class structure of a double linked list and its invariants. Listing 1.1 shows a possible implementation of the `Node` class. Invariant checks have been inserted manually using assertions. The common notion of invariants regarding design by contract is, that an invariant has to hold whenever an instance of its class is in a publicly visible state [25]. Thus, the naive approach is that the invariant becomes part of any postcondition. This means that after any method execution, both its postcondition and all class invariants must hold. Basically, we need to revalidate the invariants after the creation of an object and whenever the state of the object changes, i. e. methods causing side effects. We can detect several problems treating invariants as part of method postconditions and manually inserting checks.

Problem 1.3.1 (Invariant checking locations)**.** The first problem we can observe, is that the developer has to decide on the proper locations for the invariant checks. Although they are

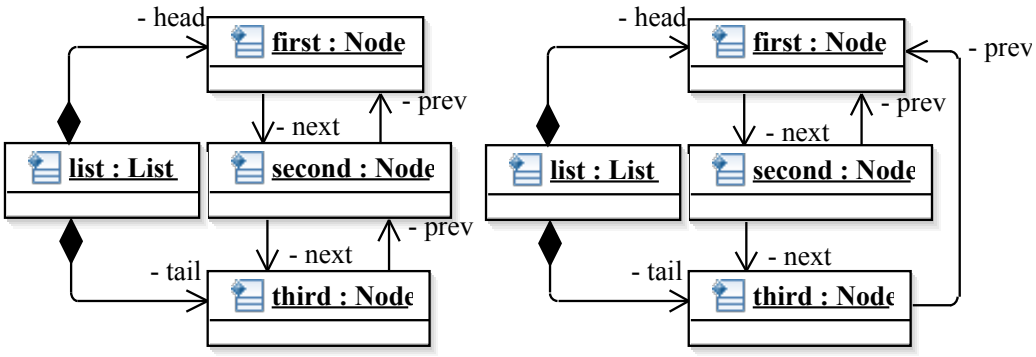
```
class Node<E> {
  Node<E> next = null;
  private Node<E> prev = null;
  private E element;
  Node(E element) {
    this.element = element;
    assert checkElementReferenced();
  }
  void setNext(Node<E> next) {
    this.next = next;

    assert checkInvariant();
  }
  Node<E> getNext() {return next;}
  void setPrev(Node<E> prev) {
    this.prev = prev;
    assert checkInvariant();
  }
  Node<E> getPrev() { return prev;  }
  E getElement() {  return element; }
  private boolean checkCorrectlyLinked() {
    return next == null || next.prev == this;
  }
  private boolean checkElementReferenced()  {
    return element != null;
  }
  private boolean checkInvariant()  {
    return checkElementReferenced() && checkCorrectlyLinked();
  }
}
```

Listing 1.1: Node implementation

quite obvious in an easy example like this, in a more complex class it may not be that easy to determine the locations of invariant checks. This is the reason why existing approaches tend to be exhaustive (checking all invariants of a class in every method, as discussed later in Chapter 6).

Problem 1.3.2 (Language "barrier")**.** Usually the language used to describe invariants at design time is different from the one used in the implementation. For example, the class diagram shown in Figure 1.1, uses the Object Constraint Language (OCL [28]) to define the invariants, which is commonly used for models written in Unified Modeling Language (UML [30]) or other Meta Object Facility (MOF [29]) based languages. On the other hand the implementation shown in Listing 1.1 uses plain Java to check the invariants *CorrectlyLinked* and *ElementReferenced*. Translating between different languages manually is error prone, especially if the languages follow different paradigms (declarative vs. imperative). Most existing approaches



(a) Initial linked list **(b)** Broken linked list after change

Figure 1.2: Instantiation of a double linked list

use proprietary languages that closely resemble the language the target application is written in.

Problem 1.3.3 (Software evolution)**.** A software system is not set in stone, it will be maintained and evolves over time. During the system's life-cycle, existing methods or invariants may be modified or new ones added. The developer introducing these changes has the responsibility to ensure that the class stays consistent, i. e. invariant checks are still performed whenever necessary. We see that inserting invariant checks manually is not a proper solution. It's simply too time consuming and error prone. In our linked list example we manually optimized the invariant check at the end of the `Node` constructor. Since we know that at this point there is simply no way that the invariant *CorrectlyLinked* could be violated, we simply omit it. But, if for whatever reason, someone changes the constructor such, that it could violate the invariant and does not change the invariant check appropriately, we end up with an invalid class definition possibly causing problems later on. Although there are existing automated approaches which deal with some of the problems described above (as described in Chapter 6), not all problems are solved or the solutions do not provide sufficient flexibility.

Problem 1.3.4 (Flaws of local checks)**.** The most serious problem is, that by just checking invariants locally for the affected classes may not reveal all violations. Figure 1.2 shows two possible states of a double linked list. The initial list shown in Figure 1.2a is perfectly fine and does not violate any invariant. After calling `third.setPrev(first)`, the list transitions into the state shown in Figure 1.2b, which is obviously not a proper double linked list any more. Using our manually inserted assertions, we are unable to detect this error, since the invariant will only be checked for the node `third`, which still holds. Due to the nature of the implementation, the violation of the invariant *CorrectlyLinked* goes unnoticed for node `second`. A similar issue is the existence of non-private fields. Generally speaking, the values of those fields can be set from outside the object they belong to (this is programming language dependent, but possible in most state-of-the-art languages). In such cases local invariant checks again fail to detect the

violation, since no method of this object is called. Thus, a fair amount of invariants would also need checking from locations, that are not belonging to the object itself. Of course, this contradicts modern design / development principles, generally speaking it is bad practice to use non-private fields, but if they are used, other developers using such a class without knowing its implementation details can be quite risky.

Existing automated approaches tend to be exhaustive, in a sense that they insert checks at the end of each publicly visible method. This leads to unnecessary checks being performed. Methods which do not modify the state of an object, should not cause invariant checks (e. g. a method such as `size()` in our illustration, shown in Listing 1.2). Some methods, on the other hand, do change the state of an object, but cause too many invariant checks nonetheless. In our example the method `setNext(...)` does not modify the field element and there is no need to check the invariant *ElementReferenced*, since at this time there is no way that it is violated, if it were not already the case. Optimizing the invariant checks at this point on the other hand directly leads to Problem 1.3.3. The worst part is that some approaches insert checks even if the object in question is not in a publicly visible state, which may lead to detecting many false negatives (i. e. detecting an invariant violation when there actually is none, e. g. because it is only temporary during the execution of a helper or recursive method). Certainly, a software system is designed for a specific purpose, it should spend most of its computation time fulfilling that purpose. Performing unnecessary invariant checks is simply a waste of computation time.

```
int size()  {
  int size = 0;
  Node<E> current;
  for(current = head; current != null; current = current.getNext
     ()) size++;
  return size;
}
```

Listing 1.2: `size()` method implementation

Problem 1.3.5 (Hard coded invariants)**.** A minor problem is that the invariants are hard-coded inside the classes. There is no way to add, remove or modify the invariants during the execution of the system. Although invariants should be defined at design time and the probability that their definition will change is pretty low, performing the invariant checks consumes CPU time. It would be helpful to disable certain invariants or add new ones during runtime, if a developer notices something suspicious going on in the system or needs it to execute faster. Existing approaches, especially those weaving the checks into the source or byte code of the application, do not allow such on the fly modifications and require recompilation and restarting the system.

1.4 Vision and Goals

By writing invariant checks manually, a developer may theoretically be able optimize their execution time, as well as, the necessary calls. But, as we already mentioned, this is both error prone and not feasible. Our vision from a user perspective is to minimize the required effort as much as possible, while still yielding correct results and at the same reduce the execution time an application spends on performing the checks by a huge degree. A developer shall only be responsible for defining the required invariants, which is already a hard enough task. The fairly error prone process of ensuring that invariant checks are performed whenever necessary, should be fully automated.

While invariant checks are especially useful during testing, we also wanted our approach to be useful to monitor productive systems. This means that our approach should on one hand provide instant feedback whether it detected violations and scale up to systems where preforming invariant checks is required more frequently. As we will discuss in Chapter 6, existing approaches tend to check invariants in an exhaustive manner. This behavior may, and most certainly will, slow down the system to a noticeable extent. We want to avoid this problem by minimizing the invariant checks performed, while making sure that they are performed often enough such that all violations are discovered. Generally speaking, it is impossible that the outcome of an invariant changes as long as no intermediate results change. To further encourage this, we wanted to make the approach as flexible as possible. A user should be able to add, remove, or modify class invariants at any time during the runtime of the system. Since checking complex class invariants might consume quite some CPU time, and a system should not waste time performing unnecessary invariant checks, if one can be sure that a certain invariant will not ever be violated, there should be a way to avoid further checks of this invariant. On the other hand, if one recognizes that the system does not work according to its specification, adding or enabling new invariants is a useful way to determine the cause of the observed errors. We want to accomplish this by decoupling the invariant checks from the system in question. First of all, the actual invariant definition are separated from the systems source code. Secondly, the invariant checking entity is a separate component that can be attached or detached from the system at any time. Keeping the invariant definition in a separate repository enables us to modify them during the runtime of a system without the need of re-compiling them. By detaching the invariant checking component, the system works as if there were no invariants defined at all, preserving the original behavior.

The approach should be incremental, meaning that it will never check all defined in batch-like fashion (checking unnecessary ones). But, the result still have to be the same, i. e. if at some point in time one would check all invariants it has to come up with the same results as our incremental one. Another goal is to make the approach as generic as possible. It should theoretically be able to work with different target runtime environments or programming languages. Summoning it up the following goals were set:

1. Usability aspects:

   (a) Minimizing required user effort: Invariants only have to be defined by users themselves and the approach takes care of when they will be checked by itself.

   (b) Flexibility: Users have to be able to change the set of defined invariants and their definitions on the fly without the need to alter source or recompilation.

2. Incremental: The approach has to work incrementally while still providing correct results.

3. Generic: Provide an architectural allowing easy adaptation to other environments / languages.

4. Performance: It should be applicable for both testing and monitoring a productive environment, i. e. not slowing down the target application such it becomes unusable.

5. Scalable: Incremental invariant checking should provide scalability up to bigger systems or ones that require lots of invariant checks within a given time frame, especially compared to existing approaches.

Ultimately, we want to come up with a monitoring tool capable of detecting a target system being in an invalid state, but still allowing it to continue its operation by using emergency reactions or fixing it on the fly (see Chapter 7). This is not part of the goals of this thesis any more, just a greater vision with this work being the first step towards this direction.

1.5 Solution

To address the identified problems, first of all we separate the responsibility of performing invariant checks from the system. We introduce a new invariant checking component that runs in an entirely different process or even on a different physical machine, in fact it is an different application which is not related to the target system whatsoever. This component is responsible for storing the invariant, determine when they need to be checked and finally report the results of checks. Therefore, we got rid of the invariants being defined in the source code of the application and the target system can now concentrate on performing the tasks it is intended to. It is capable of connecting to a target system at any time.

We reduce the number of performed invariants checks by observing the execution of the invariant checks themselves as well as changes imposed on the state of the target system. Invariants are checked incrementally, only if the results could have changed due to changes.

The gap between design and implementation is bridged by using OCL as the language for the invariants. Reusing constraints written in OCL at design time requires almost no effort.

1.6 Structure of this Thesis

This section explained the motivation behind this thesis, pointing out problems with existing approaches and showed the goals. Further on, Chapter 2 will briefly describe the technologies this work is based on, including the constraint language used in the implementation, as well as the reasoning behind choosing the particular language. Chapters 3 and 4 will discuss the approach we came up with to perform incremental invariant checking and how it has been implemented respectively. Chapter 5 depicts the evaluation of the approach, clearing whether the set goals have been reached. Chapter 6 discusses related work and points out the differences between our approach and already existing work in this area. Finally, Chapter 7 concludes this thesis, lining out what has actually been achieved and discusses some ideas how the existing approach could be extended to reach the overall vision.

Chapter 2

Background

This chapter briefly describes the technologies / libraries used in this work. Section 2.1 introduces the language used to specify invariants, why this particular languages has been chosen and what it is capable of. Section 2.2 discusses the mechanism used to communicate with the target application for invariant checking.

2.1 Constraint Language

There were lots of possibilities for choosing a suitable constraint language, including designing a new one. We decided not to create a constraint language ourselves, but instead using an existing one to decrease the effort necessary to learn how to use the tool. To further support this intention, the language had to be widely used and well recognized within the software industry. Furthermore we preferred a declarative language, a class invariant should define what structural properties need to hold in order to consider a state valid, not how it is checked. There already exist constraint languages especially designed for the purpose of constraining at the implementation level, e. g. the Java Modeling Language (JML). The downside is that most of those languages are aimed at specific programming languages, as their name suggests.

For those reasons we chose the Object Constraint Language (OCL) [28], which was originally designed to enrich meta models based on the Meta Object Facility (MOF) [29] with additional constraints not expressible using pure MOF. Because of MOF's strict modeling language architecture, OCL can not only be used at the meta modeling level, but also on the modeling level. Since OCL is a declarative language, certain specific features of some object orientated programming languages, especially imperative ones, may not be directly expressible using OCL, although it covers a common basis used by most languages. Especially basic concepts like property (field) access and operation (method) calls are supported natively. The functional nature of OCL implies that expressing iterative concepts like loops is not possible directly. Anyhow, OCL offers standard collection functions powerful enough to express most loops in a functional

fashion. Also note that OCL is entirely side effect free. An OCL expression may contain variable definitions, but reassignments of those or properties of an accessed model element is not possible.

Usually, OCL operates on a fixed (set of) predefined meta-model(s). In our case the meta-models is composed of all the classes currently loaded in the target application including Java primitive types. Note that OCL contains some UML specific constructs / extensions that are entirely irrelevant for this work. The following subsections will give a brief introduction to OCL, as well as show how OCL standard types and operations can be mapped to Java ones. Finally, Section 2.1.3 will show some examples how constraints written in Java can be translated into an OCL expression. Although there may be constraints not expressible in OCL due to the different language paradigm, we were able translate every single constraint we found in the case studies to OCL. This may have required certain workarounds, but was possible nonetheless.

2.1.1 OCL Types

Figure 2.1 shows the structure of simple standard OCL types. All those types are instances of the MOF meta-type Classifier and all types defined in a target model are considered being instances of the meta-type Class, which is itself a subtype of Classifier. Class is in some way the equivalent of java.lang.Class in Java. There is no real equivalent of Classifier in Java, but we treat as also equivalent to java.lang.Class, although instances of Classifier include primitive types. This can be done since primitive Java types are not used in our implementation. Whenever a primitive type is accessed in the target application it is immediately converted to its object oriented counterpart.

The following list describes the OCL standard types (excluding collections, see Section 2.1.1.1). Finally Table 2.2 show which standard Java types are mapped to which OCL ones or considered being subtypes thereof.

OclAny is similar to the class Object in Java in that it is treated as a supertype of all other types, including classes defined in the target model. The predefined operations on *OclAny* are equality tests (`=, <>`) and and type expressions (see Section 2.1.2.4).

OclVoid is another special type, which has a sole instance called `null` and represents the absence of a value. It is treated as a subtype of *all* types regarding type conformance, meaning that whatever type a property, operation or argument has, `null` is a valid value. Property or operation calls on null will result in `invalid` (except certain special operations like equality comparison).

OclInvalid which is not shown in the figure since it is not supported by our tool is in a sense similar to *OclVoid*. Like *OclVoid* it is also treated as a subtype of every other type and has a single instance called `invalid`. If An expression return invalid, it can be considered as a runtime error during the validation of the expression (e. g. a property call on `null`). To some degree it can be compared to runtime exceptions in Java. Example 2.1.6 in

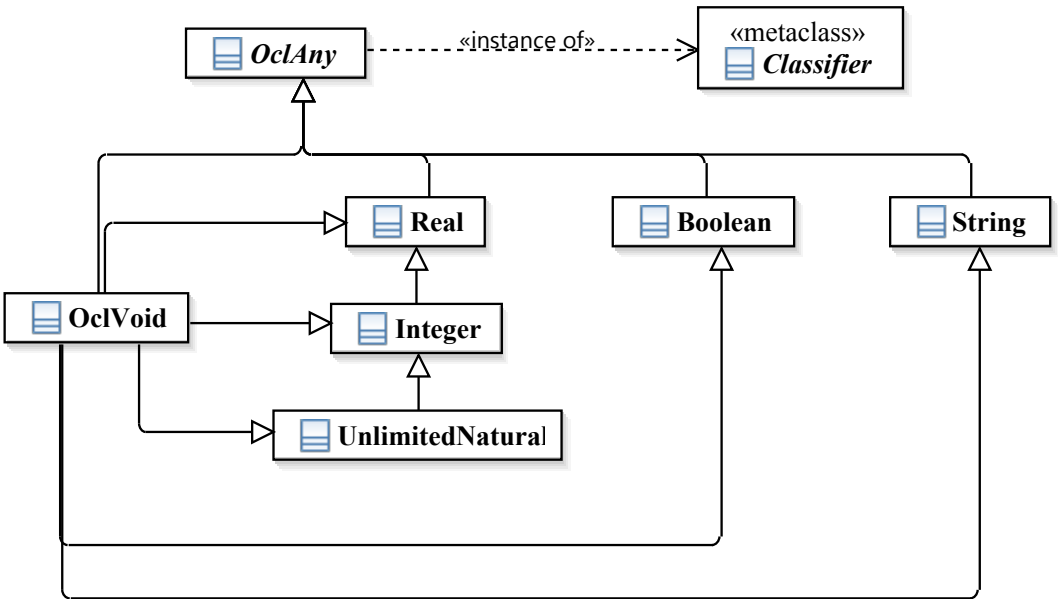


Figure 2.1: Simple OCL Types

Section 2.1.2.6 shows how basic error handling can be performed if an expression returns `invalid`.

Boolean represents standard truth values. There are exactly two instances of *Boolean*, `true` and `false` representing their respective truth values. Standard operations defined on *Boolean* are the usual boolean algebra ones, e. g. and, or, not, etc. Implementing those operations in Java is straight forward since equivalent operations for truth values already exist in Java.

String is just an ordinary character string like `'this is a String'` and is equivalent to the `java.lang.String` class in Java. Operations defined on String include concatenation, extracting substrings and converting them to other standard types if applicable.

Real represents the mathematical set of real values ($\mathbb{R}$) like 3.14. There does not exist an actual equivalent in Java since numbers are limited (have a maximum value) there, but all Java types representing numbers are treated as being subsets of *Real*. Predefined operations are standard arithmetic operations like additions, subtraction, etc. as well comparison operators like $\leq, >$.

Integer represents the set of integer values ($\mathbb{Z}$) like 42. In OCL Integer as a subtype of *Real*, which is correct mathematically speaking since $\mathbb{Z} \subseteq \mathbb{R}$. This is not true for Java types representing integer values since there exist integer values which can not be represented using types like `double` or `float`. Our implementation acts like this would be the case anyhow for simplification purposes. The operations defined on Integer are practically the

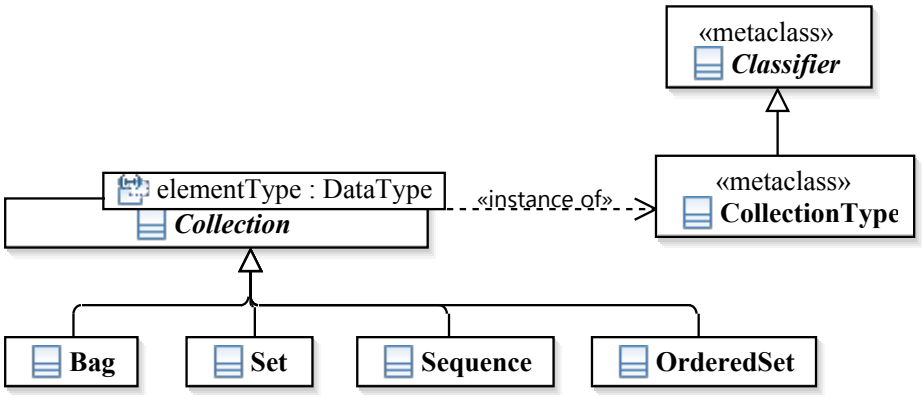


Figure 2.2: OCL Collection Types

same as for Real but expect another integer as an argument and will also return an integer if applicable.

UnlimitedNatural is irrelevant in this work. It has been adopted from UML and serves as a means to represent multiplicity values of attributes and references. Mathematically, it is the set of positive integers including a special value called `unlimited` denoting infinity ($\mathbb{N}_0 \cup \{\infty\}$). It is a subtype of integer in that it shares its behavior and operations, although an operation with an integer argument called using `unlimited` will usually result in `invalid`.

2.1.1.1 Collections

Figure 2.2 shows the structure of available collection types in OCL. Although the are indeed instances of the Classifier meta-type (*CollectionType* to be specific), they are special and are not subtypes of *OclAny*. This means that an instance of a Collection does not conform to *OclAny*, i. e. it is not permitted to pass a collection to an operation call excepting an argument of type *OclAny*. In Java there is no special treatment regarding collections, they are treated as ordinary classes and as such are indeed subtypes of the Object supertype. In OCL collections are real containers for other objects and have no attributes themselves. Even calling operations on collections has a different notation in OCL's concrete syntax (`->` instead of `.`). The *elementType* of collections is the type of the objects the collection contains (or a supertype thereof). A collection $A(X)$ (X being the *elementType* of A) conforms to an another collection $B(Y)$ if and only if $A \preceq B \wedge X \preceq Y$ (read $A \preceq B$ as A is a subtype of B). This would not be type-safe if the contents of a collection could be subject the changes , but as already mentioned OCL is side-effect free making this a valid conformance rule. In Java, the rule would be $A \preceq B \wedge X = Y$.

Collection is the abstract supertype for all other collection types. The element type is bound to the type of the elements it contains, which may be a collection again. The Collection type defines basic operations implemented by all specific subtypes. Those include, but

is not limited to, querying the number of elements it contains, whether certain elements are contained, etc. Furthermore it defines operations to convert between different kinds of collections and the "loop" constructs (see Section 2.1.2.7).

Bag is the most basic type of collections, that does not have an equivalent in Java. There is no ordering defined on the elements contained in a Bag and it allows duplicates, meaning that an element may be contained an arbitrary number of times.

Set represents the typical mathematical set, i. e. it does not allow duplicates and there is no specific order defined on the elements it contains. Additionally to the operations defined on Collection, Set defines typical set operations like union, intersection, etc. It is similar to collections in Java implementing the java.util.Set interface.

Sequence is the typical mathematical relation mapping an integer value (the index) to an associated element ($\mathbb{N} \rightarrow$ E, where E is the element type of the sequence). Additionally to the operations defined on Collection, Sequence defines operations to access its contents using an index or creating new sequences by adding / removing elements at specific positions. It closely resembles collections implementing the java.util.List interface in Java.

OrderedSet can be interpreted as a hybrid of a Set and a Sequence. It does not allow duplicates and contents are ordered using an index. The name suggests that it is equivalent to Java collections implementing java.util.SortedSet, although it is not. A SortedSet in Java uses a Comparator or the natural ordering of its contents to sort them, users can not influence directly at which positions an element will be inserted and elements can not be retrieved using an index. A similar structure in Java could be special implementation of java.util.List discarding duplicates.

2.1.1.2 OCL to Java Type Mapping

Since OCL types are used in the definitions of the invariants and the target application uses Java ones, it has to be defined when the types from the different domains are considered conforming to each other. Values extracted from the target application may be used as arguments to standard OCL operations and those returned by OCL operations may also be used as arguments of methods called in the target application. Table 2.1 shows the OCL and Java types that are considered equivalent, i. e. an instance of the Java type conforms to the OCL type and vice versa. On the other hand there are types both in OCL and in Java for which no real equivalent is available. For those types an OCL to Java (OCL types conforming to Java ones, see Table 2.2) and a Java to OCL mapping has been defined (Java types conforming to OCL ones, see Table 2.3). Application specific non-standard Java types (classes) are treated as-is, i. e. standard Java conformance rules apply. If types are used as arguments that are not covered by the conformance rules stated in the before mentioned tables users have to covert the values to acceptable types themselves, using native conversion functions.

OCL Type	Java Type
Classifier	`java.lang.Class`
OclAny	`java.lang.Object`
String	`java.lang.String`
Boolean	`java.lang.Boolean`
Collection(T)	`java.util.Collection<T>`
Set(T)	`java.util.Set<T>`
Sequence(T)	`java.util.List<T>`

Table 2.1: Equivalent OCL and Java Types

OCL Type	Java Class
Real	`java.lang.Double`
Integer	`java.lang.Integer`
Bag(T)	`java.util.Collection<T>`

Table 2.2: OCL to Java standard type mappings

Java Type	OCL Type
`java.lang.Character`	*String*
`java.lang.Byte`	*Integer*
`java.lang.Short`	*Integer*
`java.lang.Integer`	*Integer*
`java.lang.Long`	*Integer*
`java.lang.Float`	*Real*
`java.lang.Double`	*Real*

Table 2.3: Java to OCL standard type mappings

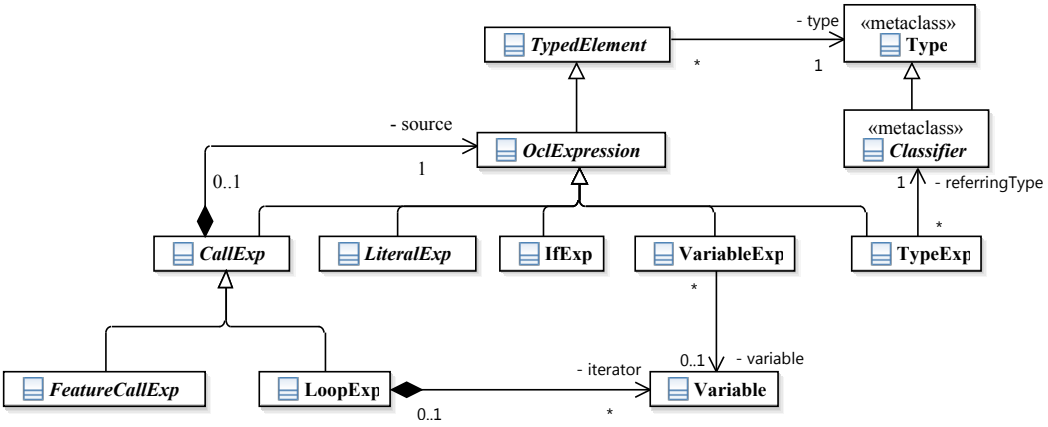


Figure 2.3: Abstract Syntax of OCL Expression

Note that we do not deal with Java primitive types (`int`, `boolean`, ...) at all, the implementation ensures that whenever a primitive is accessed in the target application it will be automatically boxed into its object oriented equivalent (`java.lang.Integer`, `java.lang.Boolean`, ...). This also applies in the other direction as well, meaning that whenever a method is called in the target application using a primitive as an argument it will be converted to the appropriate one.

2.1.2 OCL Expressions

Figure 2.3 shows partially the abstract syntax of OCL expressions, leaving out constructs that are irrelevant or not supported by our tool. In contrast to imperative languages like Java, OCL has no notion of statements. Whereas in Java a code block consists of a sequence of statements and each of those of expressions, an OCL constraint (or query) consists of single expression, however arbitrary complex. Every expression in OCL is typed, i. e. it must return some value and this value must conform to the expressions type. In the following we will describe standard OCL expressions and provide examples how to express them in the concrete syntax of OCL

2.1.2.1 Context of OCL Expressions

An important concept of OCL expressions is the context. As the name suggests OCL is primarily used to impose certain constraints on model elements. To specify for which type of elements a constraint has to hold its context has to be specified. Listing 2.1 shows the definition of the *CorrectlyLinked* invariant introduced in Figure 1.1 in the proper OCL format. The `inv` keyword states that the expression is actually an invariant (OCL also knows other expression types that will not be discussed here since they are irrelevant in this work). In conjunction with the context specified as Node, it means that this invariant has to hold for each instance of type Node. The context is used by the special variable self, which refers to the current context

Literal	Type
`3.14`	*Real*
`42`	*Integer*
`'foo'`	*String*
`true`	*Boolean*
`null`	*OclVoid*
`Set{42,'foo'}`	*Set(OclAny)*
`Sequence{7..14}`	*Sequence(Integer)*
`Tuple{name : String = 'Matt', age : Integer = 42}`	*Tuple(String, Integer)*

Table 2.4: Literal examples

element, i. e. the current Node instance the expression is validated on.

```
context Node
inv:  next <> null implies next.prev = self
```

Listing 2.1: OCL constraint with context

2.1.2.2 Literals

A *LiteralExp* in OCL is simply a constant / literal wrapped in an expression. The type of the expression is the type of literal itself and validating it always returns the literal itself. Literals may be of any of the basic types (including collections) or tuples. Literal expressions usually do not contain subexpressions (and thus are leaves of the abstract syntax tree), except for collection and tuple literals. Collection and tuple literals may contain subexpressions, which express their initial contents. Table 2.4 shows some examples of valid literal expressions.

2.1.2.3 Variables

OCL does not support actual variables, in a sense that the value they represent can not be modified, Only one-time assignments are possible. This means that variables in OCL are more or less just named constants comparable to `final` variables in Java. Variables basically occur in two types of expression: operations on collections expressing loops (see Section 2.1.2.7) and so called *LetExpressions*. Figure 2.4 shows the abstract structure of *LetExpressions* (abbreviated as *LetExp* in the figure). Variable defines the the variables name, its type (which must be stated explicitly). The value of the variable is the result of the *initExpression*, which may be an arbitrary expression, although its type must conform to the type of the variable. The second part of a *LetExpression* is the *inExpression* which is again an arbitrary expression that may refer to the previously defined variable by using a *VariableExpression*. A *VariableExpression* is simply the name of the variable in the concrete syntax of OCL and always returns the value the variable refers to. The result returned by the entire *LetExpression* is the same the one of

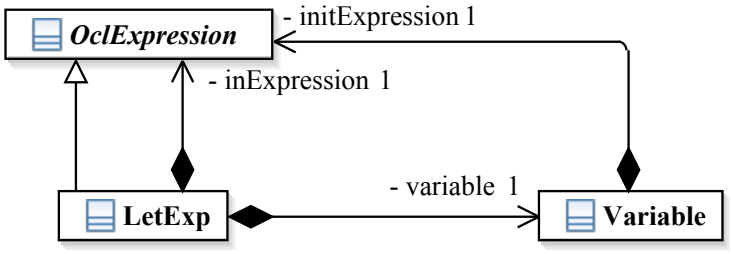


Figure 2.4: Abstract Syntax of LetExpressions

the *inExpression* (and thus its type). Because of its nature a *LetExpression* is only useful when the same (sub)expression would occur multiple times, as shown in Example 2.1.1.

A special variable is the one with the name self. It always returns the current context, e. g. the element an invariant is currently checked on.

Example 2.1.1 (LetExpression)**.** In the expression `a.b.c.d = a.b.c.e`, the subexpression `a.b.c` occurs multiple times, which on one hand causes the exact same expression to be validated multiple times as well makes the expression harder to read if the expression gets bigger. It is thus useful to replace it with the expression shown in Listing 2.2.

```
let c : OclAny = a.b.c in
   c.d = c.e
```

Listing 2.2: LetExpression example

2.1.2.4 Type Expressions

TypeExpressions are operations defined on *OclAny* and serve as constructs to perform type checking and type casting operations.

`oclType` returns the dynamic type of the object the operations is applied to. An expression like `x.oclType() = y.oclType()` can be used to check whether the dynamic types of the objects by x and y are equal.

`oclIsTypeOf` returns whether the dynamic type of the object the operations is applied to is exactly the same as the one of the provided *Classifier*. The type of `oclIsTypeOf` is *Boolean*. The expression `x.oclIsTypeOf(Node)` returns `true` if the type of the object denoted by `x` is `Node` and `false` otherwise.

`oclIsKindOf` returns whether the dynamic type of the object the operations is applied to is the same or a subtype as the one of the provided *Classifier*. It works similar to `ocIsTypeOf`, e. g. `x.oclIsKindOf(Node)` returns true if the type of the object denoted by `x` is `Node` or a subtype of `Node`.

`oclAsType` serves as operation performing type casts in OCL and behaves similar to Java type casts. It only changes the static type information used for parsing purposes of the object it is applied and returns the object itself (no modification is performed). User should ensure that the cast actually valid before performing this operation (using either `oclIsTypeOf` or `oclIsKindOf`). If the object does not conform to the supplied *Classifier*, this operation fails when validated and returns invalid. This operation is especially useful in this work when iterating over collections in the target application. Generic type parameters in Java are lost after compilation, meaning that the element type of collections cannot be retrieved during runtime. This means that when iterating over a collection the elements have to be cast to the actual element type in order to access their properties.

2.1.2.5 Model Access

In OCL expressions that access values in the target are called *FeatureCalls*. Generally, there are two types of *FeatureCalls*: *PropertyCalls* and *OperationCalls*, whereas a *PropertyCall* is used to access some property of a model element (e. g. a field of an object) and *OperationCalls* denotes calling an operation or method. The model element on which operation or property call is performed is returned by the source expression. In case of an operation call it must necessarily be an element existing in the model, it may also be an instance of an OCL standard type. The referred property or operation must exist in the static type of the source expression, i. e. the definition of the expression is refused by the parser if the property or operation does not exist in the type of the model element returned by the source expression. *OperationCalls* also contain a sequence of arguments, i. e. the arguments supplied when calling the operation. The types of the arguments must conform to the formal arguments defined by the called operation. Note that operation calls that may cause side effects (alter the models state) are not allowed, e. g. a call like `first.setNext(third)` is invalid and should be reverted by the parser. Unfortunately, we have no way (yet) to find out whether a method in a Java class is side effect free. Therefore, we abort the operation call and return a runtime error if during the execution a modification attempt occurs.

The concrete syntax of a property is similar to a field access in Java, i. e. `source.propertyName`. An operation call is similar to a method call in Java, i. e. `source.OperationName(args)`, whereas `args` is structured as `arg, arg, ...`

Example 2.1.2 (PropertyCall)**.** The expression `next.prev` returns the value of property `prev` contained in the model element returned by the source expression `next`, which is also a property call in this case but could also be some variable. The hierarchy of property calls can be arbitrary deep, as in `next.next.next.next`. Note that `next.prev` does not state the source of the `next` property call. In such a case an OCL automatically prepends the current context of the expression, which is either `self` or the currently accessed element when iterating over a collection (see Section 2.1.2.7).

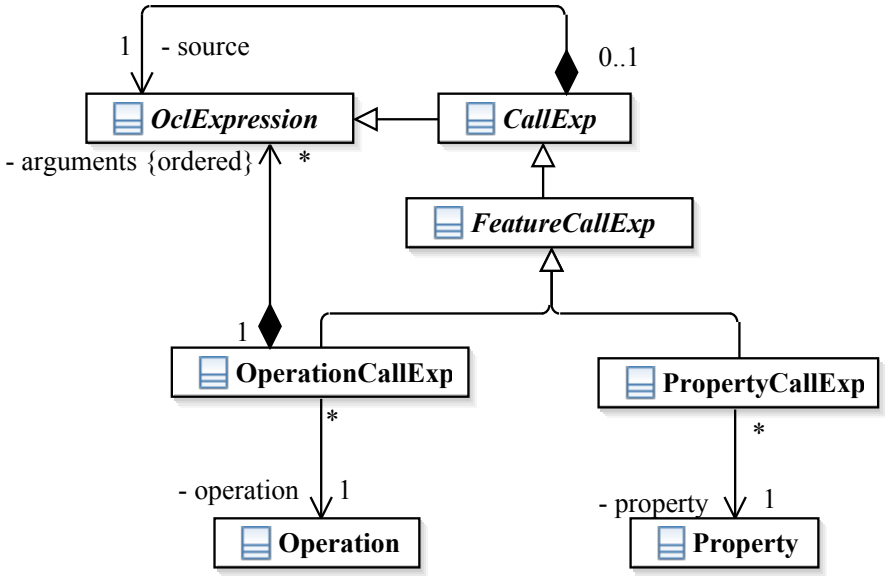


Figure 2.5: FeatureCallExpressions

Example 2.1.3 (OperationCall)**.** The expression `size()` in the invariant shown if Figure 1.1 will call the `size()` method of the currently validated List object (`self`) and yield the result returned by the method call.

2.1.2.6 Conditional Expression

There exists only one conditional construct in OCL, namely the *IfExpression* shown in Figure 2.6. It contains of a condition which is an expression returning a boolean value, and two expression validated based an whether the condition is satisfied or not. Note that since an *IfExpression* is itself a subtype of *OclExpression* it also has a type (it returns a value), depending on the types of the contained *thenExpression* and *elseExpression*. The types of *thenExpression* and *elseExpression* need not necessarily be the same, but they must have at least a common supertype. This common supertype is used as the type of the *IfExpression*. The main difference between OCL and Java is that the else branch must always be present. This is caused by the fact that the *IfExpression* must return some value, which would be undefined if the condition would not be met and no else branch would be present. In Java the then or else branches are not expressions in a strict sense, they are simply a sequence of subsequently executed statements.

Example 2.1.4 (*IfExpression*)**.** We could rewrite *CorrectlyLinked* constraint from our example using an *IfExpression* instead of the implication as shown in Listing 2.3. In this case both the then and the else branch have the same result type, i. e. *Boolean*. Thus, the entire *IfExpression* also returns a boolean value.

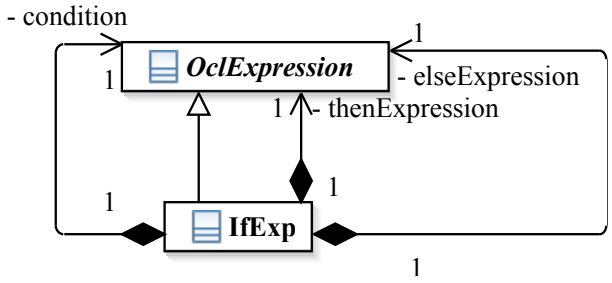


Figure 2.6: IfExpressions

```
if self.next = null then
   true
else
   self.next.prev = self
endif
```

Listing 2.3: Rewritten CorrectlyLinked using if

Example 2.1.5 (If expression with different subtypes)**.** As explained previously the types of the then and else expression may be different as shown in Listing 2.4. This expression may not make much sense or be useful and is for illustration purposes only. Clearly the *thenExpression* returns a model element of type Node, whereas the type of *elseExpression* refers to the type bound to the template parameter E. If E would yet be bound to a concrete type they could actually be equal. But, during the definition of the expression the parameter E is still unbound, meaning that the parser will infer their "nearest" supertype. In this case it happens to be *OclAny*, it is at least ensured that any type that could possibly be bound to E is in fact a subtype of *OclAny*.

```
if self.next = null then
   self.prev
else
   element
endif
```

Listing 2.4: IfExpression with different subtypes

Example 2.1.6 (Using IfExpressions for error handling)**.** An *IfExpression* can be used to handle or "catch" runtime errors during the validation of expressions. If one would want to ensure that an expression does in fact return a boolean value (not invalid) it is possible to use a construct like shown in Listing 2.5. In this example the actual expression is validated and its result assigned to a variable. Afterwards, the *IfExpression* checks whether the result is `invalid` and if so returns `false` or the returned result otherwise.

```
let ret : Boolean = ... in -- some expression resulting in a
    boolean value
if ret = invalid then
   false
else
   ret
endif
```

Listing 2.5: IfExpressions for error handling

2.1.2.7 Loops

As previously explained, OCL is a declarative language and therefore loops in the traditional sense do not exist unlike in iterative languages. But, certain operations defined on collections "iterate" over its contents validate to a result based on its contents. Basically, they work similar to a foreach loop in Java, iterating over each element, validate an expression using the specific and combine it to a single result. Such loop operations can also be compared to *reduce* operations well known from functional languages. Variables introduced by a loop expression work differently than the ones used in *LetExpression*. They are called the iterator and refer to the current element in the collection in each iteration. Defining the type of the iterator is optional (set to the element type of the collection if missing) and even its name can be omitted. If the iterator definition is not present an OCL parser has the responsibility to introduce one by itself, with a unique name. Furthermore, a loop operation alters the current context of the expression, i. e. if a property or operation call occurs inside the loop without specifying the variable it refers to the iterator variable will be pretended instead of `self`. The standard iterator operations are:

ForAll is the standard universal quantifier ($\forall$) and returns a boolean value. It contains a nested expression which also has to return a boolean value. The entire result is true if and only if the nested expression holds for each element in the collection. Listing 2.6 shows an example that checks whether all elements in a collection of integers are in a range between 0 and 99, or with logical symbols$\forall i \in numbers : i \geq 0 \wedge i < 100$. A forAll expression may also use multiple iterators, which will both iterate over the entire collection, effectively iterating over the Cartesian product of the collection. An example for this is shown in Listing 2.7, stating if two persons are considered equal, their names have to be equivalent as well.

Exists is the standard existential quantifier ($\exists$). It works similar to the forAll expression, but in this case it suffices for one element to satisfy the nested expression.

Select can be used the generate a new collection from an existing one, containing each element that satisfies a nested expression. For example the expression shown in Listing 2.8 will first

generate a sequence of integers in the range between 0 and 100 and afterwords generate a new one containing only the even ones.

Iterate is the most basic of loop expressions and the type of the value it returns may be defined by users themselves. Additionally to the iterator another variable has to be defined called the accumulator. An initial value has to be assigned to the accumulator and after each iteration the result of returned by the iteration is assigned to the accumulator except for the final iteration after which the iteration result becomes the result of the iterate expression itself. Due to the generality of the iterate expression the standard loop expression can be expressed by means of an iterate expression. Listing 2.9 shows how the examples from Listing 2.6 and 2.8 can be expressed using iterate expressions.

```
-- let numbers be a collection of integer
numbers->forAll( i | i >= 0 and i < 100)
```

Listing 2.6: `forAll` Example

```
Person.allInstances()->forAll( p1, p2 | p1 = p2 implies p1.name =
    p2.name
```

Listing 2.7: `forAll` Example with multiple iterators

```
Sequence{0 .. 100}->select( i | i.mod(2) = 0)
```

Listing 2.8: `select` Example

2.1.3 OCL translation examples

2.1.3.1 Using a List Index as Part of Constraint

Listing 2.10 shows an example of a constraint written in Java that states that each value in a list of integers must not be greater than its index in the list. This can easily be done by using a for loop iterating over the contents of the list using an index as the current index. As explained imperative loops do not exists in OCL, but it is still possible to translate it into OCL as shown in Listing 2.11. Instead of using a loop one has to create of new sequence of integers containing all the values the loop would iterate over and apply the appropriate collection operation on this newly created sequence.

```
result = true;
for (int i = 0; i < list.size(); i++)
   result &= list.get(i) <= i;
```

```
-- replacing forAll by iterate
numbers->forAll( i : Integer; acc : Boolean = true |
   acc and (i >= 0 and i < 100))
-- replacing select by iterate
Sequence{0 .. 100}->iterate{ i : Integer; acc : Sequence(Integer)
    = Sequence{} |
   if i.mod(2) = 0 then
      acc->including(i)
   else
      acc
   endif
)
```

Listing 2.9: `iterate` Examples

```
return result;
```

Listing 2.10: List Index example in Java

```
Sequence{1 .. list->size()}->forAll(i | list->at(i) < i)
```

Listing 2.11: List Index example in OCL

2.1.3.2 Create a Collection of Attributes

Listing 2.12 shows an example that iterates over a set of people and constructs a new set containing all the last names of the people in the original set. For this purpose OCL defines a special collection operation called collect, as shown in Listing 2.13. The collect operation returns a new collection containing the value of the given property of each element it iterates over. This may also be achieved by using by using an iterate operation with the same behavior.

```
Set<String> lastNames = new HashSet<>();
for(Person p : people)
   lastNames.add(p.lastName)
// do something with lastNames
```

Listing 2.12: Collection of Attributes example in Java

```
let lastNames : Set(String) = people->collect(lastName)->asSet()
   in
-- do something with lastNames

let lastNames : Set(String) = people->iterate( p : Person; acc :
   Set(String) = Set{} |
```

```
   acc->including(p.lastName)) in
-- do something with lastNames
```

Listing 2.13: Collection of Attributes example in OCL

2.2 Java Platform Debugger Architecture

The Java Platform Debugger Architecture (JPDA) [33] defines a set of specifications, tools and APIs to enable the debugging of a program running in a Java Virtual Machine (JVM). To some degree our invariant checker can be considered being some sort of debugger in that it aims at finding errors in a running program's structure enabling us to use JPDA in the first place. Obviously, the main purpose of JPDA is to debug applications written in the Java language itself, although programs written in other languages may be used as well as long as they run in a JVM (e. g. Scala). JPDA is also programming language independent regarding the debugger.

The JVM must implement the Java Virtual Machine Tool Interface (JVM TI) [34], which is used to access the state of the JVM and influence the execution of applications used to implement debugging, monitoring and profiling tools. JVM TI provides a mechanism to make use of Agents able to hook into the running program and retrieve data from the JVM. JPDA enables the debugger and the target application to run in different processes or even host machines. To communicate between the processes JDPA defines the Java Debug Wire Protocol (JDWP) [32] that specifies how the information between a debugger and a target application is exchanged. JDWP only specifies the format of messages, not the transport, i. e. over which medium the messages are exchanged. Usual transports are socket connections over TCP/IP or communicating via shared memory. Common implementations of JVMs provide a native Agent that implements the JDWP. To enable a target application to listen for incoming JDWP connections special start-up parameters have to be provided when launching the application [1]. Generally, a debugger can either connect to an already running application (provided that it has been started with proper arguments), or launch it from scratch establishing an immediate connection.

JDPA also provides a high-level Java API called Java Debug Interface (JDI) [31], which is shipped with the Java Development Kit. JDI covers most of the capabilities needed for debugging provided by JVM TI and aims at simplifying the implementation of debuggers written in Java (e. g. the Java debugger shipped with the Eclipse IDE is built on top of JDI). Our implementation is also built on top of JDI and uses its capabilities to observe / query the target application. The following subsections discuss the basic usage of JDI.

[1] See http://docs.oracle.com/javase/7/docs/technotes/guides/jpda/conninv.html for a detailed description of connection options

2.2.1 Connecting to Target Application using JDI

JDI provides three different kinds of Interfaces that can be used to establish a connection between the debugger and the target application. Generally, all connectors return a mirror of the target virtual machine after successfully establishing a connection.

***LaunchingConnector*:** this connector is used for launching a new application to debug. It essentially works as if the application is started from the operating systems command line with all the necessary launching options provided. The application will be launched on the same host machine as the debugger itself. After successfully launching the application, the debugger is immediately connected to it and the target VM is suspended just before entering the `main(...)` method. The debugger may now issue notification requests (see Section 2.2.2) and afterward resume the application. If the debugger terminates while the target is still running, the application may continue its execution but no methods for reconnecting are supported. See Example 2.2.1 for a short explanation on how this kind of connector may be used.

***AttachingConnector*:** this connector is used for connecting or "attaching" to applications already listening for incoming JDWP connections. Available transport methods depend on the underlying operating system of the host machines of both the debugger and the target application. This connection method is especially useful when debugging an application that runs a different host machine. Example 2.2.1 shows how to use this kind of connector using a socket connection for transportation.

***ListeningConnector*:** This connector works similar to the *AttachingConnector* but the other way around. In this case the debugger waits for an incoming JDWP connection from the target application itself. Available transport methods are usually the same as for the *AttachingConnector*.

Example 2.2.1 (Using the *LaunchingConnector*)**.** Listing 2.14 shows a code snippet how to launch a new application to debug. All necessary arguments must be provided, especially the class containing the initial `main(...)` method to call as well as the class path. The class path must include the location of the class containing the `main(...)` method. This example will launch a new VM and call the `main(...)` method of class `foo.Bar` after resuming the VM.

```
LaunchingConnector connector = null;
VirtualMachine vm = null;  // mirror to launched VM

for (LaunchingConnector c : Bootstrap.virtualMachineManager().
   launchingConnectors()) {
   if (c.name().equals("com.sun.jdi.CommandLineLaunch")) {
      connector = c;
      break;
```

```
    }
}

Map<String, Connector.Argument> arguments = connector.
    defaultArguments();
Argument mainArg = arguments.get("main");
mainArg.setValue("foo/Bar");  // class containing the main(...)
    method
Argument optionArg = arguments.get("options");
String classPath = "...";  // enter required class path here
String options = "..."; // addtional options; e.g. arguments
    provided to main method
optionArg.setValue("-cp␣" + classPath + "␣" + options);

try {
    vm = connector.launch(arguments);
} catch (IOException e) {
    // transport error; handle exception
} catch (IllegalConnectorArgumentsException e) {
    // illegal arguments provided; handle exception
} catch (VMStartException e) {
    // target application terminated abnormally; handle exception
}
...
```

Listing 2.14: LaunchingConnector code snippet

Example 2.2.2 (Using the SocketAttachingConnector)**.** Listing 2.14 shows a code snippet how to connect to an already running application listening for incoming JDWP connections. In this example it is assumed that the target application runs on host with address 10.10.10.10 and is listening on port 8000 for incoming connections. The optional timeout argument given in milliseconds states how the connector should wait for a connection to be established. If no connection can be established within the given timeframe the `attach(...)` method will terminate abnormally and throw an `TransportTimeoutException`. The timeout defaults to 0 meaning that the `attach(...)` method will wait forever until a connection is established.

```
AttachingConnector connector = null;
VirtualMachine vm = null;  // mirror to launched VM

for (AttachingConnector c : Bootstrap.virtualMachineManager().
    attachingConnectors()) {
```

```
    if (c.name().equals("com.sun.jdi.SocketAttach")) {
        connector = c;
        break;
    }
}

Map<String, Connector.Argument> arguments = connector.
    defaultArguments();
Argument hostname = arguments.get("hostname");
hostname.setValue("10.10.10.10");    // target host name or
    address
Argument port = arguments.get("port");
port.setValue("8000");  // listening port for incoming JDWP
    connections
Argument timeoutArg = arguments.get("timeout");
timeoutArg.setValue("0");

try {
    vm = connector.attach(arguments);
} catch (TransportTimeoutException e) {
    // connection timed out; handle exception
} catch (IOException e) {
    // transport error; handle exception
} catch (IllegalConnectorArgumentsException e) {
    // illegal arguments provided; handle exception
}
...
```

Listing 2.15: SocketAttachingConnector code snippet

2.2.2 Notifications

The main purpose of JDI is observing events occurring in the target application. Figure 2.7 shows an overview of available observable events. Most of the events in this figure will not be discussed in detail here since they are either irrelevant for this work or not extensively used. Furthermore their names quite obviously state their purpose. *LocatableEvent* has further subtypes shown in Figure 2.8. The *Location* associated with a *LocatableEvent* states the method and the line number in the applications source code where the event occurred. Note that the line number is only available if the application has been compiled with the appropriate option set.

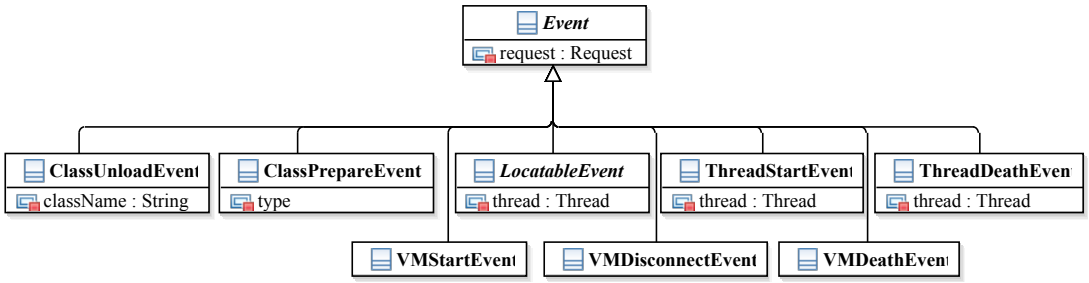


Figure 2.7: JDI event overview

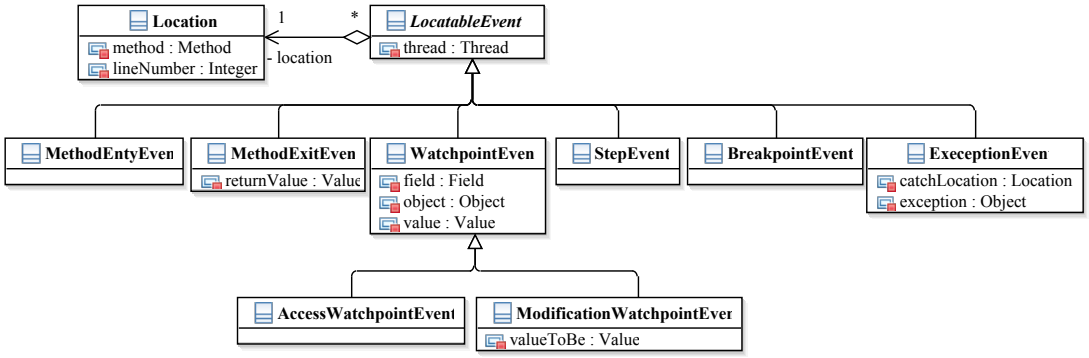


Figure 2.8: Locatable Events

The most important thing to note about JDIs notification mechanism is that nothing will be forwarded to the debugger if not specifically requested beforehand. For example when a debugger wants to observe *MethodExitEvents* it has to explicitly state so by creating a corresponding request. The structure of available requests essentially matches those of the events, i. e. for every event type a request type exists. Such requests may be enabled or disabled at any time. Furthermore, filters can be applied to requests stating that notifications should only be forwarded to the debugger if the event occurs for a given class, field, object, etc. Table 2.5 shows an overview of the filtering options for events used in our implementation.

Whenever certain events occur the debugger may want to suspend the target application to let a user investigate it. Therefore additional to the filtering options the debugger also has to specify a suspend option when creating a request. The available suspend options are:

Event	Filtering options
Class loading / unloading	Class
Method entry / exit	Thread, Class, Object
Field access / modification	Field, Thread, Class, Object
Breakpoint	Location, Thread, Object

Table 2.5: Notification Filtering Options

Suspend None: Does not suspend anything. Anyhow, the event notification must be processed by the debugger although the target application proceeds to run in parallel. Be aware that if any values are accessed in the target application by the debugger, they may have already changed in the mean time.

Suspend Thread: Only suspends the thread in the target application in which the event occurred. The thread is locked in the state when the event occurred. All other threads proceed to run normally.

Suspend VM: Suspends the virtual machine of the target application, i. e. all existing threads in it.

No matter what suspend options had been chosen for requests, JVM TI ensures that all event notifications are forwarded in the order they occurred.

2.2.3 Accessing Values in Target Application

Additionally to observing a target application a debugger usually wants to access current values of fields and method variables during execution. Whenever a debugger suspends the target application it is usually because users want to investigate the current state of the application. In our case there is no human user investigating the application, they get "replaced" by the automated invariant checker (In a way the invariant checker can be considered as the user). An important thing to note is that JDI does not allow to directly access values in the target. The debugger only holds Mirror objects, representing the actual values. Whenever the debugger accesses an attribute or calls a method on such a mirror, JDI communicates with the target application via JDWP to retrieve the actual information. Note that due to this fact every retrieval of information requires communication overhead. Value types known by JDI are shown Figure 2.9 and additionally *PrimitiveType* has subtypes for each primitive type that exists in Java (`int`, `boolean`, ...). Those subtypes provide methods to retrieve the actual values associated with the mirror. *ObjectReference* represents a reference to some object and provides methods to access / modify the fields of the object and invoke methods. Some reference types are treated specially, which are strings (instances of `java.lang.String`), arrays (reference to an array), and class objects (instances of `java.lang.Class`). *StringReferences* allow to retrieve the actual string as an instance of `java.lang.String`, and *ArrayReferences* provide a way to access the values contained in the array.

JDI also allows to retrieve reflective type information (e. g. querying the type of a value). The available information closely resembles the one provided by the `java.lang.reflect` package. This information is also accessed using *Mirror* objects, also causing communication overhead.

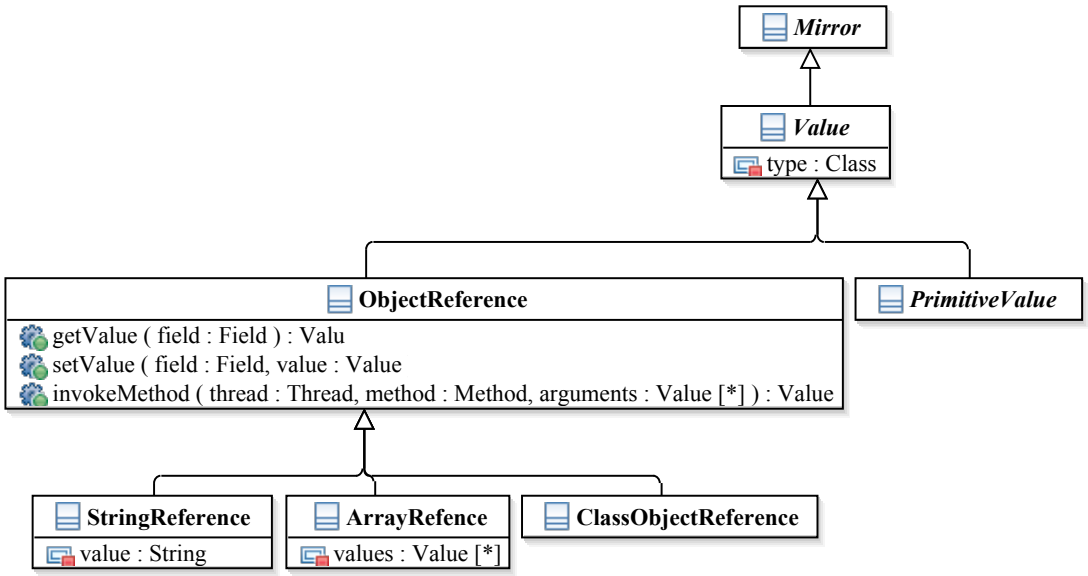


Figure 2.9: JDI Value Types

Chapter 3

Approach

In this chapter we describe how our invariant checking mechanism works in detail. Section 1.2 gave a general overview of what our approach deals with. Section 3.1 shows what kind of runtime structures we consider and introduces the data structures we use internally to keep track of invariants, their current results and any information we need to recognize if some invariant needs to be revalidated. Sections 3.3 and 3.4 illustrate what kind of events we observe in a running systems and how those events are used for invariant checking respectively.

3.1 Basic Definitions

For simplification purposes when describing how our approach works we define a simplified object orientated system as shown in Figure 3.1. We do not deal with concepts such as interfaces, primitive types etc. Each type corresponds directly to a class. A class defines a set of methods and fields, which are passed down to its subclasses and inherits those of its superclasses. Although the definition of a field belongs to a class, its concrete value depends on the object it belongs to. The somewhat obvious contracts for accessing fields and invoking a method of an object are formalized in Listing 3.1.

As already pointed out in Section 1.2, we used the Java language for the reference implementation, which provides more concepts than the ones shown in Figure 3.1. Note that all concepts of Java are all fully supported, most notably Interfaces and Primitive Types.

Our approach is based on the ModelAnalyzer approach by Egyed et al. [10, 36, 9] and uses similar data structures, especially the notion of scope elements and rule instances. Simply put those elements are used to keep track of invariants in their results the current results of. Figure 3.2 shows the structure of those concepts and how they are related. The upcoming definitions will give a detailed description.

Definition 3.1.1 (Expression)**.** Our approach uses its own language and validation engine. Although OCL is used as the constraint language, only its concrete syntax is used. The abstract

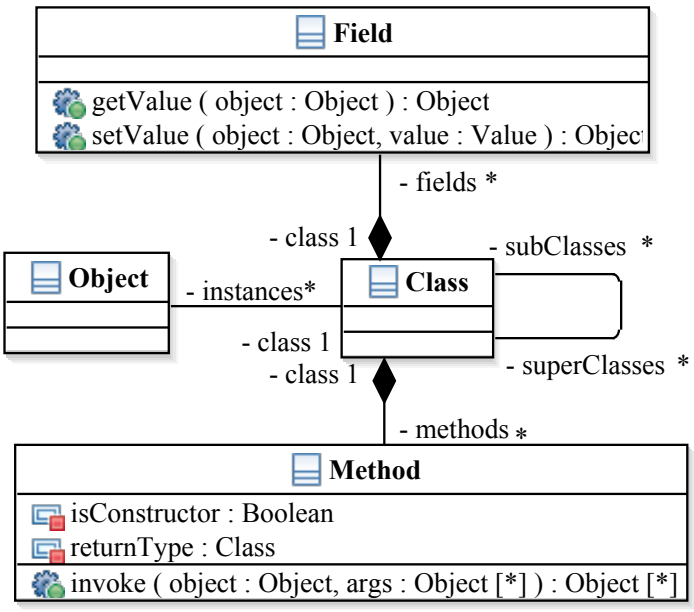


Figure 3.1: Simplified object orientated structure

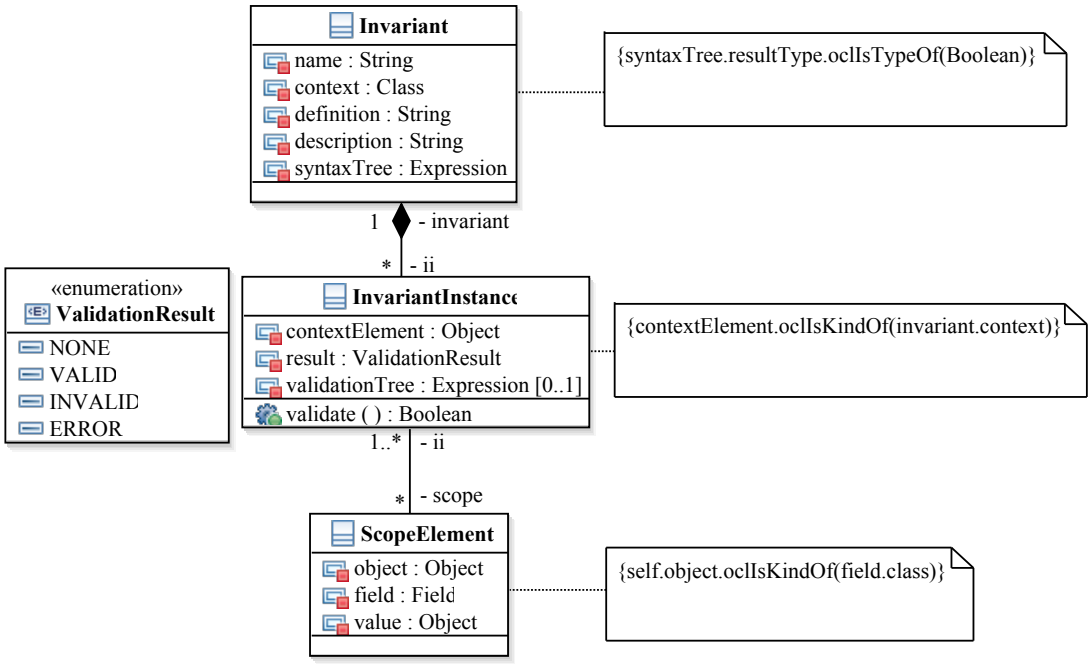


Figure 3.2: Structure of used data elements

```
context  Field::getValue(object : Object) : Object[*]
pre:  object.oclIsKindOf(self.class)
post: result->forAll(x | oclIsKindOf(self.type))

context  Field::setValue(object : Object, value : Object)
pre:  object.oclIsKindOf(self.class)
   value.oclIsKindOf(self.class)

context  Method::invoke(object : Object, args : Object[*]) :
   Object[*]
pre:  object.oclIsKindOf(self.class)
   args->size() = formalArguments->size()
   Sequence{1..args->size()}->forAll(i | args->get(i).oclIsKindOf
      (formalArgs->get(i)))
post: result->forAll(o | o.oclIsKindOf(returnType))
```

Listing 3.1: Method contracts for accessing field values and invoking methods

language used has been inspired and hence available expressions are similar to the OCL ones described in Section 2.1.2. From a user point of view this behavior is entirely irrelevant, the parser takes care of mapping the definitions of OCL expressions to the ones actually used. The expression constructs used in our approach do not just represent the abstract syntax, each expression provides an operation to validate itself (given proper context), meaning that there is no additional dedicated validator involved. Additionally they store the calculated result to perform bottom-up validation for further revalidations in the future if desired (see Section 3.4.4).

Definition 3.1.2 (Publicly visible State)**.** In a single threaded application an object is in a publicly visible state whenever a method of this object can be called from the outside (i. e. a method in another object). It transitions out of a publicly visible state as soon as a method of this object is called from the outside and the object controls the execution flow of the program. It transition back into a publicly visible state whenever it gives up control of the execution flow of the program, i. e. returning from a method of the object to a method which does not belong to the same object. As far as invariant checking is concerned, invariants have to hold whenever an object transitions into or out of a publicly visible state. Callers assume that the object is in a valid state when using its functionality and that it is still valid after the operation has been performed. It is not necessary to check invariants every time a method on an object is called from the outside, it can be relaxed to only check if it is in a publicly visible state for the very first time (after its construction) or whenever it transitions into one, due to transitivity. Other approaches tend to treat invariants as being part of the precondition of every public method, which may work in most cases but introduces the problem that they now also have to hold between recursive method calls or simply when an object invokes another public method on itself. This means that intermediate invalid states between the method calls are treated as errors, which is entirely irrelevant from a global perspective since no other object can use its

functionality anyhow. The important thing is that after the last call (returning to a method of another object) it is ensured that the object is in a valid state.

This definition does fully apply to multi-threaded applications where public visibility is thread-wise and depends on synchronization. This work does not deal with this problem and treats every application if it were single-threaded, which may lead to errors and incorrect invariant validation results.

Definition 3.1.3 (Invariant)**.** As previously mentioned we only consider class invariants. A class invariant is simply a logical expression that has to hold for every instance or object of a given class, or rather any kind of type, whenever it is in a publicly visible state. We call this type the invariant's context. Simply put, an invariant limits the set of valid states of an object. An object is in a valid state whenever all invariants defined for its type hold, i. e. validate to true. In an object oriented language a type may also have supertypes. All invariants defined for the supertypes are inherited, meaning that not only the ones defined for its specific type have to hold, but also those defined for the supertypes as well. The definition of the invariant is the actual constraint that must hold for every instance of its context. Currently we use OCL as the front-end constraint language, but due to the fact that it only used for defining the constraints (as described in Definition 3.1.1) it could be changed to another language quite easily. The concrete syntax is only used for the definition of invariants and a front-end representation of the constraint. After its definition a parser will immediately convert this into an abstract syntax tree (AST) based on our expression concept. Figure 3.3 shoes the syntax tree corresponding to the *CorrectlyLinked* invariant introduced in Figure 1.1. Tree nodes depicted using an oval shape indicate expressions validating to a boolean value, whereas diamond shapes are used for variables or field accesses returning the accessed object (always a Node in this case). The variable `self` is used in the same manner as in OCL, it refers to the context element the constraint is applied to.

The formal definition of an invariant may become quite complex and a different developer might not understand its meaning or purpose at first glance. Thus, we provide an additional description field which has no formal meaning, but should be used to enhance the understandability why the invariant exists and what it is used for. Additionally an invariant may have a name for identification purposes. An invariant is related to a set of invariant instances (see Definition 3.1.4). This set may potentially be empty if no instances of the invariants context exist yet.

Further examples will refer to an invariant in the format $Name[context]$, e. g. $CorrectlyLinked[Node]$ (Table 3.1 shows the complete definition of this invariant).

Definition 3.1.4 (Invariant Instance)**.** An invariant instance is an invariant applied to a specific object. While the invariants context states the type for which it has to hold, the invariant instances context element refers to the concrete object. This context element has to be an instance of the context of the invariant. At any point in time during the execution there exists exactly one invariant instance per combination of invariant and object. An invariant instance

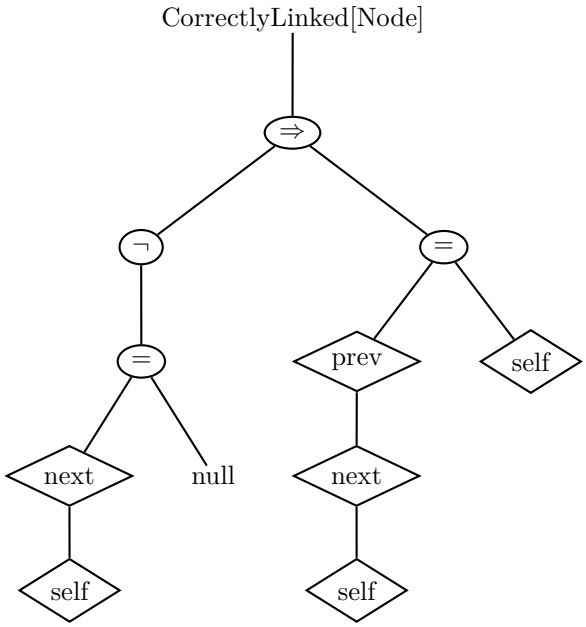


Figure 3.3: Syntax Tree of invariant CorrectlyLinked

name:	CorrectlyLinked
context:	*Node*
definition:	`next <> null imples next.prev = self`
description;	Checks whether the back pointer of the next Node is properly set
ii	∅

Table 3.1: Invariant example

is related to a set of scope elements (see Definition 3.1.5). This set of scope elements is empty in two cases: either the invariant instance has been newly created and not validated yet, or no field is accessed during the validation (indication of a useless invariant since the systems state is not queried at all).

Further examples will refer to an invariant in the format $InvariantName(ce, r, S)$, where ce is the context element, r the current result and a set of scope elements S, e. g. $CorrectlyLinked(first, \texttt{Valid}, \{(first, next), (second, prev)\})$. We may also just use $InvariantName[ce]$ if we do not care about the result or scope. The result of the validation can have one of four possible values:

None: there is no result, meaning that the invariant has been properly initiated but not yet validated. An invariant has this type of result only for a short period.

Valid: the constraint is valid for the given context element, i. e. validating the expression associated with the invariant returned true for the invariant instance's context element.

Invalid: the constraint is invalid for the given context element, i. e. validating the expression associated with the invariant returned false for the invariant instance's context element.

Error: Indicates the occurrence of a runtime error during the validation, e. g. accessing fields of null values, division by zero, etc.

Definition 3.1.5 (Scope Element)**.** A scope element is an element in the target environment that contributes in some way to the outcome of an invariant instance's validation. Since class invariants are used to specify the set of valid states of an object and the state is given by its fields values, a scope element is a field of a specific object most of the times. In case of class fields (static fields in Java) the reflective class object is used instead of a concrete instance. A scope is thus a tuple consisting of an object and a field and we will further refer to them in the format $(object, field)$. A scope element for a given object and field combination may exist only once, meaning that one scope element may be used by multiple invariant instances. It comes into existence whenever a fields value is accessed for the first time during an invariant instance's validation. Additionally, we have to assure that it is removed if it is no longer used by any invariant instance. This is done by emptying the set of scope elements prior to the validation.

Definition 3.1.6 (Changes)**.** As mentioned in Section 1.5, our approach reacts to changes occurring in the target system. The state of a system can change through the creation or deletion of an object, or the modification of one of the object's fields (scope element). Possible changes and their structure are shown in Figure 3.4. Essentially a change is a 3-tuple, describing the type of change, the changed object and the changed field respectively. The field may be missing (`null`) if the change is not a modification. For example $(\texttt{MOD}, third, next)$ denotes that the field *next* of object *third* has been modified. The upcoming sections describe how we make use of those changes in our invariant checking approach.

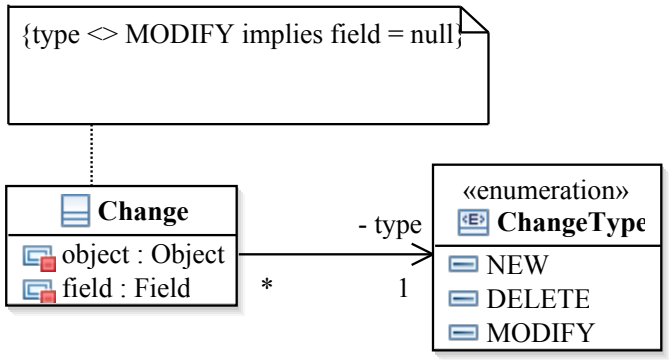


Figure 3.4: Possible changes

3.2 Architecture

Figure 3.5 shows an overview of the approach, which closely resembles the one of the actual implementation (for implementation details see Chapter 4). The following subsections discuss the responsibilities of each component and how they are connected, or more specifically what information they exchange. The arrows depict the directions of information flows between these components. Implementation details of those components are discussed in Chapter 4.

3.2.1 Observer / Wrapper

This component serves as a layer on top of the target runtime. It is primarily responsibly for two tasks:

1. Observing the target system forwarding notifications about occurring events. The event notifications necessary are further explained in Section 3.3.

2. Providing a way to access data needed for invariant validation. The data it has to be able to provide is the same as if the invariant checking mechanism would run in the target runtime. This is mainly, but not not limited to, accessing values of object fields, retrieving reflective type information and calling methods in the target runtime. How this is implemented is essentially irrelevant, as long as this information can be gathered somehow. In the simplest case this component could be omitted entirely if the invariant checking would run in the same environment as the target system and would have direct access to its data structures.

Additionally to those main tasks the wrapper must provide a way to connect to an already running target system or start it from scratch.

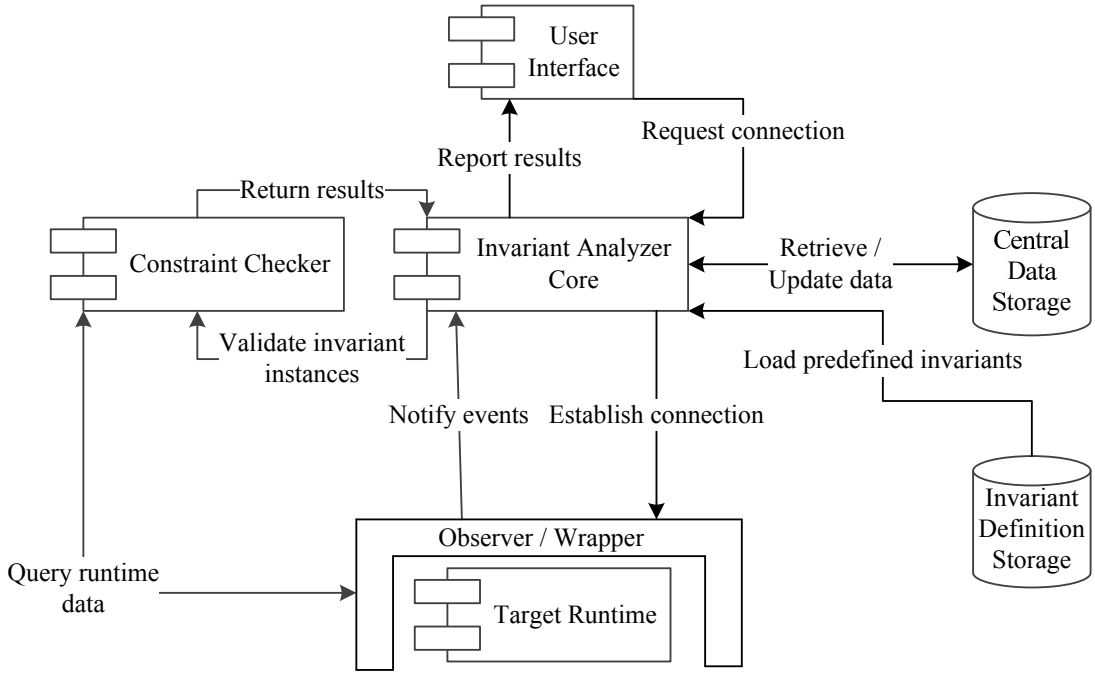


Figure 3.5: Architecture Overview

3.2.2 Constraint Checker

This component's purpose is to check actual invariants instances. It is currently based on our expression concept, meaning that it is not an actual component rather than an expression associated with an invariant knows how to validate itself. Theoretically, this could easily be exchanged by using a dedicated constraint checking component, as long as it yields the same results (e. g. using a proprietary OCL checking engine). In our case the result is not only the truth value associated with the actual validation of the constraint, but also the scope elements accessed during the validation. Most proprietary engines will not be able to provide this additional information, but could still be if the fields accessed in the target runtime would also be observed.

3.2.3 Central Data Storage

This is simply a data storage that holds all required information regarding invariant checks, i. e. invariants, invariant instances, scope elements, etc. It must provide an interface such that the core component is able to retrieve and update all necessary data. In the following algorithms and examples the current set of invariants is referred to as I, invariant instances as II and scope elements as SE.

3.2.4 Invariant Definition Storage

This is just an "offline" convenience storage and is not required for the core functionality of our approach. Once a user defines an invariant she may not want to do this step over and over again every session. The definition storage thus provides a way to save the entire definition of an invariant (including non-functionally required fields such as the description) for later use. This could be implemented in many different ways ranging form simple text files to a complex database systems.

3.2.5 User Interface

Users need some way to interact with the invariant checker. Our user interface provides the following features:

Invariant definition: User are given an editor providing means to provide all data necessary to define the invariants.

Visualizing validation results: Views showing the results of invariant validations (including scope elements) and enabling further inspection. Although it may not be necessary to provide immediate feedback of all invariant validations (sometimes simple logging and later analysis of the results may be sufficient) it is still a nice thing to have.

Establish connection to target runtime: Whether the target environment is already running or started from scratch users need to be able to configure certain connection parameters and initiate the connection. users are also able to cut this connection at any time, effectively stopping all invariant checking.

3.2.6 Invariant Analyzer Core

The core centralized component takes care of the interoperability of the other components, i. e. telling them what to do. Each of the other components separately is pretty worthless without interconnections among them, e. g. the constraint checker has no use without knowing what to check and what to do with the results. We did not want connect the other components directly since each one could be used in other environments as well. The main tasks of the core component are as follows:

- Instructing the observing component which notifications are relevant. This is derived from the outcome of previous invariant validations (especially accessed scope elements).
- Dispatching the required invariant checks to the constraint checker. The core component analyzes incoming change notifications, determining which invariant instances may be affected by them or if the creation of new ones is necessary. It will then instantiate invariants if necessary and order the constraint checker to validate those invariant instances. It

may also delete certain invariant instances entirely if it comes to the conclusion that they are no longer needed, i. e. the object it relates to does no longer exist.

- Updating the user interface. Strictly speaking, it will not actually update the user interface directly since we follow a strict Model-View-Controller (MVC) principle. But, somehow the user interface needs knowledge whether some data changed and update the views, which the core component takes care of.

- Receiving modifications form the user interface. The interaction between the user interface and the core component is not just in one direction. A user may of course modify specified invariants, add new ones etc. in which case appropriate processing of these events is required.

3.3 Observing the Runtime Environment

As previously mentioned our approach reacts to changes in an observed run time environment. The core component processes notifications of events in the target application, creates a *Change* object if necessary and stores it for further processing. Most common changes are field assignments and the creation / destruction of objects. Those changes do not necessarily trigger the validation of an invariant instance immediately. Thus, we simply store this information for further processing later on, described in Section 3.4.

3.3.1 Creation of New Objects

Since we use our notion of invariant instances, we need to keep track of the objects existing in the target environment. Of course not every object creation is relevant in our case. We only need to be informed about new objects that are instances of a class for which an invariant exists in the invariant repository, i. e. `Invariant.allInstances()->collect(context)->exists(context | object.oclIsKindOf(context))`

Generally, invariants only have to be satisfied if the object has been fully instantiated. Directly after the creation of an object, i. e. no fields have been initialized, most invariants would most certainly fail. Thus, no invariant check will be triggered, although we still need to be aware that this change occurred. At this point, it may be valid to create an invariant instance for all invariants whose context matches the objects class. We decided not to do this since it suffices to create the instances when the actual validation is triggered.

3.3.2 Scope Element Changes

Usually, in an object orientated system, the state of an object will change over time. Whenever the value of a scope elements changes the result of an invariant validation may change as well. Since intermediate invalid states are allowed (as long as the object is not publicly visible) it is

not desired to trigger a revalidation right away. Ordinarily those changes are assignments to fields of an object (scope element). The type of a field may also be an array or a collection. In those cases we also need to recognize whenever its contents changes.

Example 3.3.1. Consider we would add a method link to the definition of the Node class, like shown in Listing 3.2. If one would make a method call `third.link(n)`, after the statement `next = n` we recorded that the scope element $(third, next)$ changed. Triggering the revalidation of the *CorrectlyLinked* invariant would indicate a violation, which is in fact not the case since object third is not publicly visible yet. After the `n.prev = this` statement, this violation will resolve itself, showing the need of intermediate invalid states. Anyhow, since without actually validating the invariant at some point, we can not certain that the violation is resolved. Thus, we still record the changes: $\{(\texttt{MOD}, third, next), (\texttt{MOD}, n, prev)\}$.

```
void link(Node n) {
   next = n;
   n.prev = this;
}
```

Listing 3.2: Link method

3.3.3 Destruction of Objects

In an object orientated environment, objects have a limited lifetime. Although the disappearance of an object will not change the outcome of any invariant instance, it may cause an invariant instance to become obsolete. There are different reasons that cause the destruction of an object. This depends on whether the object had been allocated locally on a methods stack frame or globally on the heap and whether or not the runtime environment makes use of memory management with a built-in garbage collector. We do not care why the destruction of an object occurs, we just need to be informed that such an event happened, so we can remove associated invariant instances.

3.3.4 Observing Operation States

An operation or method has no explicit state per se, but we can indirectly treat the fields accessed during the execution of an operation as its state. Imagine an invariant uses an instance method in its definition e. g. the invariants depending on the `size()` method in our linked list example. The current size of the list could be stored in a separate field, or calculated each time depending on the implementation. In such a case we need to keep track of any fields `size()` may access. A change introduced to one of those fields may cause the method to return a different value and therefore possibly change the result of the invariant instance. Since it is impossible to know a priori which scope elements are accessed by a particular method, we

observe the execution of the method itself. In particular we record each field access of a method and add all of those to the scope of the currently validated invariant instance.

3.4 Invariant Checking Mechanism

Observing changes performed on the data structures of the target system has no use without proper context. This section shows how the observed changes are used to trigger the (re)validation of invariant instances if necessary. Simply put, in addition to observing the execution of a system, we also observe the validation of the invariants themselves. For every field access we construct a scope element and add it to the scope of the currently validated invariant instance. This information can then later be used to decide whether the revalidation of an invariant instance is necessary. Additionally, it will be discussed how checking a particular invariant instance works in detail.

3.4.1 Creation of New Invariant Instances

Previously, we stated that due to the mere creation of an object no invariant instance will be created. But, this needs to happen eventually, at least after the initialization of the object is completed. We know that the instantiation of an object is completed at the end of a constructor method. In case a constructor calls other constructors of the same object (especially ones defined in a super class), we need to ensure that we only react to the last return in this chain of constructor calls. After complete initialization we collect all invariants defined for its class and superclasses. The collected invariants are immediately instantiated for the previously created object and validated.

Example 3.4.1 (Creation of a new `Node` object)**.** Imagine during an execution that uses our linked list example at some point a new `Node n` is created. After the instantiation is complete, we collect all invariants and instantiate them as well, which are *CorrectlyLinked* and *ElementReferenced* in this case, lets call them $CL[n]$ and $ER[n]$ respectively. After initialization their results are $CL(n, \texttt{None}, \emptyset)$ and $ER(n, \texttt{None}, \emptyset)$ We validate those invariant instances and store their scope. Since initially the value of the field `next` is `null` and our constraint checking engine uses short-circuit evaluation, the `prev` field will not be accessed. After the evaluation is completed, we store the scope of the invariant instances and report the result. In this example the built-up scope and results would be $CL(n, Valid, \{(n, next)\})$ and $ER(n, \texttt{Valid}, \{(n, element)\})$. It is noteworthy that the invariant instance $ER[n]$ must only be validated after initialization. One can clearly see from the source code that the scope element n.element is never subject to change and no other other could possibly be accessed due to the invariants definition. Since our approach will never recognize a change for this scope element, it remains a one time validation. Other approaches may not recognize this behavior or perform this optimization, resulting in checking the invariant every time the object transitions into a publicly visible state.

Algorithmus 1 Analyzing Changes and dispatching Invariant Instance validations

Input: a set of changes CS

$II_e \leftarrow \emptyset$ {the set of invariant instances to validate}
$II_c \leftarrow \emptyset$ {the set of newly created invariant instances}
$II_r \leftarrow \emptyset$ {the set of removed invariant instances}
for all $change \in CS$ **do**
  if $change.type =$ `MOD` **then**
    for all $se \in SE$ **do**
      if $se.object = change.object \wedge se.field = change.field$ **then**
        $II_e \leftarrow II_e \cup se.iinstances$
      end if
    end for
  else if $change.type =$ `NEW` **then**
    for all $i \in \{x \in I \mid change.object.type \preceq x.context\}$ **do**
      instantiate new invariant instance ii of invariant i with context element $change.object$
      $II_c \leftarrow II_c \cup \{ii\}$
    end for
  else if $change.type =$ `DELETE` **then**
    for all $ii \in \{x \in II \mid x.contextElement = change.object\}$ **do**
      $II_r \leftarrow II_r \cup \{ii\}$
    end for
  end if
end for
$II_e \leftarrow II_e \cup II_c$
$II \leftarrow II \cup II_c$
$II \leftarrow II \setminus II_r$
report creation of invariant instances II_c
report removal of invariant instances II_r
for all $ii \in II_e$ **do**
  validate invariant instance ii and update its scope
  report new result and scope of ii
end for
perform scope clean-up

3.4.2 Triggering Revalidations

Whenever an object transitions from a non publicly visible state into a publicly visible one (usually happening after exiting a public method), we trigger a revalidation of the existing invariant instances. This works incrementally, meaning that at it does not trigger the revalidation of all existing invariant instances. Essentially, the revalidation has two separate steps: collecting the relevant invariant instances and actually reevaluating them and updating their scope. Collecting the invariant instance for revalidation, instantiating new ones and removing irrelevant ones is done by analyzing the previously collected changes and reacting accordingly, as outlined in Algorithm 1.

Example 3.4.2 (Unchanged scope)**.** Consider the running program is at the end of the method call `third.setPrev(first)`, in the initial state of our list example shown in Figure 1.2a. At this point we collected the scope element changes $\{(\texttt{MOD}, third, prev)\}$. We now collect all invariant instances, which is only $CL(second, \texttt{Valid}, \{(second, next), (third, prev\})$ in this case. After revalidating this invariant instance it will now return `false`, i. e. our approach detected an invariant violation since `second.next.prev <> second`. However, in this case the scope remains unchanged, it still accesses the fields `{second.next, third.prev}`. Meaning the invariant instance now looks like $CL(second, \texttt{Invalid}, \{(second, next), (third, prev\})$

Example 3.4.3 (Updated scope)**.** In the previous example the result of the invariant instance changes and the scope did not. Obviously, the scope of an invariant instance is also subject to change over time, commonly caused by modifying collection contents as well due to the use of short circuit evaluation. The initial scope of $CL[third]$ contains only $(third, next)$. Now consider the node n created in Example 3.4.1 and we make a call to `third.setNext(n)`. This would of course cause the invariant instance $CL[third]$ to be revalidated. Since the next pointer is actually set now, it would also validate the `next.prev = self` part, causing the invariant instance to both fail and update its scope to $CL(third, \texttt{Invalid}, \{(third, next), (n, prev)\}$.

Note that leaving a public method does not necessarily trigger invariant validations. A fair amount of methods do simply query the query the state of an object without actually modifying it ("getter" methods). In such a case whichever invariant did hold beforehand will hold afterward and vice versa. Other approaches offer the ability to mark such methods as simple queries, instructing it to not generate / perform any invariant checks after said method (see Chapter 6). Using such markers is another possibly source for human introduced errors. During the evolution of a software system a simple query may become on operation that does change the state of an object. Not removing the markers in this case may cause errors to get unnoticed. Using our approach this is not required, it will be detected automatically during runtime. Since no changes were recorded during the methods execution, no invariant checks will be triggered.

3.4.3 Scope Clean-up

After performing the revalidation of the necessary invariant instances, their scope may have changed. This means that the global scope has to be updated as well and the invariant instances each scope element refers to, as shown in Algorithm 2. This is done by iterating over the set of scope elements and invariant instances. The set of invariant instances a scope element *se* refers to is set to the union of all invariant instances which accessed this scope element. All scope elements accessed by any invariant instance are added to the global scope to ensure that it contains newly accessed ones. If after the iteration the set of invariant instances the scope element refers to is empty, it will be removed from the global scope.

Algorithmus 2 Updating global Scope

for all $se \in SE$ **do**
  $se.ii \leftarrow \emptyset$
  for all $ii \in II_e$ **do**
    $SE \leftarrow SE \cup ii.scope$
    if $se \in ii.scope$ **then**
      $se.ii \leftarrow se.ii \cup \{ii\}$
    end if
  end for
  if $se.ii = \emptyset$ **then**
    $SE \leftarrow SE \setminus \{se\}$
  end if
end for

3.4.4 Constraint Checker

The constraint checker is an adaption of the one developed for the ModelAnalyzer [38] and this section shows how it is used to validate particular invariant instances. This work does not use a proprietary OCL validator to check the invariants defined as OCL constraints, instead a customized version of the constraint checker implemented for the ModelAnalyzer is used. The specialty of this constraint checker is, that it defines its own abstract syntax, which has been influenced by OCL, but does not define a concrete syntax. Implementations may theoretically use the concrete syntax of any language, but have to ensure that it is properly parsed into the used abstract syntax. This constraint checker has originally been designed to check constraints imposed on models written in UML. In the line of this work an existing OCL parser has been adapted that is able to create the proper structures and can deal with Java types (especially the type mappings, as explained in 2.1.1.2).

When an invariant instance is validated, the syntax tree of the invariant will be copied and attached to the instance. We call this tree the validation tree. Its structure is essentially equal to the syntax tree, but expressions may store their result for later use and expression accessing the values of scope elements (field access, method calls, ...) hold a reference to those scope elements. Figure 3.6 shows an example validation tree of the invariant instance $CL[first]$. As a reminder the definition the invariant in OCL is: `next <> null implies next.prev = self` The edge labels in the tree depict the current result of the expressions and references to scope elements are shown as dashed lines. A specialty of the constraint checker is that it offers two different modes for checking constraints, a top-down and a bottom-up one. Users may define what mode should be used per invariant individually. Using bottom-up validation causes most invariant instances to get validated faster, while requiring more memory to keep track of the validation trees. When a users expects an invariant to be validated quite frequently it is recommended to use bottom-up validation to benefit from the speed-up and on the other hand use top-down validation to save memory.

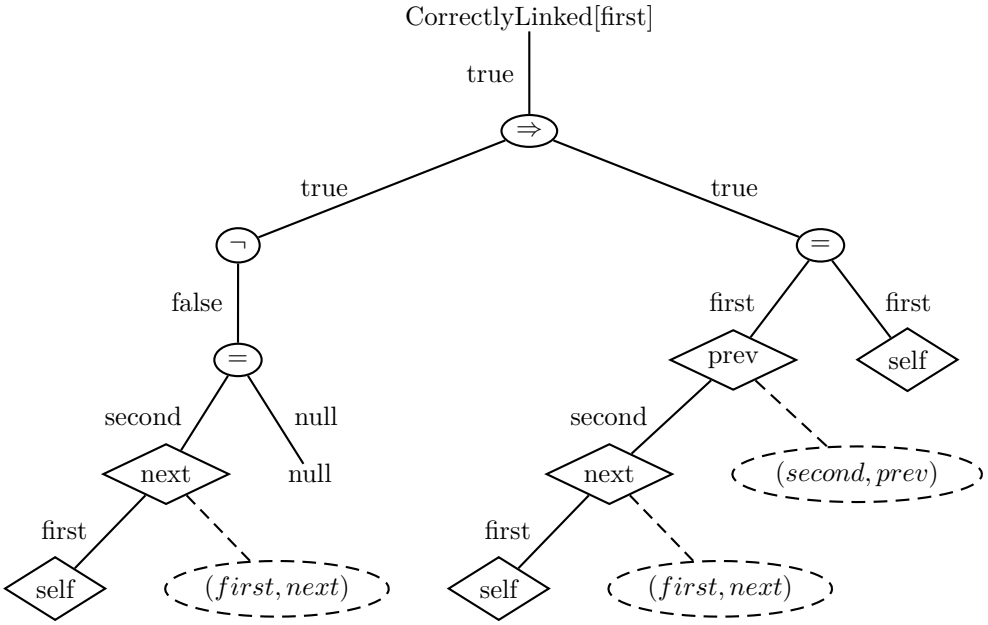


Figure 3.6: Validation Tree of $CL[first]$

3.4.4.1 Top-Down Validation

This mode works like traversing a tree beginning at its root down to the leaves, similar to a depth-first search algorithm. Each expression will first validate its children (subexpressions) and afterward combine their results to its own one. For example an expression expressing equality will first validate the left and the right side and afterward compare whether the two results are equal. It will also make use of short circuit evaluation, meaning the right hand side will only be validated if necessary. Figure 3.7 shows the validation tree of invariant instance $CL[third]$. In this case the right hand side of the implication has not been validated (indicated by using dotted lines) since it is already guaranteed to be true after validating the left side. All logical expression support this behavior. When using this mode, the entire validation tree will be discarded (removed from memory) after the validation is completed.

3.4.4.2 Bottom-Up Validation

When this mode is set, evaluating an invariant instance will not begin at its root expression if there is already a validation tree present. Instead, validation begins at expressions referring to changed scope elements. Those expression will query the new value of the scope element and afterward instruct its parent expression to revalidate itself. An expression will not validate any of the children that have previously been validated, but instead use the result of a previous validation (but use the new results pushed upwards). Whenever the new result of expression differs from the old one, revalidation of the parent is triggered.

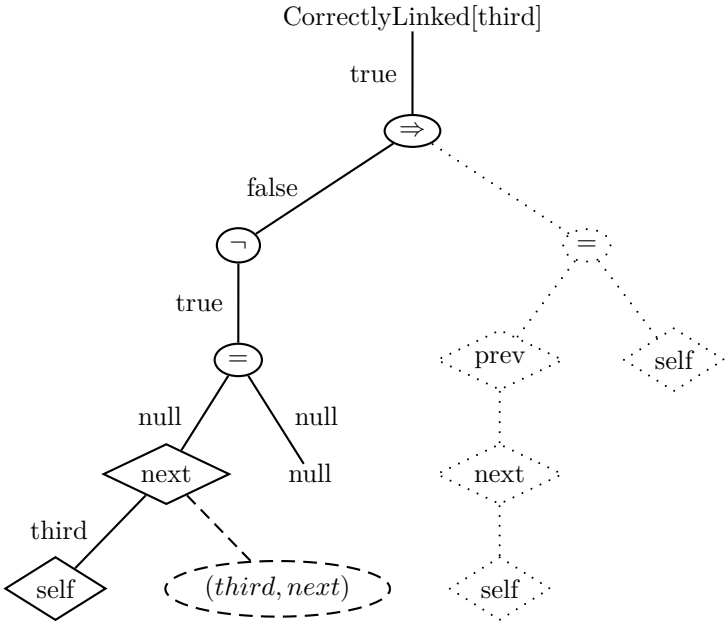


Figure 3.7: Validation Tree of $CL[third]$

Expressions iterating over collections benefit most from this mode since it performs loop unrolling. E. g. a universal quantifier is not treated as a single expression, instead it is broken down into a series of conjunctions. This means that the time necessary to validate the expression will not grow with the number of elements it would iterate over. Instead, whenever the expression is revalidated, only the subexpression in the series of conjuntions which is affected by the change of a scope elements value will be revalidated.

Example 3.4.4 (Partial revalidation)**.** Consider the validation tree of invariant instance $CL[first]$ in the state shown in Figure 3.6. Now we change the value of scope element $(second, prev)$ to third (by calling second.setPrev(third)). This causes the invariant instance $CL[first]$ to be revalidated, but now instead of revalidating it entirely, we start at the expression referring to the scope element $(second, prev)$ and work our way upwards. Figure 3.8 shows which expressions need to be reevaluated in this by using solid lines and dotted lines for untouched ones . Since the value of $(second, prev)$ changed too, the revalidation of the equals expression is triggered since it also may yield a different result now. Since its result actually did change, it will again trigger the revalidation of the implication, causing the entire invariant instance to change its result. The clear advantage is that it was only necessary to revalidate 3 out 10 expressions to detect the violation.

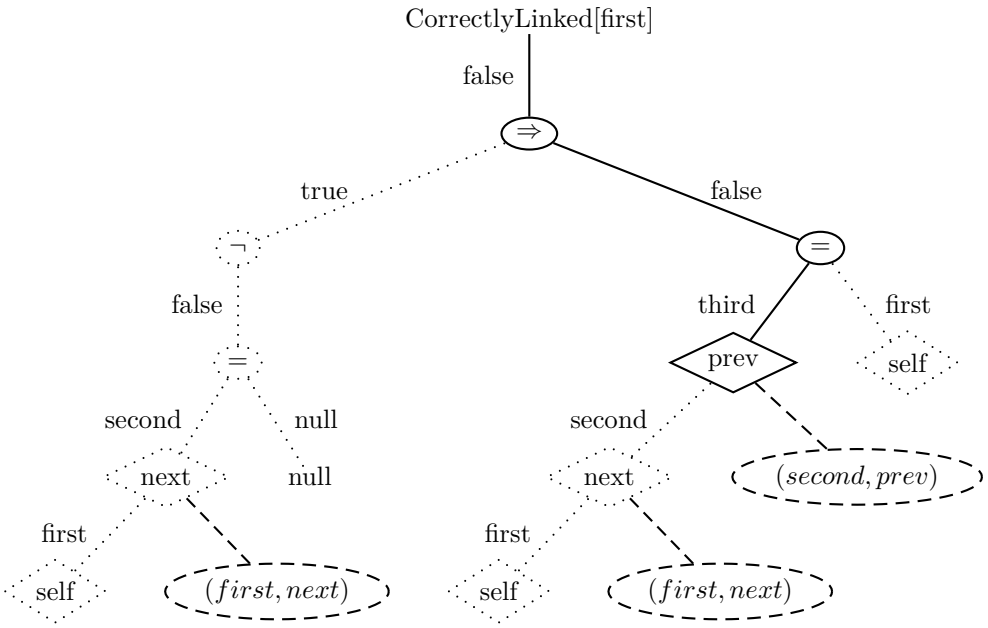


Figure 3.8: Revalidation of $CL[first]$

Chapter 4

Implementation Details

Our prototypical implementation has been fully implemented in the Java programming language and is able to observe and check invariants for arbitrary programs running in a Java Virtual Machine (JVM). It has been built on top of the Eclipse Rich Client Platform and is fully functional. Essentially, the components described in Section 3.2 have been implemented, i. e. their functionality and communication between the components. All of those components could theoretically be exchanged, given they provide similar functionality, meaning that approach can be adapted to work with other constraint languages or runtime environments. But, the implementation is merely a prototype as of yet, not strictly following rules imposed by component based architectures. This makes it hard to actually adept it in practice and improvements should be made in the future (see Section 7.6).

4.1 OCL Constraint Checker

There are many implementations of OCL constraint checking / query engines out there, but as already mentioned the constraint checker initially developed for the ModelAnalyzer has been used. OCL aims at providing means to enrich MOF based models with additional constraints not expressible in the language itself. In this domain the assumption is that a predefined meta-model is present, used by the parser to perform type-checks. Certainly, a program written in Java is neither based on MOF nor does a fixed meta-model exist during runtime. As a matter of fact, it changes each time the runtime environment loads a new class. The currently available classes and their relations (including methods and fields) can be treated as the meta-model of the program being executed. Basically, there were two ways around the obstacle of static type checking, either ignoring them statically and performing them dynamically during runtime or adept an OCL checker to cope with the different environment. Since we prefer statically typed languages for the sake of avoiding errors during constraint definition, we wanted our constraint checker to still be able to perform those checks. Actually adapting an existing implementation

such that it suits our needs seemed infeasible so we decided to build our own. Fortunately, the concepts of OCL / MOF and Java are quite similar for the most part (hence the "Object"). An existing parser capable of parsing OCL constraints into the expressions used by the constraint checker has been adapted to being able to perform proper type checks for Java classes. In Sections2.1.1.2 we showed how standard OCL types are mapped to the Java ones. Some parts of the constraint checker itself also had to be adapted, i. e. how certain expression are validated. This was mostly the case for (but not limited to) expressions responsible for accessing values or invoking methods in a target application, as well as expression performing type checks.

One important thing to note is that we have to be able to cope with the lazy-loading most JVMs perform. Type checking during parsing would fail if the type of some field used in a constraint is not yet loaded by the runtime environment. In such a case we try forcing the type (i. e. the class) to be loaded. This may happen several times during parsing a constraint. Loading the class may also fail, causing the entire parsing to fail. Whenever that happens we delay the use of the constraint and retry it later, when the runtime environment has successfully loaded the type in question.

4.2 Observing Component

The observing and wrapping component is essentially JDI (described in Section 2.2) combined with some custom code. The notifications generated by JDI suit most of our needs, although some workarounds were required since JDI does not support notifications for some events needed. Those are mostly the creation of new objects, as well as the modification of collection contents. One benefit of using JVMTI is that once an application has been started with the proper launching options, a debugging application may connect to it at any time. Another possibility is to start the target application from debugger itself, causing it to become immediately attached to it. Although our tool is not a debugger in the strict sense, it still uses similar concepts. From conceptual point of view, our invariant checker can be treated as an extended debugger since we also provide options to suspend the programs executions if a violated invariant is detected.

4.2.1 Observing the Creation of new Objects

In Java, there are generally two ways how a new object may come into existence. Memory is allocated for it and the object is created by either invoking a constructor or by cloning an already existing one. The JDI does not support notifications for object creations natively. We can simply work around this issue by observing method exit events for constructor and clone calls. At first glance this sounds easier than it actually is, we only want to create object creation notifications when the object has been fully initialized. If we would perform invariant checks on an object that has only been partially initialized (or not all) they would most certainly fail. Most times there is not just one constructor involved during initializing an object, in fact

Algorithmus 3 Generating change notifications for collection changes

Input: a collection c and a set of objects R referencing collection c

```
for all object ∈ R do
  c ← object.class
  F ← c.fields
  for all f ∈ F do
    value ← f.getValue(object)
    if value = c then
      add change (MOD, object, f) to the list of recent changes
    end if
  end for
end for
```

every class in Java has at least one superclass whose constructor is also involved. We can be positive that the initialization is completed if the next method on the method call stack is not a constructor call on the object itself.

4.2.2 Observing Changes of Collection Contents

Collections in Java are considered as ordinary objects, not given any special treatment. Thus, JVMTI will not create specialized notifications concerning changes to its contents. Such notifications are still necessary if an operation iterating over the contents is used in a constraint. We explored two different ways tor generating such notifications. One way of doing so is memorizing all fields read during retrieving its contents. Whenever the value of one of those change, there is a high probability but not a necessity that the actual contents changed. This approach works with every imaginable implementation of a collection, but introduces the drawback of requiring to observe every single one of the fields mentioned, which may be a high performance deficit. Another possibility is to observe calls to methods such as add, remove, etc. This way observing the accessed fields become obsolete, although we possibly miss some changes to the contents. Consider a collection implementation which offers means to alter the contents without calling one the standard methods, rendering our notification mechanism incapable of detecting the change.

No matter which one we choose, there is still additional work to be done. As previously mentioned, our approach relies on the notion of scope elements, a collection can simply be considered being the value of a scope element (maybe more than just one). In order to generate proper change notifications as required by our incremental detection mechanism, we still need to find those scope elements. See Algorithm 3 for a more detailed description.

4.2.3 Accessing / Converting Values

As described in Section 2.2.3, JDI only provides Mirrors to the values in the target application, but to perform invariant checking the concrete ones are needed. Whenever a value of a field

is accessed during validation, it is not accessed directly via JDI. Instead it is retrieved via the scope element representing the field. If the scope elements cache is outdated (see below) the value is accessed via JDI and converted to a usable type.

- In case the value is of a primitive type, the conversion simply retrieves the actual primitive value and converts it into its object representation (`java.lang.Integer`, `java.lang.Boolean`, ...)
- If the value is a *StringReference*, the actual string can simply be queried from.
- If the value represents an array, the converter constructs a new instance of `java.util.ArrayList` and adds the arrays contents to it. Note that the actual values of the contents will also be converted accordingly.
- If the value represents a collection, it will be converted as described in 4.2.3.1.
- Otherwise, if the value is an ordinary *ObjectReference*, it is used as-is.

4.2.3.1 Querying Collection Contents

Additionally to observing content changes of collections, we also need a way to access its contents since there are expression that iterate over the contents of collections during the validation of invariant instances. Since collections are treated as ordinary objects and we do not know the concrete implementation (how the contents is stored), there needs to be a generic way to access the contents. Fortunately, every collection in Java implementing the `java.util.Collection` interface implements a `toArray()` method. This method can be used to retrieve the contents of a collection in the form of an array. Arrays are treated specially by JDI, a mirror to a array reference allows iterating over the arrays contents. To use the contents of the collection we perform the following steps:

1. Create a new collection C_N in the invariant checking component itself (the type of collection depends on the type used in the target application)
2. Call the `toArray()` on the collection C_T in the target application.
3. Iterate over the returned array
4. Convert each value as described in Section 4.2.3
5. Add converted value to C_N
6. Return C_n

4.2.3.2 Value Caching

As described in Section 2.2.3, accessing values over JDI always involves commutation overhead, which can drastically decrease the performance during invariant checking (see Section 5.2). Thus, we decided to let scope elements cache their current value. This can greatly increase the performance since the scope elements value can be accessed directly. Each scope element also contains a flag stating whether the cached value is up to date. Whenever a modification change is received for a particular scope element the central component will reset this flag. If later on the scope elements value is used during an invariant instances validation, it checks whether this flag is set. If not (cache is outdated) the current value will be retrieved via JDI and properly converted (as shown in Section 4.2.3), otherwise it may simply use the cached value.

This caching mechanism is especially crucial for scope elements representing collections, since it requires even more effort / time to retrieve its contents. Retrieving the contents involves invoking a method via JDI (which is even more costly than accessing a value) and converting the values recursively (see Section 4.2.3.1).

4.2.4 Connection options

Basically, we could support all connection options available when using JDI (see Section 2.2.1). But currently we only support two of them, the *LaunchingConnector* and the *SocketAttaching-Connector*. Other options are not available on all operating systems or are simply not suited for our needs, so we chose not to support them. Using these two kinds of connectors is sufficient to provide the two basic ways of connecting to the target application, by starting it from the invariant analyzer or by connecting to an already running one. When starting a new one, it basically acts like starting it from a command line, all relevant starting parameters must be supplied. This includes the class path, starting class and any other desired launching options. Starting a new application from scratch is especially useful for testing purposes, ensuring that no invariant violations is missed.

Connecting to an already running application mainly serves the purpose of being able to observe long running applications, such as server software or applications with a high amount of user interactions. Generally, such applications have higher requirements regarding response time and as already mentioned the additional overhead required for validating the invariants may have an high impact on it. However, whenever someone notices something going wrong or the application shows a weird behavior, there is the possibility of connecting to it and checking the desired invariants. One of the biggest advantages using this option is that the target application does not need run on the same machine as the Invariant Analyzer, which is especially useful when observing server software. Most server machines running such software do not even have an operating system with a graphical interface making it necessary to run a debugger on different machine. Obviously, this flexibility comes at the cost of an additional overhead due to relying on network communication. Inter-process communication using JVM TI is already

rather slow and exchanging data over a network makes things even worse. The connection settings necessary for using connection are the target hosts address and the port at which the application is listening for incoming debugging connections. There is also an additional timeout setting causing the connection attempt to be aborted if no answer is received within a given timeframe. If disabled (setting 0) the Invariant Analyzer waits forever for the connection to become established.

4.2.5 Synchronization with Target Application

Since the Invariant Analyzer is completely separated from the target application, i. e. running in different processes or even machines, it would potentially be possible to not influence the original execution time at all. While this is not entirely true since observing the necessary events and forwarding notifications still requires some time, the actual processing of the notification and invariant checking itself would not influence the original program. But, this comes also with risk of introducing race conditions and probably incorrect invariant validations. If during validating an invariant the target application modifies required values, it may lead to a different result. There are cases in which this behavior does not matter since using the current values may produce a more up-to-date result. The main problem is if the modification occurs when some parts of the invariant have already been validated, causing some parts to use the old value while others use the new one. Because of this problem the target application has to be suspended entirely before validating any invariant instances. This was implemented by instructing each method exit request and breakpoint request (the ones observing object creations) to suspend it when the event notification is forwarded. After processing the event and the required invariant instances are validated, it will immediately get resumed unless otherwise specified by the user.

4.3 Invariant Definition Storage

As of yet, our storage for invariant definitions is just a simple text file containing all relevant data in an XML format. It is not necessarily required to store all invariants in one file, splitting across multiple files is supported and strongly encouraged. The graphical user interface provides the ability to load each one separately, either replacing the already loaded definitions or simply adding them. If during adding it is recognized that a newly added invariant is already present (invariants are considered equivalent if their name and context are equivalent), it will be omitted. Table 4.1 shows what data is currently stored for each invariant.

4.4 Graphical User Interface

We provide a fully functional Graphical user interface developed as an Eclipse Rich Client Platform (RCP) application. Although it still lacks seamless integration into the existing platform for Java development the existing user interface offers all required functionality for using our

Field	Type	Usage
Name	String	a simple name used to identify instances of this invariant
Description	Text	free text, may be used to describe informally what the invariant does and its purpose
Definition	OCL Expression	the formal definition of the invariant. Must be a valid OCL expression evaluating to a boolean result
Enabled	Boolean	Indicates whether this invariant will be validated at all. Setting it to `false` will still keep it in the set of defined invariants although skipping during processing of changes
Keep Validation Tree	Boolean	Switches between default top-down and bottom-up validation, see Section 3.4.4
Generate Repair Tree	Boolean	Ignored as of yet. Will be used in the future indicating whether data structure repair actions shall be calculated during validation, see Section 7.2

Table 4.1: Stored invariant information

```
<invariants>
<invariantDefinition definition="self.next␣&lt;&gt;␣null␣implies␣
   self.next.prev␣=␣self" contextElement="list.Node" description=
   "" enabled="true" keepValidationTree="false" name="
   CorrectlyLinked" repairable="false"/>
<invariantDefinition definition="self.item␣&lt;&gt;␣null"
   contextElement="list.Node" description="" enabled="true"
   keepValidationTree="false" name="ElementReferenced" repairable
   ="false"/>
...
</constraints>
```

Listing 4.1: Example Invariant Definition File

approach in a tool supported manner. The main parts are an for editor for defining / modifying invariants, views reporting results and a way start a new application to be observed or connecting to an already existing one. Defining invariants shall be as comfortable as possible, especially since during writing them one may not be aware of the internal workings of the implementation. Thus, the editor component offers all the commonly features known from existing editors for programming languages, including syntax highlighting and code completion. Figure 4.1 shows a screenshot of the editor during definition of the *CorrectlyLinked* invariant and Figure 4.2 shows an example list for code completion, including all fields and methods of the *Node* class.

Apart from defining invariants, the most relevant information are their actual results after validation. As of yet we provide three different views for navigating through the results in a tree like structure. The different views provide starting points (top level elements) for navigating them, being scope elements, invariant instances, or invariants themselves. Figure 4.3 shows an

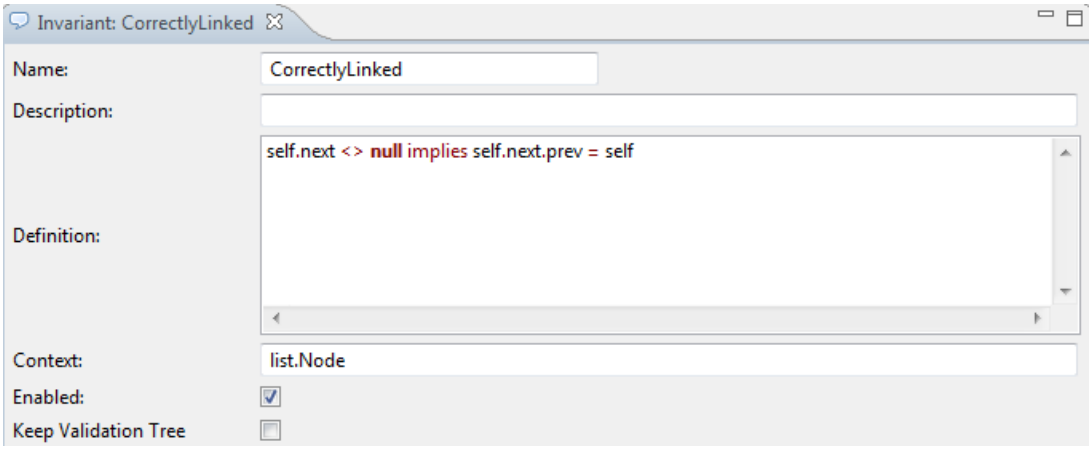


Figure 4.1: Invariant Editor

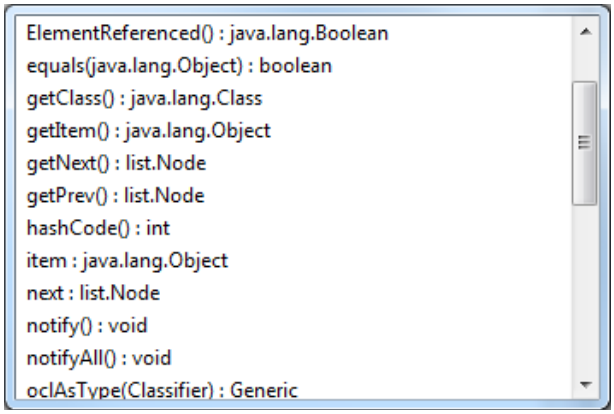


Figure 4.2: Code Completion

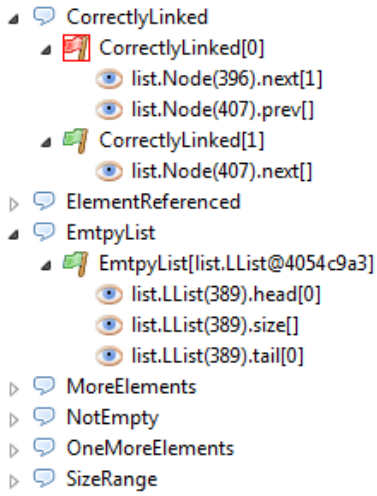


Figure 4.3: Invariant View

example the invariant view. It shows invariants as top level elements, which may be expanded to show their instances. The invariant instances again may be expanded to show the scope elements accessed by them. Validation results are shown as green (valid) or red flags (invalid). The rectangle around of the flags indicates that this invariant instance has been validated most recently, due to the last set of processed changes. The list of available invariants may become quite huge, thus the tool also offers the possibility to filter certain objects such as:

Selected invariants: only shows the selected invariants

Selected invariant instances: only shows the selected invariants instances

Selected scope elements: only shows the selected scope elements

Validated invariant instances: only shows the invariant instances affected by the last set of changes

Violated invariants: only shows the currently violated invariants (instances)

Although the OCL parser recognizes syntactically wrong invariants (in fact does not allow them), writing correct specifications is a rather hard task and one may get unexpected results due to incorrectly formulated invariants. Therefore, we offer some kind of debugging for invariants. If a user checks the *Keep Validation Tree* option of an invariant, there is an option to navigate through it, aiding in pinpointing what went wrong. The entire validation tree is presented as tree like structure, showing the results of all (sub)expressions (see Figure 4.4). Actually, the presented structure is truly a graph since it also shows the scope elements accessed during validation scope elements and variables may be accessed by multiple expressions. If someone is really new to the concept of how an OCL expression (or any other language) gets

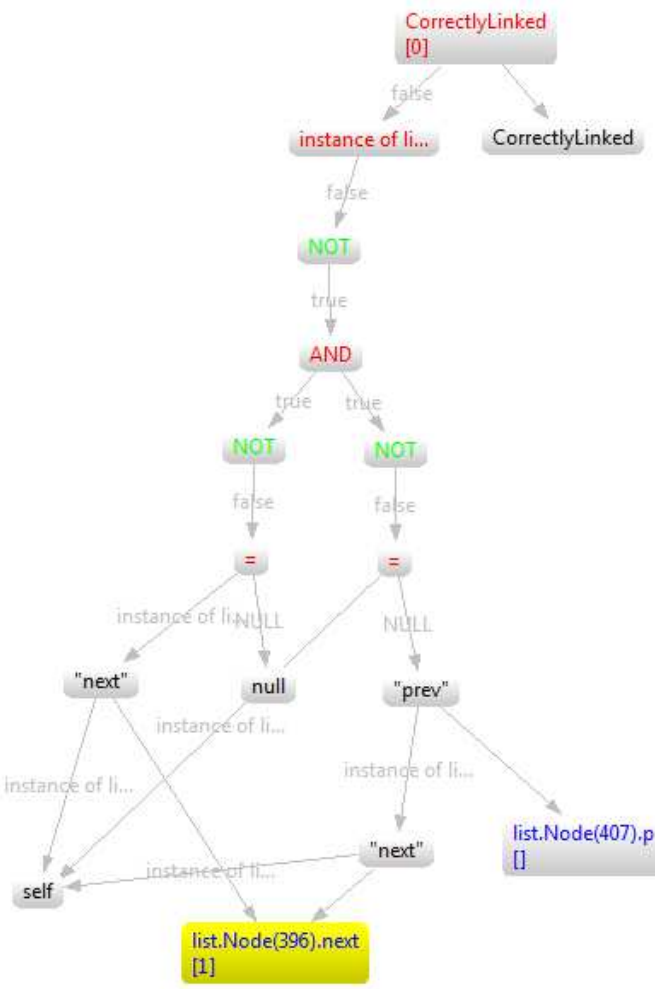


Figure 4.4: Validation Tree Navigation

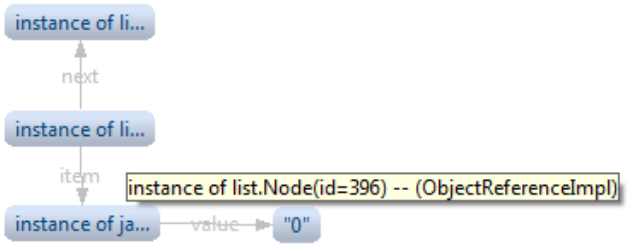


Figure 4.5: Structure Navigator

parsed into an abstract syntax tree, we also provide such a graphical view for visualizing it after parsing.

The lack of seamless integration with a debugger currently makes it hard to perform actual debugging using our tool. Nevertheless we offer an option to at least navigate through the data structures of the target application in a graphical representation (see Figure 4.5). Keep in mind that during normal execution of a target program the values may not stay the same very long, data structures in most applications change rapidly. One has to make sure to use the proper suspending options before visualizing and navigating through the data, i. e. make sure that the target application is suspended before navigating through values.

4.4.1 Setting Preferences

The InvariantAnalyzer also makes use of some global preferences that have to be properly set by the user as shown in Figure 4.6. The invariant file is the location of the invariant definition storage. It is used to load the invariant definitions contained in the given file during startup of the InvariantAnalyzer such that a user does not have to load the definitions manually. In fact it is merely a convenience option, which can be left empty. The user interface also allows set suspend options, instructing the InvariantAnalyzer the suspend the target application of certain criteria are met.

Additionally, dialogs are provided for setting the options necessary for using the provided connectors (see Figure 4.7).

4.5 Core Invariant Analyzer Component

The core centralized component takes care of the interoperability of the other components, i. e. telling them what to do. Each of the other components separately is pretty worthless without interconnections among them, e. g. the constraint checker has no use without knowing what to check and what to do with the results. We did not want connect the other components directly since each one could be used in other environments as well. The main tasks of the core component are as follows:

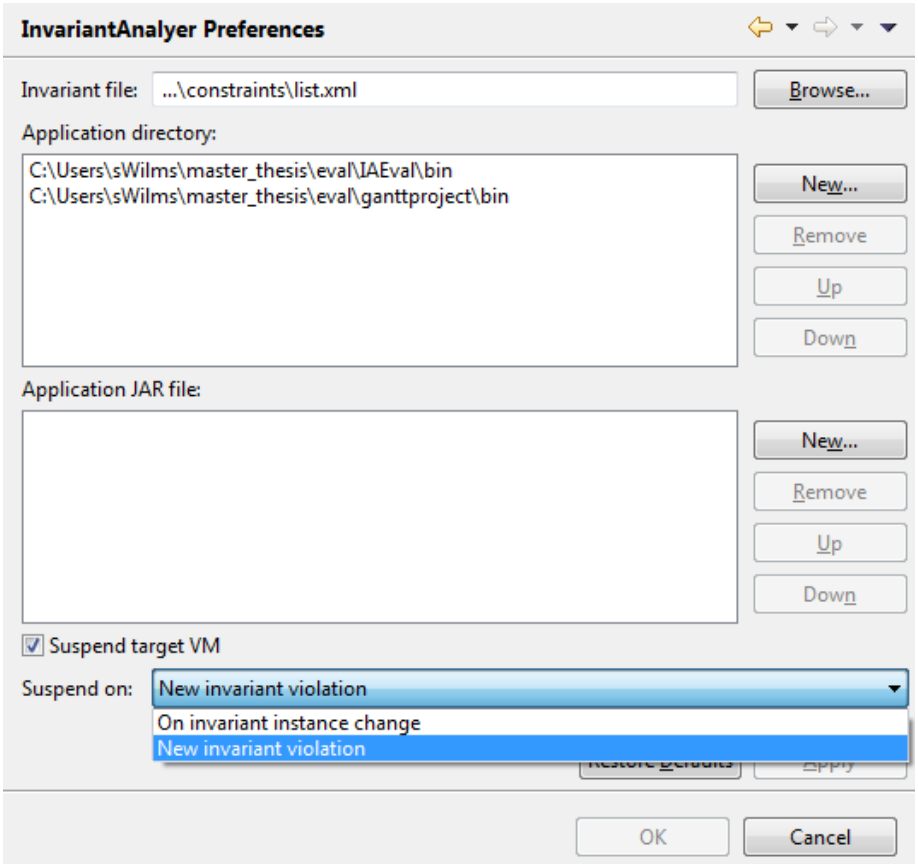
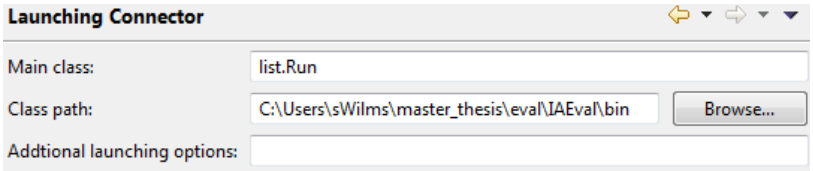
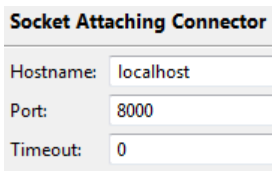


Figure 4.6: InvariantAnalyzer Preferences



(a) Launching Connector Preferences



(b) Socket Attaching Connector Preferences

Figure 4.7: Connector Preferences

- Instructing the JDI which notifications are relevant, i. e. filter the notifications forwarded by JDI, see Section 4.5.1. This is derived from the outcome of previous invariant (especially accessed scope elements). Initially, after connecting to a target application, only class load and object creation events for existing invariants are considered.

- Dispatching the required invariant checks to the constraint checker. The core component analyzes incoming change notifications, determining which invariant instances may be affected by them or if the creation of new ones is necessary. It will then instantiate invariants if necessary and order the constraint checker to validate those invariant instances. It may also delete certain invariant instances entirely if it comes to the conclusion that they are no longer needed, i. e. the object it relates to does no longer exist. This is essentially an implementation of the Algorithm 1, but with some optimizations.

- Updating the user interface. Strictly speaking, it will not actually update the user interface directly since we follow a strict model-view-controller (MVC) principle. But, somehow the user interface needs knowledge whether some data changed and update the views, which the core component takes care of.

- Receiving modifications form the user interface. The interaction between the user interface and the core component is not just in one direction. A user may of course modify specified invariants, add new ones etc. in which case appropriate processing of these events is required.

- Connecting to the target application. Whenever a user requests the Invariant Analyzer to start or connect to a target application, after setting the required connection options, the core component takes care of establishing the connection and creating initial requests for event notifications.

- Providing an invariant "queue". Due to the lazy loading of JVMs there may be invariants in the invariant definition storage for a context type that does not exist yet. Although we could force loading of the context type, it does make sense since the invariant is not yet needed anyhow. This is the reason why we also have to observe the loading / unloading of classes. Whenever a users adds an invariant manually or loads it from the invariant definition storage that has a context of a not yet loaded class, it is placed in a queue. Immediately afterward a class loading request is created for that type. When the target application loads the class all of its invariant are dequeued and properly inserted into the central data storage.

4.5.1 Notification Filtering

Mostly it is not necessary processing all event notifications generated during a programs execution. Usually, there will by a huge amount of classes for which there are not even invariants

defined (system classes, third-party libraries, etc.). For those classes we do not even need receive notifications when an object of it is created since there are no invariants to instantiate. This is also true for method exit events which we use for judging whether an object is in a publicly visible state. Furthermore, notifications informing about field modifications which are accessed by an invariant instance are irrelevant since they would change a result. Certainly, we could have chosen to not care about this and process the notifications anyhow, which would not change the results in any way but would have a great impact on overall performance. Basically, our approach would at some recognize that the notification is irrelevant and discard at anyhow. But the necessary computation time up to this can be spared by performing the filtering as soon as possible.

Fortunately, the JDI follows a request driven approach, i. e. it will forward notifications for events specifically requested beforehand. For example when creating a request for a field modification event, one has to explicitly state which field to observe. In case of a field modification it would then forward every notification for every object that modifies the given field. This can be restricted even more by adding additional filters, e. g. restricting the event to a specific object or restricting to a thread in which the change occurred. For method events (entry / exit) it provides way to restrict it to certain methods only (generally, non-public methods are irrelevant), meaning that the late filtering is necessary in this case anyhow.

Such requests may also be removed at any time, meaning that no more notifications for this kind of event will be forwarded.

The following list states when requests for certain events are created /deleted. Note that no request will be created if it already exists.

Class loading Created whenever an invariant with an unloaded context is defined or loaded. Removed immediately after the occurrence of the event since this is in fact a one time event.

Class unloading Created whenever an invariant is added to the central data model matching its context. Removed immediately after the occurrence of the event since this is in fact a one time event.

Field modification Created whenever a scope elements value is accessed during an invariant instances validation. Filtered such that only notification for field modifications of the scope elements object are received. Removed when recognized that the scope element is no longer referenced by any invariant instance, i. e. it does not exist any more.

Field access This kind of event is only observed during the execution of a method invoked in the target application by an invariant validation. The requests are created prior to invoking the method and removed immediately after it is finished.

Object creation Created whenever an invariant is added to the central data model. The is set up such that the type of the created object must be the context of the invariant or a

subclass thereof. It will be removed when no more invariants with the given context exist any more.

Object deletion Created when an invariant is instantiated. Filtered such that the notification only occurs for the object matching the invariant instances context element. Removed immediately after processing, an object can only be deleted once.

Method exit Created after receiving a notification for a field modification. A field modification indicates that some invariant instance might be afflicted. The central component will traverse the method stack trace up to the first public invoked on the object of the field modification (indicating transition into publicly visible state). Depending whether debugging information is available it will either create a breakpoint or method request at the end of the found method. Removed after processing the corresponding notification, or if the relevant invariant instances validation had been triggered by another event.

4.6 Central Data Storage

The data structures outlined in Chapter 3 do not directly correspond to the data structures used in the actual implementation. In fact the different entities have no direct connection whatsoever. All of the concepts (invariants, their instances and scope elements) are treated as first class citizens, rendering it unnecessary to follow many indirections to find the desired data. The central repository ensures that the relations among them at least exist virtually and offers operations for efficiently retrieving desired data. Currently, the central data storage is implemented as a simple that can only have a single instance, using the singleton design pattern [35]. All currently defined invariants, their instances, scope elements and their relations are simply stored in memory using multiple hash tables, providing both flexibility and rapid access. The data storage defines an interface for accessing and updating this data in a convenient way.

Chapter 5

Evaluation

This chapter evaluates whether we reached the goals explained in Section 1.4. Evaluating whether we achieved goals 1a and 2 comes down to a proof of correctness, which is shown in Section 5.1. The performance (goal 4) has been measured by providing concrete time measurements, shown in Section 5.2. An evaluation of the scalability requirement (goal 5) is discussed in Section 5.3. A brief explanation of the example applications used in the evaluation and the list of invariants defined for each of them can be found in Appendix A.

We evaluated goal number 1b by performing manual testing and it turned that invariants can be added / removed or even redefined without any indication of problems. Goal number 3 has only been achieved partially. We claim that the approach is generic by design due to its architecture. Exchanging the wrapping / observing component appropriately allows to the approach to be used for other runtime environments or programming languages. But, the current implementation is not generic enough as of yet, since the components are too strongly coupled. Improving the generic aspect of the implementation is one of the tasks for future work, see Section 7.6.

5.1 Correctness

We did not actually verify the correctness of the approach by formal proving. The correctness evaluation was made under the assumption the the constraint works correctly, i. e. that it yields correct results when instructed to validate an invariant. This means that we checked whether checking class invariants incrementally, which our approach does, works just as well as if it would not have this incremental behavior. Initially, we wanted to compare our approach to an existing one like the Java Modeling Language Run-time Assertion Compiler (JMLRAC, see Chapter 6). But since JMLRAC only introduces local checks and therefore lacks completeness, as we showed in Section 1.3, it was out of the question. Thus, we compared our approach with a “brute force” approach. The brute-force approach is both quite simple and able to detect all

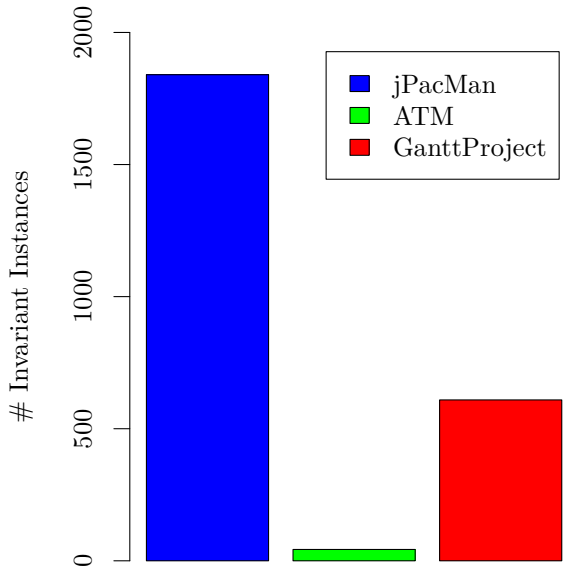


Figure 5.1: Amount of Invariant Instances during Evaluation

possible violations. At the end of each public method, i. e. object is in a publicly visible state, it will evaluate *all* defined invariants. Thus, it is guaranteed that it will detect all violations, but on the other hand performs a huge amount of unnecessary checks. We verify the correctness of our approach by comparison to the brute-force approach which is correct by definition. As already explained, the brute-force approach triggers the validation of all invariants. Although it is highly unlikely that our more sophisticated approach triggers the same evaluations, we can still compare the outcome of the validations at that point. Actual results are in fact not important, what matters is whether they changed. The brute-force approach may detect a violation, but as long as it existed already, it is mainly irrelevant. This means that whenever the result of an invariant validation changes in the brute-force approach, this change must also be detected by our incremental approach.

We empirically evaluated the correctness by running the test cases that are shipped with the source code of the ATM, jPacMan and GanttProject examples. Performing those tests we could not discover any indication that the incremental approach behaved incorrectly. Surely, this is not actual proof of correctness, but due to the number of tests and invariants involved, we are positive that the correctness property holds. We also performed manual, none automated tests, using other example applications, which also showed no sign of incorrect behavior. To give an idea about the size of the performed tests, Figure 5.1 shows the amount invariant instances that were created during execution of the tests for each application.

We conclude that we achieved goals 1a and 2. The user effort was minimized to defining the invariant, never mentioning when they have to be checked. Our approach is incremental by definition and still yields correct results compared to an exhaustive non-incremental approach.

CPU	Intel Q9550 @ 2.83Ghz
Main Memory	8 GB
OS	Windows 7 Service Pack 1 64-bit
JVM	Oracle HotSpot 64-bit Server VM (build 23.5-b02)
Java Version	1.7.0_09

Table 5.1: Evaluation Setup

Application	Time disabled [s]	Time enabled [s]	Slow-down
jPacMan	0.05	348.244	6964.88
ATM	0.019	23.76	1250.526
Gantt	0.604	1465.183	2425.8
Total	0.673	1837.187	2729.847

Table 5.2: Execution Times with invariant checking disabled and enabled

5.2 Performance

The general performance can be measured by comparing a system run when the invariant checking is disabled, to a system run that performs all required invariant checks. Obviously, in the second case the overall execution time is longer, but the question is to what extent. Thus, we measured execution times while both having invariant checking enabled and disabled. We used the same examples as in Section 5.1. All time measurements were recorded while executing the test cases on an ordinary PC. The exact configuration of this machine and the JVM used can be found in Table 5.1. By running the test cases we found out that the performance of our tool is in fact quite poor, i. e. running a test case took on average 2729.847 times longer when our tool performed the invariant checks compared to running the test cases without invariant checks. Table 5.2 shows the measured execution times of the tests, for both invariant checking enabled and disabled, as well the factor by which the invariant checking slowed down the execution.

By inserting the invariant checks in the source code of the application we saw that this lack of performance was not explainable by just the fact that invariant checks were performed, so we further investigated how this happened. Ultimately, it turned out that accessing a fields value or invoking a method in target the application during validating an invariant instance took up most of the time. Therefore, we performed some experiments to investigate to what degree the communication overhead introduced by JDI affected performance. Firstly, we took a look at how much slower it is to access a fields value via JDI compared to a native field access. Table 5.3 shows the time it took to perform various consecutive field accesses (n) in milliseconds and the calculated factor to which accessing the fields via JDI was slower. The fields value was a simple string in all cases. Furthermore we were interested in the overhead when invoking methods via JDI. Table 5.4 shows the results when invoking a simple “getter” method for the field from the previous experiment. It turns out that it does not make much of a difference whether the fields value is accessed directly or via its getter method. What actually does make

n	total native	total JDI	mean native	mean JDI	slow down
1	0.00109	1.944	0.00109	1.944	1789.890
10	0.00145	2.691	0.00014	0.269	1857.083
100	0.00290	12.885	0.00003	0.129	4444.514
1000	0.01558	107.085	0.00002	0.107	6873.683
10000	0.14275	1108.898	0.00001	0.111	7768.007
100000	1.47861	8167.816	0.00001	0.082	5523.990

Table 5.3: JDI field access overhead

n	total native	total JDI	mean native	mean JDI	slow down
1	0.00181	2.047	0.00181	2.047	1129.935
10	0.00217	5.108	0.00022	0.511	2349.714
100	0.00580	31.055	0.00006	0.311	5357.037
1000	0.04674	239.241	0.00005	0.239	5118.760
10000	0.36195	1690.134	0.00004	0.169	4669.485
100000	1.37317	12335.310	0.00001	0.123	8983.064

Table 5.4: JDI method call overhead

a difference is when arguments have to be passed to the method. Calling the "getter" method of the field showed that JDI performed even slower than when the fields value was accessed or calling the getter method, as shown in Table 5.5.

This lead us to implement the caching of scope elements values as described in Section 4.2.3.2. Unfortunately, this did not result in the desired performance increase. Since an invariant instances will only be revalidated if the value of one of the accessed fields changes, the new value has to be queried anyhow during validation. Secondly, the caching does not affect method calls. Method calls have to be revalidated every single time, since the result may not only depend on the current state of an object but also on local variables and arguments passed when invoking the method. Furthermore, we do not cache queried reflective type information, which is involved when collecting the invariant instances for revalidation or checking whether new invariant instances have to be instantiated. As discussed in Section 2.2.3, accessing this reflective type information also involves communication overhead via JDI.

Ultimately, we conclude that the performance of our approach when using JDI may be good enough for testing purposes. The longer execution time may be acceptable for the benefit of

n	total native	total JDI	mean native	mean JDI	slow down
1	0.00181	9.684	0.00181	9.684	5344.147
10	0.00217	14.911	0.00022	1.491	6858.987
100	0.00543	67.377	0.00005	0.674	12396.840
1000	0.03841	520.633	0.00004	0.521	13556.375
10000	0.40905	3943.608	0.00004	0.394	9640.801
100000	1.57752	33361.399	0.00002	0.334	21148.004

Table 5.5: JDI method call with argument overhead

ensuring that all invariant violations are detected. On the other hand it is not acceptable for monitoring productive environments. The performance may decrease to a degree that the system becomes virtually unusable or violates performance requirements. This means that we failed to achieve goal number 3. It has been decided that solving the performance issue requires too much effort to get solved as part of this thesis and should be worked on in the future as discussed in Section 7.1.

5.3 Scalability

The scalability of our approach is independent of the size or complexity of the target system. What is really important regarding execution time is, how many change events occur during a certain time frame and how many invariants are affected by those changes. Traditional approaches tend to check the invariants for a certain object after exiting a public method. We show that compared to such an approach our incremental one scales better. Because of the poor performance of JDI, we do not compare execution times directly, using for example the JMLRAC would always be faster since it does not rely on JDI and performs native field accesses and method calls. Instead we just count the number of invariant instance validations that would be triggered using the traditional approach as well as our incremental one.

In this case we do not use the same applications and their test cases as in the previous section. The provided test cases are deterministic, meaning that there would no way to influence the problem size when using those. Instead we used our linked list example and its invariants. The test constructs a list containing n elements or nodes. After the list is initialized it randomly calls m methods on each of the n nodes. Ultimately, after initialization of the list structure $n \cdot m$ methods are called. Those methods may or may not impose side effects, i. e. change the state of a node (on average one third of the method calls caused side effects). Since the traditional approach always triggers invariant validation after exiting a public method, the amount of validation remains constant for each combination of n and m in this case. In the incremental case, the amount of validations per invariant instances varies, depending on how many method imposed side effects. Since methods are picked randomly, we ran the experiment five times for each combination of n and m. The results are shown in Table 5.6, depicting the number of invariant instance validations for both the incremental and traditional approach. In case of the incremental one, we picked the median value. We can see that in the incremental case the number of validation does not grow nearly as fast with increasing values of n and m as in the traditional one.

Using these values we can calculate the speedUp the incremental approach achieved compared to the traditional one. Figure 5.2 shows a boxplot of the speedUp for every run. The interesting result is that for low m the speedUp is almost negligible, since most of the validations are performed during initializing the list. The speedUp increases with higher m, while it remains almost constant for constant m, and higher n. We conclude that goal number 5

n	m	traditional	incremental
1	1	33	17
1	10	51	20
1	100	231	59
10	1	258	100
10	10	438	130
10	100	2238	400
100	1	2508	943
100	10	4308	1225
100	100	22308	3895

Table 5.6: Number of invariant validations

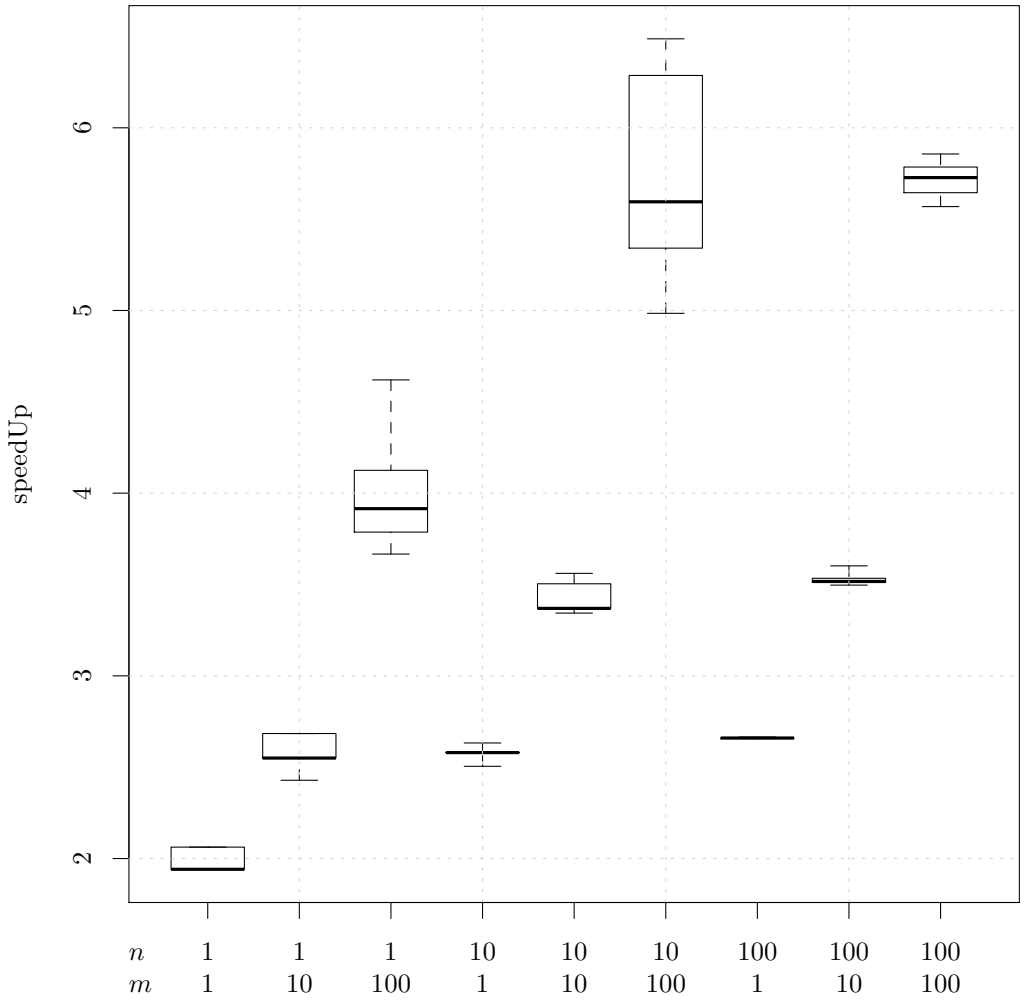


Figure 5.2: SpeedUps of incremental approach

has been achieved by showing that the amount of performed invariant checks has been reduced drastically compared to a traditional approach. Furthermore our approach is mostly independent regarding the amount of existing objects (amount of performed invariant checks grows less than linear with the amount of objects), meaning that it also scales up to bigger runtime structures.

5.4 Complexity

We can infer the complexity of our approach mainly by analyzing Algorithm 1. The algorithm consists of two major parts, finding out which invariant instances must be revalidated and afterward actually validating them. The time required to lookup the required invariant instances depends on two factors, the incoming change set as well as the time necessary to perform a single lookup.

$$CS \ldots \text{set of changes}$$
$$T_{lookup}(CS) = O(|CS| \cdot t_{lookup})$$

Algorithm 1 suggests that t_{lookup} also depends on the set of currently existing invariant instances (for modification and deletion changes) or the set of defined invariants (for creation changes). But, the actual lookup is implemented by using hashtable lookups, which can be performed in constant time ($O(1)$). Thus, T_{lookup} only depends on the set of changes, or more specifically the number of elements inside the change set.

$$T_{lookup}(CS) = O(|CS|)$$

The time required to validate the invariant instances depends on the size of the set of invariant instances to validate as well as the time necessary to perform the actual validation of an invariant instance.

$$II_e \ldots \text{set of invariant instances to validate}$$
$$t_{eval} \ldots \text{time validate an invariant instance}$$
$$T_{eval}(II_e) = O(|II_e| \cdot t_{eval})$$

How long a single validation takes depends on the definition of each respective invariant and generally grows with the number of expressions it contains. But, the number This means that t_{eval} is not constant, but actually a function depending on the number of expressions. Validating an expression requires a constant amount of time (t_{exp}), thus t_{eval} can be defined as:

$$t_{exp} \ldots \text{time to validate an expression}$$
$$e \ldots \text{number of validated expressions}$$
$$t_{eval}(e) = t_{exp} \cdot e$$

If an invariant does not iterate over a collection, e is constant and therefore t_{eval} can be neglected regarding asymptotic complexity. Otherwise, it depends on the number of elements iterated over, i. e. the e grows linear with the number of elements iterated over. In the worst case in invariant validation may iterate over all objects existing in memory, although it is highly unlikely. Empirical evaluation showed that the validation time of most invariants is mostly independent of the number of existing objects. For invariants not showing this behavior, i. e. iterating over a steadily increasing number of elemens, using the bottom-up validation mode described in Section 3.4.4.2 it becomes independent of the number of elements. For those reasons we treat t_{eval} as being constant for a specific invariant. Thus:

$$T_{eval}(II_e) = O(|II_e|)$$

The set II_e depends on the set of changes, or more specifically how many invariant instances are affected by a given change. It will increase with the number of invariants, the more invariants exist, the more likely it is that an invariant instance is affected by a change since more fields will be accessed if more invariants are defined. Additionally, the factor p describes how likely it is that an invariant is affected by the change of a fields value. It depends on the execution flow of the system and can be determined empirically for a specific system run. Usually p is low since a lot of changes will not affect any invariant since will access the changed field, especially if the value belongs to a system library or other reliable third-party libraries ($0 < p \ll 1$) Thus:

$$I \ldots \text{set of invariants}$$
$$|II_e| = |I| \cdot p \cdot |CS|$$
$$T_{eval}(CS, I) = O(|I| \cdot p \cdot |CS|)$$

The total overall complexity of the algorithm is thus:

$$T_{total}(CS, I) = T_{lookup}(CS) + T_{eval}(CS, I) = O(|CS|) + O(|I| \cdot p \cdot |CS|)$$

But, this is only the complexity for performing the algorithm once. During an entire system run, it will be triggered m times depending on how often the context element of an invariant instance transitions into a publicly visible state. We call the time between triggering validation an execution increment. Since the number of changes during each execution increment is not

constant and depends on the execution flow of the system, we introduce a factor η, representig the average amount of changes per execution increment. The overall complexity of the algorithm can be given as:

$$T_{overall}\left(m, \eta, I\right) = m \cdot \left(T_{lookup}(\eta) + T_{eval}\left(\eta, I\right)\right) = O\left(m \cdot \left(\eta + |I| \cdot p \cdot \eta\right)\right)$$

We can conclude that the main driving factor in terms of computational complexity is the overall number of changes during a system run, as well as execution time of the system. The longer the system is executed the higher the number of execution increments m will get. In most cases the set of defined invariants will not change during the execution and could also be treated as constant.

5.5 Threats to Validity

We see the biggest threat to the validity of the evaluation in the chosen example applications. One might argue that the applications used are too small and not representative enough since they are not "real" systems. We used the applications because it is hard to find bigger open source applications including predefined class invariants and we had no access to formally defined invariants of proprietary systems. On the other hand the used examples from different domains and we find that they are big enough. Furthermore the invariants could be more diverse / complex. This is why in case of the GanttProject application we introduced some invariants that do not exactly check meaningful properties, but instead exist for the sole purpose of having some more complex invariants. Regarding the scalability evaluation, one might argue that we purposely implemented the example and the performed test such that it clearly favors our approach. As a matter of fact, we could have constructed the test such that the achieved speedUp would be negligible by only invoking methods that cause side effects. But, this is not the case in real systems either. Usually, a fair amount of methods of a class do not modify the state in any way. There are even systems that only use immutable objects, i. e. objects whose state can never change. Our approach would only check the invariants for such an object exactly once, after initialization. All further method calls on the object can solely query its state and do not modify it.

Chapter 6

Related Work

There are many approaches dealing with verifying correct system behavior. In general a piece of software can either be verified statically by means of formal proving or dynamically during runtime. Over the years several specification languages have been developed. Although the programming language Eiffel had several flaws in its design [6], it was the first major language which explicitly contained the notion of preconditions, postconditions and invariants in its specification [23]. Later languages do not support those concepts directly, but efforts are being made to enrich existing languages by creating new specification languages and embed them into the original one, e. g. the Java Modeling Language (JML) [20]. Unfortunately, those languages tend to be incomplete, meaning that programming language develops faster than the specification. At least in case of the JML the tool support is rather poor, possibly caused by the fact that the group(s) developing JML and its tools is / are different from the one developing Java. Stable versions of the tool-chain do not support newer features of Java and tools that do support those are rather unstable and sometimes lack further development. The .NET programming language family does not support design by contract natively either, but efforts are being made to integrate it by providing a so called code contracts library [26, 12]. This project is possibly more promising since it is developed and supported by Microsoft themselves. Languages like the Object Constraint Language (OCL [28]) aim at providing functionality to specify behavioral or structural aspects during design time. Usually all those languages are as powerful as first order logic.

6.1 Static checking

The concept of statically checking or proving the correctness of a program was introduced by Hoare in the late 1960s [16] and refined by Dijkstra [7, 8]. Using these techniques one can actually mathematically proof that an algorithm works correctly. Simply put, it has to be proven that taking the precondition of a function, after the execution of the function its

postcondition holds. In this manner we could show that the `setPrev` method in our illustration is incorrect, since the proof fails. The proof for the method `setNext` indeed succeeds, although we can construct an invalid linked list using this method as well. But as mentioned before, this is caused by the fact that the invariant is incomplete. If we would use the complete definition, i. e. checking both directions, it would in fact turn out that both methods are incorrect. However, as the name suggests static verification is only useful if the entire system (especially function calling relationships) are known prior to execution. As systems grow in size and complexity or even exchange components during runtime and change their behavior, formal proving is rendered impossible.

Another problem that remains is that formally proving a program requires manual effort. Theoretically, not even any tool is required, just pen and paper would suffice. But, since proving even a slightly bigger program increases the needed effort drastically, tool support is of the essence. Tools like the KeY environment [1] drastically reduce the required manual effort, but can never eliminate it entirely. This is caused by the fact that the satisfiability problem for certain groups of first order language is generally undecidable [15]. Although the most common groups used for defining class invariants are actually decidable, proving them is still an NP-complete problem [5]. Thus, proving the correctness of some constraints may either run indefinitely or would require way too much time without human guidance.

There are also tools available that do not require manual effort, but do not actually verify the correctness of a program. One example would be the Extended Static Checker for Java (ESC/Java [13]) which can be used to find errors in Java programs annotated with constraints written in the Java Modeling Language (JML). Unfortunately, it is neither sound nor complete [19], although efforts are made to complete its completeness [17]. It is not sound in a way that is produces false positives, i. e. when it states that a function is correct it actually means that it could not find an error, not that there is none. This behavior is caused by the presence of loops. To formally prove loops, loop invariants are required, which can be defined using JML but are ignored as far as ESC is concerned. What ESC actually does, is that it performs loop unrolling by performing the loop n number of times (customizable). Obviously, the higher n is, the higher the chance that it will find an incorrectness but does not guarantee that there is no error if none is found. Incompleteness relates to the fact that it does not support all language features of Java and JML. The code contracts project for the .NET family also provides a tool for static checking called `cccheck` [11]. This tool is more sophisticated in that it takes loop invariants into account and furthermore, tries to infer a minimal one from the postcondition if none is present.

Yet another means to verify correct system behavior statically are model checkers [4], mostly useful to verify the correct interactions of a multi-threaded system. Simply put, one has to come up with a formal model of the system and properties it has to fulfill, written in some sort of temporal logic. A model checker can then more or less simulate the systems behavior and verify whether the properties hold or not. One of the major challenges using a model checker

is ensuring that the formal model actually models the "real" system. Again, a software system evolves over time and the model may become inconsistent regarding the system, if not properly maintained. Fortunately, a lot of research effort is into coming up with transformers capable of inferring a formal model from a programs source code.

6.2 Dynamic checking

The drawbacks of statically checking the correctness of a program can be avoided by doing it dynamically at runtime. Although it eliminates those drawbacks, it has others. Most notably the common known fact that dynamic checking can never verify the correctness of a program. It is thus a simple means of testing, i. e. finding errors during the execution of the program. Furthermore, executing the checks at runtime takes up computation time and thus may be unsuited in a productive environment. Although our own approach has the same problems as well, it aims at drastically reducing the necessary execution time.

The goal of most existing approaches is to generate new code for existing classes, which then contains the required checks (either directly in the compiled code or by generating new source code). This is exactly the behavior introduced in Eiffel. While this works perfectly fine for method pre- and postconditions, it only approached invariant checking by introducing local checks, i. e. adding the invariant to the precondition and postcondition of every (public) method defined in the respective class. We already showed in Section 1.3 that by doing so too many invariant checks may be performed as well as missing some invariant violations. The list of approaches with such a behavior would be quite huge. Usually, they either weave the required checks into the existing classes, e. g. the runtime assertion compiler for the JML [3]. Other approaches aim at not modifying the original code and instead generating new classes that perform the invariant checks by making use of common design patterns for object oriented software [22]. Leaving the original code untouched makes it easier to enable and disable the checks during runtime in a productive environment. The approach which comes close to our own from an architectural point of view [27], also entirely uncouples the component performing the checks from the target application, making it even easier to run the original application without checks being performed.

All related approaches we came across still have the issue of being to exhaustive, i. e. performing too many checks when it comes to invariants, whereas our approach is fully incremental and avoids most unnecessary checks. At least the JML has a mechanism to inform the runtime assertion compiler that a method is a side-effect, and thus no invariant checks need to be performed as part of their postconditions. The behavior that they will not recognize certain invariant violations can be partially ignored, provided they will detect a violation if the invariants are complete. Our approach on the other hand is in some cases still able to detect violations even though the invariant definitions are incomplete. Furthermore, it detects violations caused by objects others would not. It comes down to whether one just wants to only detect errors

during program execution or be provided with more complete information, which may be useful during debugging.

Chapter 7

Conclusion and Future Work

This master thesis introduced a novel approach for incremental class invariant checking. We evaluated our implementation showing that it does in fact work correctly and theoretically scales up to bigger systems. By using OCL as the constraint language for defining invariants, we bridge the gap between design and implementation, as well as provide an implementation language independent way of formulating them. The invariant definitions are entirely decoupled from the implementations source, providing superior flexibility for users. The ability to add and remove invariants during runtime, or even modify, allows to only check currently relevant ones or adept them as needed. But, there are still things that need to be addressed in order to be useful in a productive software development environment. Most of all, all the identified performance issues have to be addressed as well as properly integrating the implementation in an integrated development environment. Additionally, from a conceptual point of view our overall vision has not been achieved yet. This work is simply the first step in this direction by providing a basis future features can benefit from. In particular we want our approach to be able to aid user in fixing errors in software systems as well as using all concepts of design by contract. In the following sections we discuss the further steps that need to be addressed to reach the final goal.

7.1 Increasing Performance

We showed that as of yet the performance of our tool is rather poor. But, it has be noted that this is not caused by our implementation itself, rather than by the use of JDI and its inter-process communication. While on one hand performing the invariant checks in a separate process provides more flexibility, it also causes additional overhead. Reasons are mainly the need for socket communication between the processes and necessary synchronization. Every single time some data is queried from the virtual machine the target application is running in, this overhead becomes relevant. This does not apply for accessing field values or invoking methods

during the invariant checks, but also whenever accessing reflective type or similar information which happens quite frequently. One way to avoid those problems would be to let both tasks run in a single process, which raises the question how the necessary event notifications are gathered in this scenario. We are currently working on a way to weave additional code into existing classes notifying our invariant analyzer whenever events such as a field modification, method exit, etc. occurs. Generally, this could theoretically be done by modifying either the source code or the compiled byte code. But, since the source code may not be available (imagine an application using third-party components / libraries), only modifying the resulting byte code directly is left as a solution. Although running the invariant checks in the same process will definitely slow down the execution speed of the target application, we are positive that it will increase the overall performance. We still need to gather evidence to prove this claim, but the fact that we already freeze the target application during invariant checks to ensure correct results makes it pretty much obvious.

7.2 Fixing Errors

Although our proposed approach works quite well regarding the detection of invariant violations, the goal is not just detecting the existence of errors but, ultimately correcting them. We want to provide additional support helping a developer to fix the errors existing in the system. Fixing the errors in the source code immediately is not always possible, especially if these errors are found during the execution of a productive system. Due to economic or organizational reasons it may not be desired to shut down an entire system, fix the error, redeploy it, and restart the whole thing. But, a system that is in an invalid state may cause huge financial loss, or even human casualties. Therefore, as a first step, we want to provide means to "hotfix" the data structures used by the system and thus transition it back into a safe state. The proposed fixes will be quite similar to the ones shown in [37], although in case of a running system, user interaction will mostly be neither desirable nor feasible. Some sort of automated fixing is required in this case. If there is only one possible fixing action, then there is no arguing about which one to apply and fixing the error automatically is easy. Unfortunately, this case occurs quite seldom. Optimally, we will provide the opportunity to not only define the invariants themselves, but also how a possible violation should be resolved. Another option would be to use some kind of a supervised machine learning approach, i. e. let a user choose which action to take a finite number of times and use this information to automatically apply those actions for similar violations in the future.

Fixing the data structures in the running system is not even half of the story. While it is fine as an intermediate solution, it will not prevent the error from occurring over and over again. Eventually, the error needs to be fixed in the application's source code. We find the error reports generated by most existing approaches insufficient to actually aid a developer in fixing the flawed code. Mostly, they merely provide a method call stack that leads to the erroneous

method causing an invariant's violation. We are confident that the additional data we collect during an invariant's validation is useful for debugging purposes. Our approach does not only collect the method calls involved in the violation, but also all the scope elements (including their values), as well as the changes causing the defect. Furthermore, the proposed actions for fixing the defected data structures, can also be used for debugging purposes, as shown in [21].

The ultimate vision, which may be utopia, is that we end up with an approach capable of automatically fixing a target application to a certain extent, stopping the invariant violations from occurring. State-of-the-art runtime environments are capable of "hot-swapping" the code of methods even during runtime. Using this possibility it is even feasible to fix the errors in a productive system without the need shut it down ever. Obviously the changes performed on the code during runtime need to be written back to the source code eventually.

7.3 Supporting Additional Design by Contract Paradigms

Previously, we stated that checking paradigms such as pre- and postconditions (method contracts) does not require an approach such as ours that performs monitoring a global scale since it is safe to check them locally at the beginning and end of a method respectively. But, form a development / testing process point of view it not desirable to check those using different approaches / tools. Eventually our tool should include capabilities for checking all kinds of constraints, not only class invariants, including loop invariants. While it indeed suffices to check method contracts locally, they still share some of the issues we identified regarding class invariants, especially the room for human errors. An invariant definition may become invalid or irrelevant due to software evolution, which is also true for other types of constraints. Furthermore a class invariant defined in a superclass must also be maintained by its subclasses, which also applies to postconditions of overridden methods. For preconditions this is the other way around to some degree. One may choose that a method shall maintain the precondition of the overridden one, but need not necessarily be the case. Confusing this behavior is another error source avoidable by using an automated approach.

Loop invariants are quite closely to class invariants in that they need to be maintained over a longer time period. Basically, they have to hold when entering a loop and after each loop iteration. Although we could use our existing functionality to support loop invariants, i. e. by only enabling during the execution of a loop, we are not quite sure how to implement it properly. Our approach makes it unnecessary to know any implementation details and there is simply no way to pinpoint the exact location of the beginning / end of loop without knowing those details. For starters, additionally to the events we already gather, we would need events notifying us about entering a loop and end of an iteration. This is hard to achieve because there are no special statements in Java byte code making these kind of events explicit. Another issue would be how define the context of a loop invariant. One would not be able to state that an invariant must hold during the execution of a specific loop inside a method without doing

it in the actual source code, simply because loops are not named constructs in Java.

7.4 Invariant Checking Triggering Options

Traditionally class invariants have to hold whenever an object is in publicly visible state. But, for some reason, one may want to make this condition more strict, or relax a bit. For example, consider an invariant that should prevent a field from overflows. For such a constraint it makes sense to ensure that it holds at any time during execution, since an algorithm using its value will most certainly fail. This does not mean that the invariant becomes part of the methods precondition, which has to be the case anyways, but that it also has to hold during the execution of the method itself. It may also be the other way around, i. e. deferring the invariant check and making its validation depending on some other condition. In our obviously flawed linked list implementation the *CorrectlyLinked* invariant will be checked after each call of the `setNext(...)` or `setPrev(...)` method. Since the fields are not accessible directly form within the List class, there is no possible way that allows for intermediate incorrect states. Usually, the List class would provide methods like `add(...)` or `remove(...)` to modify its contents. If it has to only rely on the `setNext(...)` and `setPrev(...)` for doing so, there is no way to insert or remove an element from the list without violating the *CorrectlyLinked* invariant at some point. This problem can be avoided by allowing to specify an alternative condition for triggering the invariant check. In this case it would suffice to state it should be triggered whenever the list object containing the node transitions into a publicly visible state, or one could be more specific define a set of methods that trigger the validation.

Essentially, we want to provide as much freedom as possible regarding triggering conditions, without making it overly complex. While on one hand this option provides more freedom it necessarily also increases the manual effort. We still have to figure out what sort conditions actually make sense, there may be certain conditions that in combination contradict each other. Our approach needs at least a way to detect invalid conditions and desirably provide some guidance for the user.

7.5 Seamless Integration with IDE

Although currently our tool works good enough for a prototypical implementation, it is remotely from being fit for usage in active development. For our tool to actually be useful during development / testing it needs to be seamlessly integrated within an Integrated Development Environment. It is already built upon the Eclipse Platform, so to fully integrate it with its Java Development Tools, especially the debugger, is a logical step. The Eclipse Platform supports the notion of so called Run Configuration. We should make use of this and provide our type of configuration by probably extending the default debugging configuration and the built-in debugger. Doing so would make it much easier to launch a program for testing purposes using

our InvariantAnalyzer and being able to pinpoint to locations in the source code. Extending the debugger would also provide easy access to the functionality of stepping in the programs executions once a violations is detected, enhancing the chance of finding the error that caused the violation in the source code.

Furthermore, currently it is necessary to provide specific class path information to define invariants prior to runtime. Doing this during development is redundant, since it is already configured in the launched Eclipse project(s).

7.6 Refactoring Implementation

Although the approach is generic in a sense that it can be applied to arbitrary object oriented languages, the implementation does not. Currently, it is merely a prototype and its components are too strongly coupled to allow easily exchanging them. Quite some refactoring is required in order to come up with an implementation that allows easy adaptation. Therefore it is necessary to introduce structures representing concepts shared by most object oriented languages. Invariant checking should then solely work on those structures. An additional layer on top of the wrapping / observing component will then be used to map the language specific concepts to generic ones and vice versa.

Bibliography

[1] B. Beckert, R. Hähnle, and P. H. Schmitt, Eds., *Verification of Object-Oriented Software: The KeY Approach*, ser. LNCS 4334. Springer-Verlag, 2007. 78, 91

[2] R. Calinescu, C. Ghezzi, M. Z. Kwiatkowska, and R. Mirandola, "Self-adaptive software needs quantitative verification at runtime," *Commun. ACM*, vol. 55, no. 9, pp. 69–77, 2012. 1

[3] Y. Cheon, "A runtime assertion checker for the java modeling language," Department of Computer Science, Iowa State University, Tech. Rep., 2003. 79

[4] E. M. Clarke, O. Grumberg, and D. Peled, *Model checking*. MIT Press, 2001. 78

[5] S. A. Cook, "The complexity of theorem-proving procedures," in *STOC*, M. A. Harrison, R. B. Banerji, and J. D. Ullman, Eds. ACM, 1971, pp. 151–158. 78

[6] W. R. Cook, "A proposal for making eiffel type-safe," *Comput. J.*, vol. 32, no. 4, pp. 305–311, 1989. 77

[7] E. W. Dijkstra, "Guarded commands, nondeterminacy and formal derivation of programs," *Commun. ACM*, vol. 18, no. 8, pp. 453–457, 1975. 77

[8] ——, *A Discipline of Programming*. Prentice-Hall, 1976. 77

[9] A. Egyed, "Instant consistency checking for the UML," in *ICSE*, 2006, pp. 381–390. 33

[10] ——, "UML/Analyzer: A Tool for the Instant Consistency Checking of UML Models," in *ICSE*. IEEE Computer Society, 2007, pp. 793–796. 33

[11] M. Fähndrich, "Static verification for code contracts," in *SAS*, ser. Lecture Notes in Computer Science, R. Cousot and M. Martel, Eds., vol. 6337. Springer, 2010, pp. 2–5. 78

[12] M. Fähndrich, M. Barnett, and F. Logozzo, "Embedded contract languages," in *SAC*, S. Y. Shin, S. Ossowski, M. Schumacher, M. J. Palakal, and C.-C. Hung, Eds. ACM, 2010, pp. 2103–2110. 77

[13] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata, "Extended static checking for java," in *PLDI*, J. Knoop and L. J. Hendren, Eds. ACM, 2002, pp. 234–245. 78

[14] H. L. Gantt, *Organizing for work*. Harcourt, Brace and Howe, 1919. 93

[15] K. Gödel, "Über formal unentscheidbare sätze der principia mathematica und verwandter systeme i," *Monatshefte für Mathematik*, vol. 38, no. 1, pp. 173–198, 1931. 78

[16] C. A. R. Hoare, "An axiomatic basis for computer programming," *Commun. ACM*, vol. 12, no. 10, pp. 576–580, 1969. 77

[17] P. R. James and P. Chalin, "Faster and more complete extended static checking for the java modeling language," *J. Autom. Reasoning*, vol. 44, no. 1-2, pp. 145–174, 2010. 78

[18] J.-M. Jézéquel and B. Meyer, "Design by contract: The lessons of ariane," *IEEE Computer*, vol. 30, no. 1, pp. 129–130, 1997. 1

[19] J. R. Kiniry, A. E. Morkan, and B. Denby, "Soundness and completeness warnings in esc/java2," in *Proceedings of the 2006 conference on Specification and verification of component-based systems*, ser. SAVCBS '06. New York, NY, USA: ACM, 2006, pp. 19–24. [Online]. Available: http://doi.acm.org/10.1145/1181195.1181200 78

[20] G. Leavens and Y. Cheon, "Design by contract with jml," *Draft, available from jmlspecs.org*, 2006. 77

[21] M. Z. Malik, J. H. Siddiqui, and S. Khurshid, "Constraint-based program debugging using data structure repair," in *ICST*. IEEE Computer Society, 2011, pp. 190–199. 83

[22] B. A. Malloy and J. F. Power, "Exploiting design patterns to automate validation of class invariants," *Softw. Test., Verif. Reliab.*, vol. 16, no. 2, pp. 71–95, 2006. 79

[23] B. Meyer, *Eiffel: The Language*. Prentice-Hall, 1991. 77

[24] ——, "Applying "design by contract"," *IEEE Computer*, vol. 25, no. 10, pp. 40–51, 1992. 1

[25] ——, *Object-Oriented Software Construction, 2nd Edition*. Prentice-Hall, 1997. 3

[26] Microsoft Corporation, "Code contracts." [Online]. Available: http://research.microsoft.com/en-us/projects/contracts/ 77

[27] D. J. Murray and D. E. Parson, "Automated debugging in java using ocl and jdi," in *AADEBUG*, 2000. 79

[28] *Object Constraint Language*, Object Management Group Std., Rev. 2.2, 02 2010. [Online]. Available: http://www.omg.org/spec/OCL/2.2 4, 11, 77

[29] *Meta Object Facility*, Object Management Group Std., Rev. 2.4.1, 08 2011. [Online]. Available: http://www.omg.org/spec/MOF/2.4.1/ 4, 11

[30] *Unified Modeling Language*, Object Management Group Std., Rev. 2.4.1, 08 2011. [Online]. Available: http://www.omg.org/spec/UML/2.4.1 1, 4

[31] Oracle Corporation, *Java Debug Interface*. [Online]. Available: http://docs.oracle.com/javase/7/docs/jdk/api/jpda/jdi/ 26

[32] ——, *Java Debug Wire Protocol*. [Online]. Available: http://docs.oracle.com/javase/7/docs/technotes/guides/jpda/jdwp-spec.html 26

[33] ——, *Java Platform Debugger Architecture*. [Online]. Available: http://docs.oracle.com/javase/7/docs/technotes/guides/jpda/ 26

[34] ——, *Java Virtual Machine Tool Interface*. [Online]. Available: http://docs.oracle.com/javase/7/docs/technotes/guides/jvmti/ 26

[35] W. Pree, *Design patterns for object-oriented software development*, ser. ACM Press books. Addison-Wesley, 1994. 65

[36] A. Reder and A. Egyed, "Model/Analyzer: A Tool for Detecting, Visualizing and Fixing Design Errors in UML," in *ASE*, C. Pecheur, J. Andrews, and E. D. Nitto, Eds. ACM, 2010, pp. 347–348. 33

[37] ——, "Computing repair trees for resolving inconsistencies in design models," in *ASE*, M. Goedicke, T. Menzies, and M. Saeki, Eds. ACM, 2012, pp. 220–229. 82

[38] ——, "Incremental consistency checking for complex design rules and larger model changes," in *MoDELS*, ser. Lecture Notes in Computer Science, R. B. France, J. Kazmeier, R. Breu, and C. Atkinson, Eds., vol. 7590. Springer, 2012, pp. 202–218. 47

Appendix A

Evaluation Tools / Applications

A.1 ATM

This is a simple example that simulates an Automated Taller Machine (ATM) and is one of the examples used in the official book for the KeY software verification tool [1] and can be downloaded from http://www.key-project.org/thebook/examples/10UsingKeY/Bank-JML/. The invariants used were defined in the source code as annotations and written in JML, which we translated to OCL. Listing A.1 summarizes all invariants used. Since this application is only used in the book as an example how to formally specify an application and verify its correctness, no user interface is provided. Neither does is come with proper unit tests, but there is a test case simulating sessions at an ATM (insert bank card, enter PIN, ...). This test case has been used for the correctness and performance evaluation.

```
context bank::OfflineAccountProxy
inv:  offlineBalance >=0 and offlineBalance <= 1000

context bank::CentralHost
inv:  accounts->size() = maxAccountNumber
inv:  accounts <> null
inv:  Sequence{1 .. self.maxAccountNumber}->forAll(i | let
   account : bank::PermanentAccount = self.accounts->at(i) in
   account <> null implies account.accountNumber = (i - 1))

context bank::Clock
inv:  clockInstance <> null

context bank::ATM
inv:  Sequence{1 .. self.maxAccountNumber}->forAll(i | let proxy
```

```
   : bank::OfflineAccountProxy = self.accountProxies->at(i) in
   proxy <> null implies proxy.accountNumber = i - 1)
inv:  accountProxies <> null
inv:  accountProxies->size() = maxAccountNumber
inv:  ( online and insertedCard <> null ) implies centralHost.
   accounts->at(insertedCard.accountNumber) <> null
inv:  centralHost <> null
inv:  insertedCard <> null implies
        insertedCard.accountNumber < maxAccountNumber
inv:  insertedCard <> null implies
        insertedCard.accountNumber >= 0
inv:  insertedCard <> null implies not(insertedCard._invalid)
inv:  customerAuthenticated implies insertedCard <> null

context bank::Withdrawal
inv:  amount > 0

context bank::Account
inv:  transactions <> null

context bank::PermanentAccount
inv:  amountForLatestWithdrawalDay >= 0
```

Listing A.1: ATM Invariants

A.2 jPacMan

This application is a simple implementation of the popular Pac-Man game originally developed by Namco in Java. It is just a student project, but nevertheless fully functional and can be downloaded from https://code.google.com/p/in3205/. All invariants are implemented directly in Java as separate methods and checked by using assertions, similar to the example *Node* implementation show in Listing 1.1, Section 1.3. We translated them to OCL and removed the checks in the source code to avoid double checking them. The invariants used are shown in Listing A.2. Although the application comes with a fully functional user interface for playing the game we did not use it for the evaluation. Instead we ran the provided unit tests for the classes defining the applications data model.

```
context jpacman::model::Board
inv:  height >= 0 and width >= 0
```

```
context jpacman::model::Guest
inv:  location <> null implies self = location.inhabitant

context jpacman::model::PlayerMove
inv:  foodEaten >= 0 and thePlayer <> null and self.mover =
   thePlayer

context jpacman::model::Game
inv:  not(playerDied() and playerWon()) and thePlayer.pointsEaten
    <= totalPoints
inv:  theBoard <> null and thePlayer <> null and theStack <> null
    and monsters <> null and totalPoints >= 0

context jpacman::model::MonsterMove
inv:  theMonster <> null and self.mover = theMonster

context jpacman::model::Player
inv:  pointsEaten >= 0

context jpacman::model::Food
inv:  points >= 0

context jpacman::model::Cell
inv:  self.inhabitant <> null implies self = self.inhabitant.
   location
inv:  board <> null and board.withinBorders(x, y)

context jpacman::model::Move
inv:  self.mover <> null and mover.location <> null and (self.
   initialized implies not(self.movePossible() and self.
   playerDies))
```

Listing A.2: jPacMan Invariants

A.3 GanttProject

GanttProject is an open source project management tool, available at http://www.ganttproject.biz/. Its main purpose is to define so called Gantt charts, originally developed by Henry Gantt [14]. Gantt charts are used to outline a projects schedule. The project is broken down into tasks,

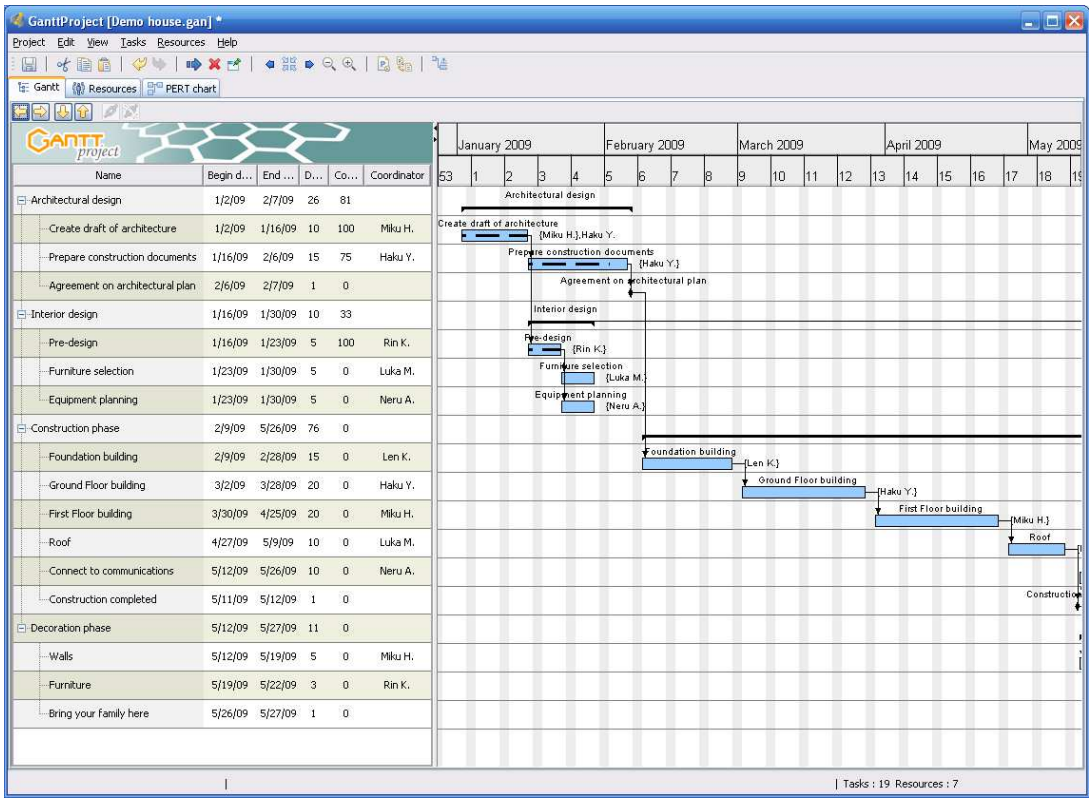


Figure A.1: GanttProject Screenshot

each having a start date and an end date. Those tasks can be hierarchically structured and dependencies defined among them, e. g. a task can only be started if its predecessor is finished. By assigning the completion percentage of the tasks, one can monitor the projects progress. Figure A.1 shows a screenshot of GanttProject while creating a Gantt chart [1]. Additionally, users can define human resources and assign them to tasks.

Unfortunately, GanttProject does not come with any predefined invariants. Thus, we wrote some ourselves by examining test cases or data structures to infer class invariants. Some of the invariants do not actually define property that usually have to hold and were added just for testing purposes. Ultimately, we came up with the invariants shown in Listing A.3.

```
context net::sourceforge::ganttproject::task::dependency::
    TaskDependencyImpl
inv:  self.myDependant.getEnd().compareTo(self.myDependee.
    getStart()) <= 0

context net::sourceforge::ganttproject::GanttTask
```

[1] Screenshot taken from http://en.wikipedia.org/wiki/GanttProject

```
inv:  self.isMilestone <> self.myAssignments.myAssignments.values
   ()->exists(oclAsType(net::sourceforge::ganttproject::task::
   ResourceAssignment).getLoad().round() > 0)
inv:  self.getDependenciesAsDependee().toArray()->forAll(d | d.
   getDependant().getStart().compareTo(self.myStart) <= 0)
inv:  self.myStart <> null implies self.myStart.compareTo(self.
   myEnd) <= 0
inv:  self.isMilestone implies self.myLength.getLength() = 0
inv:  self.getNestedTasks()->isEmpty()

context net::sourceforge::ganttproject::resource::HumanResource
inv:  self.myLoadDistribution.myLoads->forAll(oclAsType(net::
   sourceforge::ganttproject::resource::LoadDistribution::Load).
   load.round() <= 100)
inv:  self.myLoadDistribution.myTasksLoads->forAll(l | let load :
    Integer = l.oclAsType(net::sourceforge::ganttproject::
   resource::LoadDistribution::Load).load.round() in load <> -1
   implies load = 100)
```

Listing A.3: GanttProject Invariants

Glossary

IDE Integrated Development Environment

JDI Java Debug Interface

JDWP Java Debug Wire Protocol

JML Java Modeling Language

JPDA Java Platform Debugger Architecture

JVM Java Virtual Machine

JVM TI Java Virtual Machine Tool Interface

MOF Meta Object Facility

MVC Model-View-Controller

OCL Object Constraint Language

RCP Rich Client Platform

UML Unified Modeling Language

Curriculum Vitae

Personal Data

Name	Sebastian Wilms
Address	Michael-Hainisch-Str. 20
	4040 Linz
E-Mail	seb.wilms@gmail.com
Date of Birth	July 7, 1984
Nationality	Austrian

Work Experience

	July 2012 - Current
Position	Technical Assistant
Organization	Johannes Kepler University Linz
	Institute for Systems Engineering and Automation
Responsibilities	Technical support and software development
	July 2010 - June 2012
Position	Project Assistant
Organization	Johannes Kepler University Linz
	Institute for Systems Engineering and Automation
Responsibilities	Enhancing and maintaining a Consistency Checking Framework
	December 2004 - April 2007
Position	Software Developer and System Administrator
Organization	Wurm & Partner Unternehmensservice GmbH
Responsibilities	Primary focus on maintaining an ERP-System
	for publishing companies
	Maintenance of server infrastructure and second level support

Education

	April 2011 - Current
Awarded qualification	Diplomingenieur (MSc in Software Engineering equivalent)
Organization	Johannes Kepler University Linz
MSc Thesis	Instant Incremental Class Invariant Checking for Java Virtual Machines
	October 2006 - March 2011
Awarded qualification	BSc in Computer Science
Organization	Johannes Kepler University Linz
BSc Thesis	Generating fixing suggestions for inconsistencies in UML Design Models
	September 1998 - June 2003
Awarded qualification	Matura (A-Levels)
Organization	Höhere technische Lehranstalt Vöcklabruck

Languages

German	Native
English	Fluent (spoken and written)
Spanish	Basics

Technical Skills

Operating Systems	Linux, Windows
Programming Languages	C/C++, Java, Scala, Haskell, ...
Databases	SQL (Oracle, mySQL), PROGRESS 4GL
Modeling Languages	MOF, UML, OCL
Frameworks	Eclipse RCP, EMF, JUnit, ...

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Masterarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe. Die vorliegende Masterarbeit ist mit dem elektronisch übermittelten Textdokument identisch.

Linz, am 10. Juli 2013

Sebastian Wilms