Reasoning about Backward Compatibility of Class Libraries

Yannick Welsch

Vom Fachbereich Informatik
der Technischen Universität Kaiserslautern
zur Verleihung des akademischen Grades

Doktor-Ingenieur (Dr.-Ing.)

genehmigte

Dissertation

Datum der wissenschaftlichen Aussprache: 29. August 2013

Dekan:	Prof. Dr. Arnd Poetzsch-Heffter
Vorsitzende der Promotionskommission:	Prof. Dr. Katharina A. Zweig
Erster Berichterstatter:	Prof. Dr. Arnd Poetzsch-Heffter
Zweiter Berichterstatter:	Prof. Dr. Peter Müller

D 386

Abstract

Backward compatibility of class libraries ensures that an old implementation of a library can safely be replaced by a new implementation without breaking existing clients. Formal reasoning about backward compatibility requires an adequate semantic model to compare the behavior of two library implementations. In the object-oriented setting with inheritance and callbacks, finding such models is difficult as the interface between library implementations and clients are complex. Furthermore, handling these models in a way to support practical reasoning requires appropriate verification tools.

This thesis proposes a formal model for library implementations and a reasoning approach for backward compatibility that is implemented using an automatic verifier. The first part of the thesis develops a fully abstract trace-based semantics for class libraries of a core sequential object-oriented language. Traces abstract from the control flow (stack) and data representation (heap) of the library implementations. The construction of a most general context is given that abstracts exactly from all possible clients of the library implementation. Soundness and completeness of the trace semantics as well as the most general context are proven using specialized simulation relations on the operational semantics. The simulation relations also provide a proof method for reasoning about backward compatibility. The second part of the thesis presents the implementation of the simulation-based proof method for an automatic verifier to check backward compatibility of class libraries written in Java. The approach works for complex library implementations, with recursion and loops, in the setting of unknown program contexts. The verification process relies on a coupling invariant that describes a relation between programs that use the old library implementation and programs that use the new library implementation. The thesis presents a specification language to formulate such coupling invariants. Finally, an application of the developed theory and tool to typical examples from the literature validates the reasoning and verification approach.

Acknowledgments

If you make it plain you like people,
it's hard for them to resist liking you back.

— Lois McMaster Bujold

I would like to thank my thesis advisor, Prof. Dr. Arnd Poetzsch-Heffter, for the patient guidance, encouragement and advice he has provided throughout my time as his student. I have been extremely lucky to have a supervisor who cared so much about my work, provided me with much liberties, and who responded to my questions and queries so promptly. I extend my thanks to Prof. Dr. Peter Müller for acting as second reviewer and Prof. Dr. Katharina Zweig for chairing my doctoral committee.

Many thanks go to Mathias Weber and Peter Zeller for participating in the discussions and implementation of the BCVERIFIER tool and to Ilham Kurnia for carefully proof reading this thesis. A very warm thank you also goes to the other current and former members of the Software Technology Group, especially Patrick Michel and Ilham Kurnia for sharing the joys and sorrows of a PhD student.

A special thanks goes to the anonymous reviewers of my papers I came to love and hate at the same time. Their (sometimes harsh) criticism truly helped me improve the presentation of my ideas.

My research was funded by the Federal State of Rhineland Palatinate and the project "Highly Adaptable and Trustworthy Software using Formal Models" (HATS), which is funded by the European Union. I thank all partners of the HATS project, in particular the project lead, Prof. Dr. Reiner Hähnle, for making such a wonderful project possible.

I owe a huge debt of gratitude to my family and friends, providing me with much support to pursue my goals. Last but not least, I thank my life partner Julie for always believing in me.

Contents

List of Figures

List of Definitions, Theorems and Lemmas

Definitions

Theorems

Lemmas

1 Introduction

Computer Science is a science of abstraction
– creating the right model for a problem and devising
the appropriate mechanizable techniques to solve it.

— A. Aho and J. Ullman

Object-oriented libraries are usually realized by the complex interplay of different classes. As libraries evolve over time, adaptations have to be made to their implementations. Sometimes such evolution steps do not preserve backward compatibility with existing clients, classified as *breaking API changes* by Dig and Johnson [DJ06], but often libraries should be modified, extended, or refactored in such a way that client code is not affected. Guaranteeing backward compatibility is especially important for libraries with many users, for example the standard class libraries that are part of the Java platform. Unfortunately, the state of the practice in backward compatibility checking is that only informal guidelines (e.g., [Riv07]) and tools checking simple syntactic aspects (e.g., [EclPDE]) are available.

Reasoning about backward compatibility in a *formal* way requires an adequate semantic model to compare the behavior of two library implementations. In the object-oriented setting with inheritance and callbacks, the model must especially account for the complex interface between library implementations and clients. In addition, the library developer is usually unaware of the client implementations that are using the library. Since the Seventies [Plo77; Mil77] much theoretical work has been done on representation independence and fully abstract semantic models for various calculi and programming languages. It is still an important and popular topic each year at major conferences on theoretical computer science. Meanwhile, a lot of effort has been spent in the last fifteen years on bringing program verification for object-oriented programs to the mainstream by automating the verification process and developing powerful verification tools (e.g., VeriFast [Jac+11], Spec# [BLS05], ESC/Java [Fla+02], KeY [BHS07], JACK [Bar+06], Krakatoa/Why [MPU04], LOOP [BJ01], Jive [MP00]). However, when it comes to verifiers that compare different program parts with respect to their full functional behavior and cover main-stream programming languages such as Java, no such tools exist.

This thesis makes contributions at the theoretical level of studying backward compatibility for class libraries as well as advancing the state of the art in formal verification of backward compatibility. Most of these contributions have been published or are currently under publication [WP13; Wel+13; WP12; Poe+12; DPW12; WP11].

1.1 Challenges and Approach

To illustrate the intricacies of defining an adequate semantic model for object-oriented library implementations, we consider a simple utility library in Figure 1.1 that provides classes to implement the Subject/Observer pattern. The observers, which implement the Observer interface, can be added to the Subject using the addObserver method and are stored in a linked list. The Subject also offers the possibility to get an iterator, basically a cursor, to navigate over the list of registered observers. Furthermore, the Subject provides the convenience method notifyObservers to update all the registered observers with the given argument. Let us further consider a client for the library, in the following also called a *program context*. The program context, defined in Figure 1.2, consists of an implementation of the Observer interface and a main method which uses the Subject. The example illustrates some of the difficulties when dealing with representation independence of OO libraries:

Information hiding: A part of the objects of a library can be provided to the context whereas others can be hidden by the library implementation. For example, LinkedList objects can be used directly by the context (as the LinkedList class is public) as well as serve as an internal representation of the Subject class.

Complex representation: There may be multiple objects (e.g., iterators) available to the context that access shared internal data of the library (e.g., the LinkedList representation).

Type abstraction: The Iterator interface allows abstracting from internal class implementations, i.e., clients of the library do not need to be aware of the implementation type of the iterator. Vice versa, the library does not need to know about possible classes in the program context implementing the Observer interface.

Callbacks: During a notification, i.e., call of the method update, an observer can make a callback to the Subject under consideration. In particular, new observers can be added using the addObserver method during the notification process.

In the following, we give a rough overview of the approach to backward compatibility taken as part of this thesis. A prerequisite for backward compatibility is that the new library implementation provides at least the same interface as the old implementation. In this thesis, we require the new library implementation to be *source compatible* with the old implementation. This means that all client code that compiles against the old library implementation also compiles against the new implementation. Source compatibility solely depends on the well-formedness conditions and type system of the language. As the definition of source compatibility quantifies over all possible clients, which are usually unknown, it is not useful for automatic checking. We thus derive checkable conditions and prove them sound and complete with respect to source compatibility.

The central part of the thesis focuses on behavioral aspects of backward compatibility. Backward compatibility is formalized in terms of a contextual preorder relation: A new library implementation is *backward compatible* with an older implementation if the new library implementation preserves the observable behavior of the old implementation in all possible class contexts of the old implementation. The notion of observation we consider is based on the functional behavior of the library in terms of inputs and outputs, in contrast to quality aspects such as memory consumption or timing behavior. For example, replacing the LinkedList representation of the Subject class by an array is a backward compatible change. Mutual compatibility corresponds to the classical notion of contextual equivalence: Two class library implementations are equivalent if they exhibit the same operational behavior in every possible class context. Comparing the behavior of two library implementations is challenging because (1) the possible contexts are unknown and complex, (2) the stacks and heaps can be significantly different between the library implementations, and (3) the standard definition of backward compatibility does not lend itself to inductive proofs. To meet these challenges, we exploit denotational methods.

```
public interface Observer {
  public void update(Object arg);
}
public class Subject {
  private LinkedList obs;
  public void addObserver(Observer o) { ... }
  public Iterator iterator() { return new ObsIter(); }
  public void notifyObservers(Object arg) {
    while ( ... ) { ... o.update(arg); ... }
  }
  ...
  private class ObsIter implements Iterator {
    private int currIdx;
    ObsIter() { ... }
    public boolean hasNext() { ... }
    public Object next() { ... }
  }
}
public interface Iterator {
  public boolean hasNext();
  public Object next();
}
public class LinkedList { ... }
```

Figure 1.1: Observer example

```
public class IntObs implements Observer {
  private int count = 0;
  public void update(Object arg) {
    count += ((Integer)arg).intValue();
} }

// Body of main method
Subject sj = new Subject();
Observer ob = new IntObs();
sj.addObserver(ob);
sj.notifyObservers(new Integer(5));
```

Figure 1.2: Program context for Observer example

A denotational semantics is called *fully abstract* [Plo77; Mil77] if program parts that have the same denotation are exactly those that are contextually equivalent. In particular, a fully abstract semantics has to abstract from stacks and heaps to meet challenge (2) above. Proving that two sets of classes are equivalent in the (fully abstract) denotational setting amounts to proving that they have the same denotation. We introduce a trace-based semantics that abstracts from the state and heap representation in the old and new implementation. It solves challenges (2) and (3). To obtain a finite representation of all contexts and solve challenge (1), we construct a non-deterministic *most general context* that exhibits exactly all the possible behavior of contexts with respect to the trace-based semantics. The behavior of a library implementation is then expressed as the set of (finite) traces that the most general context exhibits with the library implementation. The central contributions of this thesis are the design of such a fully abstract semantics for sealed packages of a sequential Java subset, a detailed explanation of the full abstraction proof and a method for reasoning about backward compatibility using simulation relations.

We first derive an adequate semantic model for library implementations. A library implementation is formalized in terms of its input/output behavior in order to abstract from its complex representation (heap and stack configurations). The main questions to address are what to consider as the points where such observable behavior occurs and what the input/output information is. As we consider a sequential setting, control flow can at a fixed point in the execution either be in code of the library or code of the program context. As the observer is outside the library, the points of observation are thus those where the program context is executing, which we also call the *observable states*.[1] The observable states are not statically bound to fixed program points such as start and end of methods. For example, calling the method o.update(arg) in line 9 of the Subject class can lead to an observable state where the update method defined in the class IntObs is executing. However, it could also be possible that the library has a class that implements the Observer interface and was registered with the Subject. In that case, calling the update method does not lead to an observable state, as execution stays within code of the library. The behavior of a program context with a library is described by a trace, i.e., a sequence of labels that record the input/output between the context and the library, such as name of the method that was called or values that were passed as part of the method call. The form of the input/output labels is chosen in a way that they capture all relevant information about the behavior of the library.

[1]We use the term *observable states* in allusion to the *visible states* based techniques [Dro+08; Mey97; MPL06]

The trace-based semantics is defined in terms of the interactions between code belonging to the library and code belonging to the program context. We start with a standard operational semantics for our formalized language. In the first step, the operational semantics is augmented in a way that the interactions between library and context can be made explicit. In the second step, traces of interaction labels are used to semantically characterize the library behavior. A non-trivial aspect is the treatment of inheritance, because with inheritance, some code parts of a class or an object might belong to the context and other parts to the library under investigation. Using a standard operational model as a starting point has the advantage that we can use simulation relations applied to standard configurations (i.e., heap, stack) for the full abstraction proof and as a reasoning method for backward compatibility. Furthermore, it provides a direct relation to many program analysis techniques. In the third step, we construct a non-deterministic *most general context* (MGC) that exhibits exactly all the possible behavior of contexts with respect to the trace-based semantics. The behavior of a library implementation is then expressed as the set of finite traces that the MGC exhibits with the library implementation. As mentioned earlier, we give a (standard) formal definition of backward compatibility in terms of a contextual preorder relation, relying on the operational semantics of the language. Using the trace-based semantics and the MGC construction, an alternative definition of backward compatibility between two library implementations can be given in terms of set inclusion between the sets of traces of the library implementations. A large part of this thesis shows how to prove that this definition is equivalent to the previous one, known as *full abstraction* result for our formalized language. The proof relies on specialized simulation relations between the program configurations of program contexts with the old and new library implementation. The simulation relations also provide a basis to reason about backward compatibility. The trace-based definition of backward compatibility can be equated to the existence of a special simulation relation between the MGC with the old library implementation and the MGC with the new library implementation. This relation has to hold in simulating observable program states, i.e., the states where the behavior of the library implementations can be observed. Backward compatibility can thus be proven by stating such a relation (known as coupling invariant) that relates the configurations of both libraries and that holds at corresponding observable states in the execution, and proving that the relation has the simulation property.

1.2 Formal Setting

We formalize our approach for a sequential object-oriented kernel language, called LPJAVA (*Lightweight Package Java*), that essentially extends CLASSIC-JAVA [FKF99] with packages and simple access control. A *library implementation* is a finite set of packages. The design of LPJAVA is motivated as follows:

- LPJAVA covers the core OO features, in particular interfaces, classes, inheritance, and subtyping; in addition, we also include downcasts.
- LPJAVA supports package-local types and private fields as light-weight, language-defined encapsulation mechanisms. This enable the change of several classes from one implementation to the next without breaking compatibility. For example, this makes it possible to exchange data structures implemented by several classes.

The selection of the formal language framework is always a tradeoff. On one hand, it should focus on the essential aspects to keep the proofs manageable. On the other hand, it should be as realistic as possible with respect to the features that are in the focus of the analysis. The focus here is on the type and object interface at package boundaries. In particular, we allow downcasts and exposition of references with a class type, because this is very common in existing OO languages. Furthermore, we focus on encapsulation mechanisms defined by the programming language. Other encapsulation mechanisms (for example defined by additional specification constructs) and more general encapsulation policies are discussed in the related work section.

To keep the formal framework manageable, LPJAVA has some limitations. We assume that packages are sealed (cf. [GPV01], Sect. 2), meaning that once a package is defined no new class and interface definitions can be added to the package. This means that program contexts can not define a class as part of the same package as library classes, thus giving stronger encapsulation guarantees.

Similarly to Jeffrey and Rathke [JR05b], we assume that library implementations are definition-complete, i.e., every type used in the library implementation must be declared in it. This means that library implementations can be type-checked/compiled in isolation, containing all dependencies. This excludes for example library implementations with classes that have unknown superclasses or create objects of an unknown type. This restriction does not limit communication in the OO setting, as due to dynamic dispatch, calling methods defined in the program context from the library code is still possible. For example, the method call `o.update(arg)` in line 9 of the `Subject` class can dispatch to the `update` method defined in the class `IntObs` of the program context. Definition-completeness ensures that the library implementation defines the interface with which it communicates with program contexts. From the point of view of types, the interface is fixed but still allows subtypes. From the point of view of objects, the interface of a library is very dynamic. For example, a list object can dynamically create new iterator objects providing multiple access points to the list.

Definition completeness of libraries introduces an asymmetry between the library and its contexts: A context can create objects of library classes, but not the other way round. To avoid this asymmetry, stronger module concepts with mutual dependencies and well-defined import interfaces would be needed for OO languages. As we developed the fully abstract semantics mainly for reasoning about backward compatibility of existing library implementations, the asymmetry is not a practical limitation.

In LPJava, fields are private. Fields with other accessibility modifiers, in particular public fields, can be simulated by using getter and setter methods with appropriate naming conventions. Thus, the theory can handle public field access, though at the price of syntactical inconvenience. For simplicity, we also decided to only consider public methods in this thesis. From a practical point of view this is a limitation. However, allowing private and package-local methods would not change the behavior at the package boundary and only add further cases to the proofs. Supporting protected methods would complicate source compatibility [WP10], but has no substantial effect on contextual compatibility, because contexts can invoke protected methods in subclasses.

As a further simplification, we only consider finite traces, which makes the presented semantics not amenable to the study of liveness properties. The notion of observation also does not take memory consumption or other non-functional properties such as timing behavior into account. If the old library implementation does not reach an observable state, for example by looping, we allow the new library to behave in an arbitrary way. This means that the new library implementation can show additional behavior in these cases. LPJava is deterministic up to the non-deterministic nature of the memory allocator. As the language cannot directly observe the address of a reference (but check whether another reference points to the same address) and has no pointer arithmetic, we follow the other works in this setting [BN05a; KW07] and use a (bijective) relation between heap locations to talk about corresponding objects in different program runs.

1.3 Contributions

We present the first tool-supported formal verification approach for backward compatibility or equivalence checking of object-oriented libraries. The thesis provides the following technical contributions:

1. We provide a formal model of source compatibility for LPJAVA and a set of sound and complete checkable conditions to ensure source compatibility.
2. We develop a novel fully abstract trace-based semantics for library implementations in LPJAVA. The semantics meets the following challenges:
   a) Information hiding: To achieve full abstraction, objects of the library implementation can be hidden from the context and vice versa. This is realized by distinguishing in the dynamic semantics whether an object is exposed or internal.
   b) Complex representation: The semantics allows arbitrary sharing of internal representations between different interface objects. In particular, the shape of object structures is not restricted, which is prevalent in other works (e.g., [BN05a]).
   c) Type abstraction and downcasts: Subtyping and package-local types allow objects of a package-local class to be exposed at the level of a public supertype. The trace labels do not only keep track of the identity of objects, but also of the types of the object visible to the context. These are the types that the object can be cast to and which determine the methods that can be invoked on the object. The technical challenge here is that objects do not always have a unique public type. For example, the library might expose an object of an internal class type that implements two (disjoint) public interfaces.
   d) Callbacks: The trace semantics naturally accounts for callbacks, having also labels for method calls that are executed by code of the library and result in code of the program context to be executed.

   The fully abstract semantics for LPJAVA packages comes with two technical contributions. First, we present a simulation-based proof technique for full abstraction by augmenting a standard reduction-style small-step operational semantics with additional information. Second, we develop an explicit and finite representation of the most general context. Whereas in other works, the most general context is implicit in the semantics (e.g., [JR05b; Ábr+04]), we separate the most general context from the trace semantics and give it a program representation that is as close as possible to a normal program context. To represent the MGC, the core language only needs to be extended by a single new non-deterministic expression.

3. Inspired by the simulation-based proofs of the full abstraction theorem, we develop a simulation-based reasoning method needed to prove library implementations backward compatible.

4. We present the *Invariant Specification Language* (ISL) to specify the simulations in the form of coupling invariants. ISL is a first-order logic-based assertion language that can express complex data and control flow relations between two library implementations.

5. We present the BCVerifier tool that implements the verification approach using the automatic verifier Boogie. The web frontend [BCVb] and the code of the tool [BCVa] are publicly available for inspection.

6. We show that our theory and tool is applicable to a variety of classic examples from the literature. The examples (and more) are also available on the website [BCVb] and can directly be checked online.

1.4 Outline

The thesis is structured into nine chapters. Chapter 2 introduces the formalized Java dialect LPJava and gives a detailed account of source compatibility. Chapter 3 formalizes the operational and trace-based semantics of LPJava, gives a formal definition of observable behavior and shows that the traces capture exactly the observable behavior. Chapter 4 gives the construction of a most general context (MGC) and shows that the MGC simulates exactly all possible standard program contexts. Chapter 5 presents the "full abstraction" theorem and its proof using specialized simulation relations. Chapter 6 describes a proof method for reasoning about backward compatibility of libraries based on the developed semantics and simulation relations. Chapter 7 shows how the developed reasoning approach can be applied to typical examples in the literature. Chapter 8 gives a more detailed overview of the ISL specification language and discusses the BCVerifier tool. Chapter 9 presents directions for future work and concludes. The related work on the specific contributions of this thesis is presented in the corresponding chapters (Sections 2.5, 5.2, 6.3 and 8.3).

2 Interface Compatibility

A class, in Java, is where we teach objects how to behave.

— R. Pattis

A prerequisite for backward compatibility is that the new library implementation provides at least the interface[1] of the old implementation. In modern object-oriented languages, interfaces of libraries are complex due to the interplay of inheritance, subtyping, namespace mechanisms and accessibility modifiers. We assume that libraries consist of a collection of sealed packages [GPV01], meaning that clients cannot add new class definitions to the packages. In particular, non-public types are not visible to clients, which allows for more interesting changes in new library implementations. We also assume that no introspection, reflection or other magic is used by clients to break the encapsulation properties of libraries. Depending on whether a library is distributed in source or binary form, two notions of *interface compatibility* are relevant.

Source compatibility ensures that every program that *compiles* against the old library implementation also compiles against the new library implementation. In such a case, we call the new implementation source compatible with the old one. For languages with elaborate static encapsulation mechanisms like Java, source compatibility is a complex property and, prior to this work, checking tools did not exist. Checking source compatibility for packages is a difficult task with two central challenges:

- *Complexity:* The complexity of package interfaces is often underestimated. The reason is the intricate interplay of mechanisms to express and restrict subtyping aspects, such as abstract and final types and methods, with

[1]We use the word *interface* here in the general sense, which should not be confused with the programming concept of *Java interfaces*.

the mechanisms to control encapsulation. For example, the information whether or not two non-public types are in a subtype relationship may affect source compatibility of Java packages [WP10].

- *Modularity:* Source compatibility should be checked in a modular way, i.e., without knowing the client code. A checking technique is needed that can abstract from the infinite number of possible client contexts.

As the definition of source compatibility quantifies over all possible client contexts, it cannot directly be used for automatic checking. We thus derive statically checkable conditions for compatibility that are proved necessary and sufficient. Such checkable conditions give interesting insight into the encapsulation of Java packages, allow us to discuss language and program design aspects and provide the basis for package-local refactoring tools.

Typical requirements for source compatibility in Java are that existing public types cannot be removed (but new public types can be added under the restriction that no “star” imports are used by clients). Furthermore, the subtype relation between public types must be preserved. Describing source compatible changes to classes or interfaces is a bit more tricky, as object-oriented libraries often present two interfaces: (1) A *caller interface*, which enables the creation of objects and calling of methods; and (2) an *implementor interface*, which provides the possibility to extend the functionality of the provided types via inheritance. Changes of a class which are compatible with respect to the caller interface can be incompatible for the implementor interface and vice versa. On the one hand, adding a new method to an interface usually breaks compatibility with program contexts that implement the interface but ensures compatibility for callers. On the other hand, widening the return type of a method parameter breaks compatibility for callers but ensures compatibility for the implementors [Riv07] as the covariance of the return types is ensured for overriding methods. To distinguish caller and implementor interface, object-oriented languages support different access modifiers. In Java, these encompass the modifiers **public**, **protected** and **final**. Unfortunately, the modifiers are often not expressive enough. For example, the Java access modifiers do not allow us to specify that a class can only be subclassed in the library but not by clients. One way is to overcome the deficiencies of the language by introducing additional annotations [EclPDE] to refine the description of library interfaces. For example, Eclipse developers use additional annotations to refine the description of library interfaces [EclPDE]. Another way is to accept the limitations and adopt special programming patterns to extend library implementations in a source compatible way. For example, the implementation of the Eclipse IDE uses the concept of *extension interfaces* to guarantee source compatibility. Existing interfaces are not adapted but only new interfaces introduced with the additional methods. Objects have then to be cast to the right interface type if the extension is to be used. Other, more ad hoc solutions exist, but are not the focus of this thesis. Just as an example for such solutions, the developers of the Hamcrest Matcher framework have added a method `void _dont_implement_Matcher___instead_extend_BaseMatcher_()` to the `Matcher` interface.[2]

Binary compatibility ensures that every program that *links* against the binary form of the old library also links against the binary form of the new implementation. It is formally defined in the Java Language Specification (JLS) [Gos+05]. Source and binary compatibility are incomparable; neither one implies the other [Dar08]. However, a set of checkable rules can be established for both forms of interface compatibility. Our theory of backward compatibility was developed in the setting where clients are recompiled with the new library implementation. In the following, only source compatible library implementations are considered. Nonetheless, we believe that large parts of the theory are directly transferable to a setting where binary compatibility is desired instead.

This thesis defines source compatibility for sealed packages of a formalized Java subset with only a limited set of access modifiers. The reason for this is that we want to give a larger focus on behavioral aspects of backward compatibility and a more substantial presentation of typing aspects would deviate from these goals without providing further substantial insight. In other work, however, we have formally explored a larger subset of Java that contained all relevant access modifiers [WP10].[3] In Section 2.1, we formalize a subset of Java, called LPJava. Section 2.2 provides a formalization of the well-formedness conditions and typing rules of the language. Section 2.3 gives a formal definition of source compatibility for LPJava and provides a set of checkable rules that ensure source compatibility. Section 2.4 discusses the impact of source compatibility on language and program design and possible applications. Section 2.5 presents the related work.

[2] http://hamcrest.org/JavaHamcrest/javadoc/1.3/org/hamcrest/Matcher.html

[3] This study led us to find a bug in the Eclipse JDT compiler, see https://bugs.eclipse.org/271303

$$
\begin{aligned}
K,X,Y &::= \overline{P} \\
P &::= \textbf{package}\ p\ ;\ \overline{D} \\
D &::= \textbf{public}^{?}\ \textbf{class}\ c\ \textbf{extends}\ p.c\ \textbf{implements}\ \overline{p.i}\ \{\ \overline{F}\ \overline{M^{\mathsf{c}}}\ \} \\
&\ \ \mid\ \textbf{public}^{?}\ \textbf{interface}\ i\ \textbf{extends}\ \overline{p.i}\ \{\ \overline{M^{\mathsf{a}}}\ \} \\
F &::= \textbf{private}\ p.t\ f\ ; \\
M^{\mathsf{a}} &::= \textbf{public}\ p.t\ m(\overline{p.t\ x})\ ; \\
M^{\mathsf{c}} &::= \textbf{public}\ p.t\ m(\overline{p.t\ x})\ \{\ E\ \} \\
E &::= x \mid \textbf{null} \mid \textbf{new}\ p.c \mid (p.t)E\ \textbf{err}\ E \mid E.f \mid E.f = E \\
&\ \ \mid\ \textbf{let}\ p.t\ x = E\ \textbf{in}\ E \mid E.m(\overline{E}) \mid (E == E\ ?\ E : E) \\
t &::= c \mid i
\end{aligned}
$$

Figure 2.1: Syntax of LPJava

2.1 Formalization of LPJava

The formalized language considered in the following is a sequential object-oriented language called LPJava (*Lightweight Package Java*). It has the standard object-oriented features such as classes, interfaces, inheritance, dynamic dispatch, and mutable state. To allow for interesting scenarios of library evolution, we also consider the Java namespace mechanism to hide certain types in libraries, known under the name of a package system [Gos+05].

Notation. We use the *overbar* notation $\overline{x}$ to denote a finite sequence and the *hat* notation $\widehat{x}$ to denote a finite set. The empty sequence and set are denoted by $\bullet$ and the concatenation of sequence $\overline{x}$ and $\overline{y}$ is denoted by $\overline{x} \cdot \overline{y}$. Concatenation is sometimes implicit by writing terms in juxtaposition, e.g., $\overline{x}\ \overline{y}$. Single elements are implicitly treated as sequences/sets when needed. The function $\mathsf{last}(\ldots)$ returns the last element of a sequence. The expression $\mathcal{M}[x \mapsto y]$ yields the map $\mathcal{M}$ where the entry with key x is updated with the value y, or, if no such key exists, the entry is added. The empty map is denoted by $\varnothing$ and $\mathsf{dom}(\mathcal{M})$ and $\mathsf{rng}(\mathcal{M})$ denote the domain and range of the map $\mathcal{M}$. To improve readability, we often use the underscore place-holder _ instead of a free logical variable that occurs only once in the formula. Proofs are written in a hierarchically structured style as advocated by Lamport [Lam12]. A list of all mathematical symbols and a glossary are available on pages 155 and 159 of the thesis.

Syntax. The syntax of LPJAVA is presented in Figure 2.1. We use the following meta-variables to represent elements of different syntactic categories: c denotes class names (incl. Object), i interface names, p package names (incl. lang), f field names, m method names and x local variable and method parameter names (incl. **this**). As class and interface names can often occur in similar places, we use the meta-variable t to subsume both syntactic categories c and i. For simplicity, we mix syntactic categories with names of their typical elements. The meta-symbol $^?$ denotes an optional item in the grammar. A package, denoted by P, has a name and consists of a sequence of type declarations. We assume that packages are sealed (cf. [GPV01, §2]), meaning that once a package is defined no new class and interface definitions can be added to the package. Types are fully qualified by their package name. Classes and interfaces, denoted by D, can be declared either package-local or public. Primitive data types (like **boolean**, **int**, etc.) are not considered as they do not provide additional insight. Abstract methods, occurring in interfaces, are denoted by $M^{\mathbf{a}}$, whereas methods with bodies, occurring in classes, are denoted by $M^{\mathbf{c}}$. Fields are denoted by F. All methods are assumed to be public and all fields to be private. For reasons of conciseness, we sometimes omit the **public** and **private** modifiers on methods and fields in the following. We assume that every class has a default constructor. Similarly to Java [Gos+05], the default constructor has the same access modifier as its class. LPJAVA has the typical expressions, denoted by E, for a formalized OO subset. In addition, it also allows explicit casting, which leads to more distinguishing power from class contexts. The operator $(p.t)E_1$ **err** E_2 encodes both an *instanceof* and *cast* operator, i.e., it yields the value of E_2 if the value of E_1 cannot be cast to $p.t$. Sequencing of expressions can be done using the expression **let** $p.t\ x = E_1$ **in** E_2. To make the program text more readable, we allow abbreviating the previous expression by $E_1; E_2$ if x does not freely appear in the subexpression E_2.

Terminology. To establish a precise terminology throughout the thesis, we give formal definitions for what we consider as codebase, library implementation, program and program context.

Definition 2.1 (Codebase)
A *codebase* consists of a sequence of packages (i.e., $\overline{P}$) and is denoted by the meta-variables K, X or Y. ◇

Definition 2.2 (Library implementation)
A codebase is called a *library implementation* if it satisfies all the well-formedness conditions of the language, i.e., well-formed type hierarchy, well-typedness of method bodies, etc. Well-formedness of a codebase X is formalized as $\vdash X$ in Figure 2.3 and explained in the next section. $\diamond$

Definition 2.3 (Program)
A *program* is a codebase that has a main class with a main method and that satisfies all the well-formedness conditions of the language. $\diamond$

Definition 2.4 (Program context)
If we join a codebase K (with a main method) and a library implementation X to form a program, we call K a *program context* of X. $\diamond$

To join two codebases into a larger codebase, we write them in juxtaposition (e.g., KX). In the following, we use K for codebases that take the role of program contexts and X and Y for codebases that take the role of library implementations.

Definition 2.5 (Deterministic and most general program contexts)
In order to distinguish standard program contexts from most general (program) contexts of Chapter 4, we call the previous ones *deterministic* program contexts. Program contexts then simply subsume both deterministic and most general program contexts. $\diamond$

2.2 Well-formedness and Typing

In this section, we introduce the notation and rules used to describe the well-formedness and typing conditions for LPJava. Most of the presented relations take the codebase X to be checked as an explicit parameter. Often this parameter is left implicit in language formalizations. However, as we want to talk later on about different codebases when comparing them for compatibility, it is useful to make explicit which codebase we are referring to.

Names. We denote by $\mathsf{uniquenames}_X$ that package declarations in a codebase X do not share the same package name. We further assume that class and interface names in each package are unique and field and method names declared in each type are unique (i.e., we do not consider method overloading). We define $\mathcal{P}_X$ as the set of package names for which there is a package declaration in X. We define $\mathcal{C}_X$ as the set of (qualified) class identifiers for which there is a declaration in X. Similarly, $\mathcal{I}_X$ represents the set of declared interfaces. We then define the set of types $\mathcal{T}_X \overset{\text{def}}{=} \mathcal{C}_X \cup \mathcal{I}_X \cup \{\mathsf{lang.Object}\} \cup \{\perp\}$, where $\perp$ will be used to type the **null** expression and is called the *null type*. To simplify the presentation we use the meta-variable T to denote all kind of types ($T ::= p.c \mid p.i \mid \perp$). The public types of X are characterized by the predicate public_X. In particular, $\mathsf{public}_X(\mathsf{lang.Object})$ and $\mathsf{public}_X(\perp)$ hold for any codebase X under consideration.

Subtyping. The following symbols express relations between the types. For a codebase X, we define the direct subtype relation $<^{\mathbf{d}}_X$ as the least relation with the following properties. The relation contains $p_1.c_1 <^{\mathbf{d}}_X p_2.c_2$ if a class c_1 in package p_1 has an **extends** clause mentioning the class $p_2.c_2$. Similarly, it contains $p_1.i_1 <^{\mathbf{d}}_X p_2.i_2$ if an interface i_1 in package p_1 has an **extends** clause mentioning the interface $p_2.i_2$. If a class c_1 in package p_1 mentions an interface $p_2.i_2$ in its **implements** clause, the relation contains $p_1.c_1 <^{\mathbf{d}}_X p_2.i_2$. The direct subtype relation also contains $p.i <^{\mathbf{d}}_X \mathsf{lang.Object}$ for each interface type $p.i$ in $\mathcal{I}_X$ that has an empty **extends** clause. By $<_X$ we denote the transitive closure of $<^{\mathbf{d}}_X \cup \{(\perp, p.t) \mid p.t \in \mathcal{C}_X \cup \mathcal{I}_X \cup \{\mathsf{lang.Object}\}\}$, by $\leq_X$ the reflexive closure of $<_X$.

Field and method membership. To describe that a type declaration provides methods or fields, we introduce the membership relation $\in_X$. We write $\langle f, p_0.t_0\rangle \in_X p.c$ to describe that a field f of type $p_0.t_0$ is declared in a class c of package p. In contrast to fields which are always private, methods are public and can thus be inherited. The membership symbol $\in_X$ for methods also considers all transitively inherited members. We write $\langle m, \overline{T}, T\rangle \in_X p.t$ to describe that a method m of signature $\overline{T}$ is member of the type $p.t$. The additional type T is used to denote in which supertype the method is implemented if the method is inherited. Method signatures are simply written as a sequence $\overline{T} \overset{\text{def}}{=} (p_0.t_0 \cdot \overline{p_1.t_1})$ where the first element $p_0.t_0$ of the sequence represents the return type of the method and $\overline{p_1.t_1}$ denotes the parameter types. Method membership is defined inductively. We have two base cases (1) If a method m with signature $\overline{T}$ is declared in a class $p.c$ then $\langle m, \overline{T}, p.c\rangle \in_X p.c$. (2) If a method m with signature $\overline{T}$ is declared in an interface $p.i$ then $\langle m, \overline{T}, \perp\rangle \in_X p.i$. We use $\perp$ as a place-holder as methods in interfaces provide no implementation. The inductive steps are then given in the rules below. A method is inherited from a superclass if it is not overridden, i.e., there is no method with the same name and signature declared in the inheriting class (**D-Inh-Method-C**). Methods from superinterfaces are always inherited (**D-Inh-Method-I**).

D-Inh-Method-C

$$\frac{p_1.c_1 <^{\mathbf{d}}_X p_2.c_2 \qquad \langle m, \overline{T}, T\rangle \in_X p_2.c_2 \qquad \langle m, \overline{T}, p_1.c_1\rangle \notin_X p_1.c_1}{\langle m, \overline{T}, T\rangle \in_X p_1.c_1}$$

D-Inh-Method-I

$$\frac{p_1.i_1 <^{\mathbf{d}}_X p_2.i_2 \qquad \langle m, \overline{T}, \bot\rangle \in_X p_2.i_2}{\langle m, \overline{T}, \bot\rangle \in_X p_1.i_1}$$

Judgments. Using the presented notation, we formalize well-formedness of a codebase. The following judgments are used:

- $\vdash X$ denotes that the codebase X is well-formed, i.e., X is a library implementation.
- $X \vdash P$ denotes that the package declaration P is well-formed in codebase X.
- $X, p \vdash D$ denotes that the type declaration D (class or interface) of package p is well-formed in codebase X.
- $X, p.t \vdash M^{\mathbf{a}}$ and $X, p.t \vdash M^{\mathbf{c}}$ denote that the methods $M^{\mathbf{a}}$ and $M^{\mathbf{c}}$ declared in type $p.t$ are well-formed in X.
- $X, p.c, \Gamma \vdash E : T$ denotes that the expression E in class $p.c$ has type T under local variable typing Γ, which is a map from local variable names to types. The result type T represents the exact static type of the expression E.

The well-formedness and typing rules are given in Figures 2.3 and 2.4 and explained in the following.

Context conditions. A codebase X is well-formed (see rule **T-Lib**) if it satisfies the context conditions $C1_X$-$C4_X$. The context conditions for a codebase X are defined in Figure 2.2. Free logical variables in the conditions are universally quantified. Condition $C1_X$ states that types occurring in extends and implements clauses must be accessible. Accessibility, defined as $\mathsf{acc}_X(p.t, p')$ in Figure 2.5, guarantees that a type is defined and either public or part of the same package. The type system guarantees that the type of expressions, if not $\bot$, is accessible in the context of the method in which the expression was defined

$$
\begin{array}{rr}
p_1.t_1 <^{\mathbf{d}}_X p_2.t_2 \Rightarrow \mathsf{acc}_X(p_2.t_2, p_1) & (\mathrm{C1}_X) \\
\langle m, \overline{T_1}, \_\rangle \in_X T \wedge \langle m, \overline{T_2}, \_\rangle \in_X T \Rightarrow \overline{T_1} = \overline{T_2} & (\mathrm{C2}_X) \\
\langle m, \overline{T}, \_\rangle \in_X p_1.i_1 \wedge p_2.c_2 <^{\mathbf{d}}_X p_1.i_1 \Rightarrow \exists T' : \langle m, \overline{T}, T'\rangle \in_X p_2.c_2 & (\mathrm{C3}_X) \\
\mathsf{public}_X(T) \wedge \langle m, \overline{T}, \_\rangle \in_X T \Rightarrow \mathsf{public}_X(\overline{T}) & (\mathrm{C4}_X)
\end{array}
$$

Figure 2.2: Context conditions for a codebase X

T-LIB

$$\frac{\begin{array}{c} \mathsf{uniquenames}_X \\ <_X \text{ is acyclic} \qquad \mathrm{C1}_X\text{-}\mathrm{C4}_X \\ X = \overline{P} \qquad X \vdash \overline{P} \end{array}}{\vdash X}$$

T-PACKAGE

$$\frac{\begin{array}{c} P = \mathbf{package}\ p;\ \overline{D} \\ p \neq \mathsf{lang} \qquad X, p \vdash \overline{D} \\ \exists t : p.t \in \mathcal{C}_P \cup \mathcal{I}_P \wedge \mathsf{public}_P(p.t) \end{array}}{X \vdash P}$$

T-CLASS

$$\frac{\begin{array}{c} \mathsf{acc}_X(\overline{p_1.t_1}, p) \\ X, p.c \vdash \overline{M^{\mathbf{c}}} \end{array}}{X, p \vdash \ldots \mathbf{class}\ c \ldots \{\overline{p_1.t_1\ f\,;}\ \overline{M^{\mathbf{c}}}\}}$$

T-INTF

$$\frac{X, p.i \vdash \overline{M^{\mathbf{a}}}}{X, p \vdash \ldots \mathbf{interface}\ i \ldots \{\overline{M^{\mathbf{a}}}\}}$$

T-METHSIG

$$\frac{\begin{array}{c} (\mathbf{this} \cdot \overline{x}) \text{ pairwise distinct} \\ \mathsf{acc}_X(p_0.t_0 \cdot \overline{p_1.t_1}, p) \end{array}}{X, p.t \vdash p_0.t_0\ m(\overline{p_1.t_1\ x})\ ;}$$

T-METHOD

$$\frac{\begin{array}{c} X, p.c \vdash p_0.t_0\ m(\overline{p_1.t_1\ x})\ ; \\ \Gamma = \emptyset[\mathbf{this} \mapsto p.c][\overline{x} \mapsto \overline{p_1.t_1}] \\ X, p.c, \Gamma \vdash E : T \qquad T \leq_X p_0.t_0 \end{array}}{X, p.c \vdash p_0.t_0\ m(\overline{p_1.t_1\ x})\ \{\ E\ \}}$$

Figure 2.3: Well-formedness conditions for codebases

(see Lemma 2.2). Condition $C2_X$ enforces the absence of method overloading. Condition $C3_X$ ensures that a class implements all the methods of the interfaces that it implements. Condition $C4_X$ states that the signature of (public) methods (defined or inherited) in public classes must only contain public types. The C# language specification [ECM06] defines restrictions that are more general than $C4_X$. In particular, [ECM06, Sect. 10.5.4 on accessibility constraints] presents conditions that among others require parameter and return types to be at least as accessible as the method itself. These additional constraints, discussed in Section 2.4, lead to important properties; they ensure, for example, that public interfaces can always be implemented.

Well-formedness and typing. For a codebase to be well-formed, each package declaration in it must be well-formed (T-LIB). The notation $X \vdash \overline{P}$ is used to abbreviate $\forall P \in \overline{P} : X \vdash P$. A package declaration is well-formed (T-PACKAGE in Figure 2.3) if it does not use the existing package name lang and each type declaration in it is well-formed. An additional sanity condition we assume is that each package declares at least one public type.[4] Class and interface declarations are well-formed if their members are well-formed (T-CLASS and T-INTF). Field and method types must be accessible in the package in which they are used (T-CLASS, T-METHSIG and T-METHOD). The typing of expressions is straightforward and stated mostly for completeness (Figure 2.4). We highlight a few particularities. All types that are explicitly mentioned in the program must be accessible (T-NEW, T-LET and T-CAST). The rule T-IF only allows expressions of comparable type (modeled using the predicate $\mathsf{cmp}_X(T_1, T_2)$ in Figure 2.5) and yields the maximal type of both subexpressions (modeled using the predicate $\mathsf{max}_X(T_1, T_2)$ in Figure 2.5).

2.3 Source Compatibility

A prerequisite for a library implementation to be backward compatible with another one is that whenever the first library implementation joined with a program context yields a program, then the second library implementation joined with the same program context must also yield a program. This property, focusing solely on typing and not behavioral aspects, is called *source compatibility*.

[4]This simplifies the source compatibility conditions by requiring $\mathcal{P}_X = \mathcal{P}_Y$ instead of $\mathcal{P}_X \supseteq \mathcal{P}_Y$.

$$\textsc{T-Null}\quad \frac{}{X, p.c, \Gamma \vdash \textbf{null} : \bot} \qquad \textsc{T-Var}\quad \frac{\Gamma(x) = p_1.t_1}{X, p.c, \Gamma \vdash x : p_1.t_1} \qquad \textsc{T-New}\quad \frac{\mathsf{acc}_X(p_1.c_1, p)}{X, p.c, \Gamma \vdash \textbf{new}\ p_1.c_1 : p_1.c_1}$$

$$\textsc{T-Get}\quad \frac{\begin{array}{c} X, p.c, \Gamma \vdash E : p.c \\ \langle f, p_1.t_1 \rangle \in_X p.c \end{array}}{X, p.c, \Gamma \vdash E.f : p_1.t_1} \qquad \textsc{T-Set}\quad \frac{\begin{array}{c} X, p.c, \Gamma \vdash E_1.f : p_1.t_1 \\ X, p.c, \Gamma \vdash E_2 : T \qquad T \leq_X p_1.t_1 \end{array}}{X, p.c, \Gamma \vdash E_1.f = E_2 : p_1.t_1}$$

$$\textsc{T-Call}\quad \frac{\begin{array}{c} X, p.c, \Gamma \vdash E : p_1.t_1 \\ \langle m, p_0.t_0 \cdot \overline{p_2.t_2}, \_ \rangle \in_X p_1.t_1 \\ \overline{X, p.c, \Gamma \vdash E : T} \qquad \overline{T \leq_X p_2.t_2} \end{array}}{X, p.c, \Gamma \vdash E.m(\overline{E}) : p_0.t_0} \qquad \textsc{T-Let}\quad \frac{\begin{array}{c} X, p.c, \Gamma \vdash E_1 : T_1 \qquad T_1 \leq_X p_1.t_1 \\ \mathsf{acc}_X(p_1.t_1, p) \qquad x \notin \mathsf{dom}(\Gamma) \\ X, p.c, \Gamma[x \mapsto p_1.t_1] \vdash E_2 : T_2 \end{array}}{X, p.c, \Gamma \vdash \textbf{let}\ p_1.t_1\ x = E_1\ \textbf{in}\ E_2 : T_2}$$

$$\textsc{T-If}\quad \frac{\begin{array}{c} X, p.c, \Gamma \vdash E_i : T_i \\ \mathsf{cmp}_X(T_1, T_2) \qquad \mathsf{cmp}_X(T_3, T_4) \\ T = \mathsf{max}_X(T_3, T_4) \end{array}}{X, p.c, \Gamma \vdash (E_1 == E_2\ ?\ E_3 : E_4) : T} \qquad \textsc{T-Cast}\quad \frac{\begin{array}{c} \mathsf{acc}_X(p_1.t_1, p) \\ X, p.c, \Gamma \vdash E_i : T_i \\ \mathsf{cmp}_X(p_1.t_1, T_1) \qquad T_2 \leq_X p_1.t_1 \end{array}}{X, p.c, \Gamma \vdash (p_1.t_1)E_1\ \textbf{err}\ E_2 : p_1.t_1}$$

Figure 2.4: Typing rules for expressions

$$\begin{aligned}
\mathsf{acc}_X(p.t, p') &\overset{\text{def}}{=} p.t \in \mathcal{T}_X \wedge (\mathsf{public}_X(p.t) \vee p = p') \\
\mathsf{cmp}_X(T_1, T_2) &\overset{\text{def}}{=} T_1 \leq_X T_2 \vee T_2 \leq_X T_1 \\
\mathsf{max}_X(T_1, T_2) &\overset{\text{def}}{=} \begin{cases} T_2 & \text{if } T_1 \leq_X T_2 \\ T_1 & \text{if } T_2 \leq_X T_1 \end{cases}
\end{aligned}$$

Figure 2.5: Additional typing functions

2.3.1 Formalization

Using the previous definition of well-formedness, we formalize source compatibility for LPJAVA as follows:

Definition 2.6 (Source compatibility)
A library implementation Y is *source compatible* with a library implementation X if for any codebase K: $\vdash KX$ implies $\vdash KY$. $\diamond$

In particular, the definition states that every program context of X is also a program context of Y. The definition is not useful to compute that a library implementation Y is source compatible with X, because it quantifies over an infinite set of contexts. However, a set of checkable conditions that are necessary and sufficient for Y to be source compatible with X can be given.

Discussion. We explained source compatibility in the introduction as the property between two library implementations that allows all clients which *compiled* against the old library implementation to compile against the new one. However, when considering whether a client can compile against a library implementation, one has to make the distinction between *observability* [Gos+05, §7.3 and §7.4.3] (sometimes also called *visibility*[5]) and *accessibility* of types [Buc10]. Observability is a property of the host platform. At compile time, observability of a type means that the compiler can locate the type definition. For most Java compilers (e.g. `javac`), observability can be influenced by setting the class- or source-path accordingly. Most module systems (e.g., [OSGI]) on the JVM manage observability via class loaders (i.e., at runtime). In the context of this thesis, we do not consider observability and focus on accessibility, which is a property based on the access modifiers of the language (e.g., **private**, **protected**, **public**). When we talk about a context compiling against a library implementation, we assume that all the concerned packages and types are observable.

A restriction of our setting is that we only consider sealed packages. That is, we do not allow contexts to add new classes and interfaces to packages contained in the library implementations X and Y that are compared for compatibility.[6] We selected the above definition from a number of other candidates, mainly because it is simple and can handle interesting practical scenarios. In

[5] Not to be confused with the meaning of visibility in the setting of declaration scopes for programming languages [Gos+05, §6.3.1]

[6] However, context codebases can still extend classes and interfaces from the library implementations.

particular, the compatibility definition allows us to compare single packages that do not import other packages. More importantly, it allows to check compatibility of library implementations X and Y that share common library packages (e.g., `java.util`, etc.).

Other definitions of compatibility are possible. The first alternative is whether to define compatibility for packages or for library implementations. As a package often imports other packages, two package versions might import different packages. If we define compatibility for packages and allow that packages import other packages, we have to be careful about the set of contexts. For example, consider two implementations P_1 and P_2 of a package with name p where P_1 imports a package P_1' and P_2 imports a package P_2' with a different name. Even if P_1 and P_2 are "intuitively" compatible, a context K which is well-formed with P_1P_1' might not be well-formed with P_2P_2' just because it has a conflict with P_2' (e.g., contains a package with the same name). Thus, one can only quantify over codebases K that are not in conflict with P_1 and P_2.

For some situations, the sketched problem can be handled by allowing the hiding of imported packages (as some module systems do). But, as illustrated in earlier work [WP10], there are other situations where even non-public types can affect the well-formedness of entities outside the package. That is why we defined compatibility for library implementations X and Y. Compatibility of packages P_1 and P_2 of the example above is treated in our setting as compatibility of the library implementations $X \overset{\text{def}}{=} P_1P_1'P_2'$ and $Y \overset{\text{def}}{=} P_2P_1'P_2'$.

Considering library compatibility, as we do, has the additional advantage that we can compare library implementations where several packages have new versions. Furthermore, we can allow recursive package dependencies within the library implementation. We also investigated more structured versions of the definition where compared library implementations X and Y import from compatible library implementations X' and Y'. Such a more structured definition would not lead to different results for the problem of this thesis. However, it is a step towards well-understood import interfaces and helpful to check compatibility of large library implementations incrementally. We have investigated such scenarios with "open" packages [DPW12]. But the generalization complicates the definitions and proofs here in such a way that it deviates from the central ideas without adding substantial insight.

2.3.2 Checkable Conditions

In the following we formalize syntactically checkable conditions which guarantee source compatibility for LPJava. The formal definitions, explained below,

$$\mathcal{P}_X = \mathcal{P}_Y \tag{S1$_{X,Y}$}$$

$$\mathsf{public}_X(T) \;\Rightarrow\; \mathsf{public}_Y(T) \tag{S2$_{X,Y}$}$$

$$\mathsf{public}_X(T_1) \wedge \mathsf{public}_X(T_2) \wedge T_1 \leq_X T_2 \Rightarrow T_1 \leq_Y T_2 \tag{S3$_{X,Y}$}$$

$$\mathsf{public}_X(T) \;\Rightarrow\; (\exists T_1 : \langle m, \overline{T}, T_1\rangle \in_X T \;\Leftrightarrow\; \exists T_2 : \langle m, \overline{T}, T_2\rangle \in_Y T\,) \tag{S4$_{X,Y}$}$$

Figure 2.6: Conditions to check source compatibility in LPJAVA

are presented in Figure 2.6. The package names occurring in X must exactly be those occurring in Y (S1$_{X,Y}$). Every public type defined in X must appear in Y (S2$_{X,Y}$). Note that this implies that public class (interface) types declared in X are public class (interface) types declared in Y, because the set of class and interface identifiers are disjoint in LPJAVA. The subtype hierarchy between public types of X must be preserved in Y (S3$_{X,Y}$). Finally, for public types of X, every method which is part of the type (declared or inherited) in X must also have a method with the same signature in Y and vice versa (S4$_{X,Y}$). We state that the conditions are necessary and sufficient in the following theorem. For a rationale we refer to the subsequent proofs.

Theorem 1 (Soundness and completeness of checkable conditions)
A library implementation Y is source compatible with a library implementation X if and only if S1$_{X,Y}$-S4$_{X,Y}$ hold. ◇

PROOF: By Lemmas 2.1 and 2.3. □

Compatibility is a preorder relation, that is, it is reflexive and transitive, but not antisymmetric. Conceptually, compatibility can be seen as a “structural subtype relation” on library implementations (although the exact characterization of what a library signature or interface is will remain implicit in this presentation). We prove both directions of the theorem.

Lemma 2.1 (Completeness of checkable conditions)
Consider two library implementations X and Y. If Y is source compatible with X then S1$_{X,Y}$-S4$_{X,Y}$ hold. ◇

PROOF: By contraposition:
ASSUME: $\vdash X$ and $\vdash Y$ such that S1$_{X,Y}$-S4$_{X,Y}$ do not hold
PROVE: $\exists K : \vdash KX$ and $\nvdash KY$

The proof considers each of the conditions on a per-case basis. For each condition, it assumes that the previous conditions hold. Let us consider a package name p_0 in the following which does neither occur in X nor Y.

$\langle 1\rangle 1.$ CASE: $\neg \mathrm{S1}_{X,Y}$

  $\langle 2\rangle 1.$ CASE: $\exists p \in \mathcal{P}_Y \setminus \mathcal{P}_X$

    $\mathsf{uniquenames}_{KY}$ does not hold if $K \overset{\text{def}}{=}$ **package** p; **public class** c {}

  $\langle 2\rangle 2.$ CASE: $\exists p \in \mathcal{P}_X \setminus \mathcal{P}_Y$

    $\langle 3\rangle 1.$ $\exists t : p.t \in \mathcal{T}_X \wedge \mathsf{public}_K(p.t)$

      By T-PACKAGE

    $\langle 3\rangle 2.$ $p.t \notin \mathcal{T}_Y$

      By assumption $\langle 2\rangle 2$

    $\langle 3\rangle 3.$ Q.E.D.

      T-CLASS ($\mathsf{acc}_{KY}(p.t, p_0)$) does not hold by step $\langle 3\rangle 2$ if $K \overset{\text{def}}{=}$ **package** p_0; **public class** c { $p.t\ f$; }

$\langle 1\rangle 2.$ CASE: $\mathrm{S1}_{X,Y}$ and $\neg \mathrm{S2}_{X,Y}$

    LET: $p.t$ such that $\mathsf{public}_X(p.t) \wedge \neg \mathsf{public}_Y(p.t)$

  T-CLASS ($\mathsf{acc}_{KY}(p.t, p_0)$) does not hold if $K \overset{\text{def}}{=}$ **package** p_0; **public class** c { $p.t\ f$; }

$\langle 1\rangle 3.$ CASE: $\mathrm{S1}_{X,Y}$, $\mathrm{S2}_{X,Y}$ and $\neg \mathrm{S3}_{X,Y}$

    LET: $p_1.t_1, p_2.t_2$ such that $\mathsf{public}_X(p_1.t_1) \wedge \mathsf{public}_X(p_2.t_2) \wedge p_1.t_1 \leq_X p_2.t_2 \wedge p_1.t_1 \not\leq_Y p_2.t_2$

  T-METHOD does not hold if $K \overset{\text{def}}{=}$ **package** p_0; **public class** c { $p_2.t_2\ m(p_1.t_1\ x)$ { x } }

$\langle 1\rangle 4.$ CASE: $\mathrm{S1}_{X,Y}$, $\mathrm{S2}_{X,Y}$, $\mathrm{S3}_{X,Y}$ and $\neg \mathrm{S4}_{X,Y}$

  $\langle 2\rangle 1.$ CASE: $\mathsf{public}_X(p.t) \wedge \langle m, p_1.t_1 \cdot \overline{p_2.t_2}, \_\rangle \in_X p.t \wedge \langle m, p_1.t_1 \cdot \overline{p_2.t_2}, \_\rangle \notin_Y p.t$

    $\langle 3\rangle 1.$ CASE: $\exists \overline{T} : \langle m, \overline{T}, \_\rangle \in_Y p.t \wedge \overline{T} \neq p_1.t_1 \cdot \overline{p_2.t_2}$

      $\langle 4\rangle 1.$ CASE: $t = c$

        $\mathrm{C2}_{KY}$ does not hold if $K \overset{\text{def}}{=}$ **package** p_0; **public class** c_0 **extends** $p.c$ { $p_1.t_1\ m(\overline{p_2.t_2\ x})$ { **null** } }

      $\langle 4\rangle 2.$ CASE: $t = i$

        $\mathrm{C2}_{KY}$ does not hold if $K \overset{\text{def}}{=}$ **package** p_0; **public interface** i_0 **extends** $p.i$ { $p_1.t_1\ m(\overline{p_2.t_2\ x})$; }

    $\langle 3\rangle 2.$ CASE: $\nexists \overline{T} : \langle m, \overline{T}, \_\rangle \in_Y p.t$

      T-CALL does not hold if $K \overset{\text{def}}{=}$ **package** p_0; **public class** c_0 { lang.Object $m(p.t\ x)$ { $x.m(\overline{\textbf{null}})$ } }

  $\langle 2\rangle 2.$ CASE: $\mathsf{public}_X(p.t) \wedge \langle m, p_1.t_1 \cdot \overline{p_2.t_2}, \_\rangle \notin_X p.t \wedge \langle m, p_1.t_1 \cdot \overline{p_2.t_2}, \_\rangle \in_Y p.t$

    $\langle 3\rangle 1.$ CASE: $\exists \overline{T} : \langle m, \overline{T}, \_\rangle \in_X p.t \wedge \overline{T} \neq p_1.t_1 \cdot \overline{p_2.t_2}$

LET: $\overline{T} \overset{\text{def}}{=} p_3.t_3 \cdot \overline{p_4.t_4}$
⟨4⟩1. CASE: $t = c$
C2$_{KY}$ does not hold if $K \overset{\text{def}}{=}$
package p_0; **public class** c_0 **extends** $p.c$ { $p_3.t_3\ m(\overline{p_4.t_4\ x})$ { **null** } }
⟨4⟩2. CASE: $t = i$
C2$_{KY}$ does not hold if $K \overset{\text{def}}{=}$
package p_0; **public interface** i_0 **extends** $p.i$ { $p_3.t_3\ m(\overline{p_4.t_4\ x})$; }
⟨3⟩2. CASE: $\nexists \overline{T} : \langle m, \overline{T}, \_\rangle \in_X p.t$
⟨4⟩1. CASE: $t = c$
C2$_{KY}$ does not hold if $K \overset{\text{def}}{=}$
package p_0; **public class** c_0 **extends** $p.c$ { $p_0.c_0\ m()$ { **null** } }
⟨4⟩2. CASE: $t = i$
C2$_{KY}$ does not hold if $K \overset{\text{def}}{=}$
package p_0; **public interface** i_0 **extends** $p.i$ { $p_0.i_0\ m()$; } □

Lemma 2.2 (Expression type is accessible)
If $\vdash X$ and $X, p.c, \Gamma \vdash E : p_0.t_0$ and $\mathsf{acc}_X(\mathsf{rng}(\Gamma), p)$, then $\mathsf{acc}_X(p_0.t_0, p)$. ◊

PROOF: By induction on the typing derivation of E. □

Lemma 2.3 (Soundness of checkable conditions)
Consider two library implementations X and Y. If S1$_{X,Y}$-S4$_{X,Y}$ hold, then Y is source compatible with X. ◊

PROOF SKETCH: Direct proof:
ASSUME: $\vdash X$ and $\vdash Y$ and S1$_{X,Y}$-S4$_{X,Y}$ hold and $\vdash KX$
PROVE: $\vdash KY$
We first prove a few properties, the main proof then happens in the last step:
⟨1⟩1. $p \in \mathcal{P}_K \wedge \mathsf{acc}_{KX}(p_1.t_1, p) \Rightarrow \mathsf{acc}_{KY}(p_1.t_1, p)$
Directly from Def. of acc and S2$_{X,Y}$
⟨1⟩2. $\mathsf{acc}_{KX}(p_1.t_1, p) \wedge \mathsf{acc}_{KX}(p_2.t_2, p) \wedge p_1.t_1 \leq_{KX} p_2.t_2 \Rightarrow p_1.t_1 \leq_{KY} p_2.t_2$
By S2$_{X,Y}$, S3$_{X,Y}$ and sealed packages
⟨1⟩3. $p.t \in \mathcal{T}_K \Rightarrow (\exists T_1 : \langle m, \overline{T}, T_1\rangle \in_{KX} p.t \Leftrightarrow \exists T_2 : \langle m, \overline{T}, T_2\rangle \in_{KY} p.t)$
We show one direction, the other is similar: PROOF: By induction on the derivation of $\langle m, \overline{T}, T_1\rangle \in_{KX} p.t$ (as $<_{KX}$ acyclic and finite):
⟨2⟩1. Induction basis: $T_1 = p.t$ or $p.t$ is an interface that defines the method
Trivial
⟨2⟩2. Induction step: $T_1 \neq p.t$ and $p.t$ is not an interface that defines the method
Rule D-INH-METHOD-C or D-INH-METHOD-I must apply:

⟨3⟩1. CASE: $p.t <^{\mathbf{d}}_{KX} p_0.t_0 \wedge p_0.t_0 \in \mathcal{T}_K$
By induction hypothesis

⟨3⟩2. CASE: $p.t <^{\mathbf{d}}_{KX} p_0.t_0 \wedge p_0.t_0 \in \mathcal{T}_X$
By $\mathsf{S4}_{X,Y}$

⟨1⟩4. Consider $p.c \in \mathcal{C}_K$ and $\mathsf{acc}_{KX}(\mathsf{rng}(\Gamma), p)$.
If $KX, p.c, \Gamma \vdash E : T$ then $KY, p.c, \Gamma \vdash E : T$.

PROOF: By induction on the typing derivation of E:

⟨2⟩1. Induction basis

⟨3⟩1. CASE: **T-NULL** or **T-VAR**
Trivial

⟨3⟩2. CASE: **T-NEW**
By step ⟨1⟩1

⟨2⟩2. Induction step

⟨3⟩1. CASE: **T-GET**, **T-SET** or **T-LET**
By step ⟨1⟩2 and Lemma 2.2

⟨3⟩2. CASE: **T-IF** or **T-CAST**
By step ⟨1⟩2, Lemma 2.2 and Def. of $\mathsf{cmp}(,)$

⟨3⟩3. CASE: **T-CALL**
By step ⟨1⟩2, Lemma 2.2, step ⟨1⟩3 and $\mathsf{S4}_{X,Y}$

⟨1⟩5. $\vdash KY$

PROOF: By induction on the typing derivation of $\vdash KY$:

⟨2⟩1. Induction basis

⟨3⟩1. CASE: **T-METHSIG**
By step ⟨1⟩1

⟨2⟩2. Induction step

⟨3⟩1. CASE: **T-LIB**

⟨4⟩1. $\mathsf{uniquenames}_{KY}$
Trivial

⟨4⟩2. $<_{KY}$ acyclic
By $<_{KX}$ and $<_Y$ acyclic and $\vdash Y$

⟨4⟩3. $\mathsf{C1}_{KY}$
By $\mathsf{S2}_{X,Y}$

⟨4⟩4. $\mathsf{C2}_{KY}$-$\mathsf{C4}_{KY}$
By $\mathsf{S2}_{X,Y}$-$\mathsf{S3}_{X,Y}$ and step ⟨1⟩3

⟨3⟩2. CASE: **T-PACKAGE**, **T-INTF** or **T-CLASS**
By step ⟨1⟩1

⟨3⟩3. CASE: **T-METHOD**
By step ⟨1⟩2, Lemma 2.2 and step ⟨1⟩4 □

2.4 Discussion

Interfaces are well-understood on the object or type level (programming in the small). Type systems allow compilers to check that type-related programming errors are avoided, e.g., suitable co-/contravariance typing and appropriate choice of access modifiers in overriding methods. However, interface support on the package level (programming in the large) still needs improvement. We envision that future package constructs allow compilers to check for compatibility in the same way as today's compilers check for nominal or structural subtyping. In this section we discuss the impact of source compatibility on language and program design by illustrating issues and solutions for a selected choice of language constructs. We also present possible applications for syntactic compatibility relations.

2.4.1 Language and Program Design

Simpler accessibility system. In Section 2.2 we restricted our language by additional accessibility conditions that go beyond those given in the JLS. Context condition $C4_X$ requires that public methods of public types only have public parameter and return types. For example, while being a legal Java program, we reject the following code as the parameter type of the method m is not public:

```
package p;
public interface i₁ { public i₁ m(i₂ x); }
interface i₂ {}
```

The C# language specification [ECM06] defines similar restrictions. Section 10.5.4 of the specification on accessibility constraints presents conditions that among others require parameter and return types to be at least as accessible as the method itself. These additional constraints lead to important properties; they ensure for example that public interfaces can always be implemented, which is not the case in Java. For example the interface i_1 above cannot be implemented outside of p, although it is public.[7] The constraints further ensure that at each call site the method parameter and return types are types which are accessible to the calling context. Accounting for non-accessible types in these cases complicates the syntactic conditions and proofs.

[7]The same problem can occur for classes, e.g., if an abstract method uses nested types of restricted accessibility as a parameter type.

Codebase	Total methods	Occurrences (number of methods)				
		Accessibility of method:				
		public			**protected**	
		Accessibility of parameter type:				
		protected	**package**	**private**	**package**	**private**
①	77508	5	47		162	
②	47975				2	
③	146397	44	79	17	51	2
④	48604		5			
⑤	164314		35		24	
⑥	16183		4		3	
⑦	2628					
⑧	966		2			
⑨	8966		2			

Legend of codebases considered (with packages):

①	JRE rt.jar Version 1.6.0_16-b01 (com.sun.* | sun.*)
②	JRE rt.jar Version 1.6.0_16-b01 (rest)
③	Eclipse 3.5.1 (org.eclipse.*.internal*)
④	Eclipse 3.5.1 (org.eclipse.* rest)
⑤	NetBeans 6.7.1 (org.netbeans.*)
⑥	ActiveMQ 5.3.0 (*.activemq*)
⑦	BCEL 5.2
⑧	JUnit 4.7
⑨	Lucene 2.9.1

Figure 2.7: Case study of the additional accessibility restrictions. The table presents the number of methods in public types which have the given characteristics.

To substantiate our claim that the restrictions are acceptable and reasonable, we investigated the impact of this simplification on real, industrial-strength Java libraries and programs. We developed a tool [SCWeb] that can analyze huge codebases for *counter examples* violating the rules. To handle full Java code, the tool goes beyond the subset considered in this thesis. In particular, it can handle nested classes with all access modifiers and covers the additional cases for access modifiers in methods' signatures as well. In our analysis, we mainly focused on libraries and frameworks, because they usually provide more interesting encapsulation aspects.

The results are shown in Table 2.7. As we did not eliminate duplicates which occur due to inheritance of methods, a realistic count would lead to even less occurrences. In summary, the number of occurrences (= violations) is very small (blank space indicates no occurrence), and most of them are in ECLIPSE packages containing the name "internal". These packages are, according to the ECLIPSE naming conventions [Ecl12], part of the platform implementation and not part of the exposed API. We found that most of the occurrences were design errors which can easily be fixed. In conclusion, the additional context conditions lead to simpler package interfaces and simplify compatibility checking without imposing restrictions for practical use of the language.

Ambiguous names. We considered the set of package, class and variable identifiers to be disjoint in our formalization, which is not the case in Java. For example, in Java, the name `a.b.c` might refer to the class `c` in package `a.b`, but could as well refer to the static member class `c` of the class `b` in package `a`. To deal with this issue, the JLS provides precedence rules for disambiguation [Gos+05, §6.3.2]. For example, variables will be chosen in preference to types and types will be chosen in preference to packages. As consequence, even adding a private field can be an incompatible change. As an illustrative example, consider the code in Figure 2.8 (inspired by the excellent Java Puzzlers book of Bloch and Gafter [BG05, Puzzle 68]) which consists of the package `p` (which represents the library implementation to be evolved) and the package `k` which represents one possible context. If we add **private static** `d` c_2; as a static field to class c_1, the field obscures the class c_2 and the context expression $p.c_1.c_2.c_3$ does not compile anymore against our new library implementation as the private field c_2 is not accessible.

In our Java subset, we avoided issues of ambiguous type names, as we require all our type names to be fully qualified. For full Java, however, adding a public type to a library implementation can result in ambiguous type names in the context if the context uses "star" imports (e.g., **import** `p.*;`) [Dar08]. Naming

```
package p;

public class c1 {
  public static class c2 { public static Object c3; }
}
public class d { Object c3; }

package k;

class c { Object m(){ return p.c1.c2.c3; }}
```

Figure 2.8: Example for name disambiguation in Java

conventions and programming guidelines in general [Ecl12; Net05; Mic99] help to avoid the issues described above. In particular, one might want to restrict the set of contexts (e.g., contexts should not use “star” imports).

Adding a method. As we have shown in Section 2.3.2, adding a method declaration to an interface is already an API breaking change. In order to deal with this issue, ECLIPSE developers adopt the following convention described in [DJ06]. They create a new interface which extends the old interface with the new method. However, this has the drawback that, in order to use the new interface, clients need to resort to casting in order to access the additional methods. *Defender methods* [Goe11] in Java 8 allow the possibility to provide a default implementation of methods in interfaces, which alleviates some of the problems.

Another possible solution is to give programmers more control over the two different API’s (client and implementor) provided by the interface. For example (as advocated for in [BGP01]), programmers might declare interfaces which can be publicly used from a client point of view, yet only implemented within the same package (This restriction can currently be encoded in Java by adding a dummy method with a non-public parameter type to the interface).

Checked exceptions. Proper handling of checked exceptions is enforced by the JLS [Gos+05, §11.2]. If a method declares that it might throw a checked exception, then call sites have to provide an exception handling block to deal with the exceptions (or alternatively they can just declare throwing the exception themselves). It is thus obvious that, if we change a method in our library implementation such that it can throw a checked exception by adding a **throws** clause, possible calling contexts will stop from compiling.

The more interesting question is whether removing the throws clause of a method is an incompatible change. For API consumers which extend the type where the method is declared, this may break the requirement that overriding methods must not be declared to throw more checked exceptions than the overridden ones [Gos+05, §8.4.6]. For API users, which use the method at calling sites, removing the throws clause is also an incompatible change. This is due to the fact that if a client declares an exception handling block for a checked exception, then there must be a preceding call site of a method which declares throwing such an exception [Gos+05, §11.2.3]. In order to provide better support for this last kind of API users, the JLS could instruct compilers to only issue warnings as this additional restriction does not have an influence on type safety.

Further topics. We have only considered a selected range of programming language constructs which influence compatibility. In our formalized subset, we have not considered constructors with reduced accessibility, static members, nested classes, nested packages, generics and many more Java features.

Future programming languages and module systems should consider such compatibility issues, e.g., a good definition of a module should lead to simple syntactic compatibility conditions. We have seen in this section, that for OO languages, sometimes more static restrictions than in Java (e.g., $C4_X$), as well as less restrictions (e.g., checked exceptions) would provide better support for compatibility. We argue that simpler compatibility checking should be a design goal for programming language design. The aim is to reduce the number of breaking changes by providing the right abstractions for API evolution. A prerequisite is that language designers become aware of compatibility issues when designing the abstractions and well-formedness rules of the language.

2.4.2 Static Compatibility Checking

The ECLIPSE Platform provides guidelines [Riv07] and tools [EclPDE] to support (compatible) API evolution. The tools detect *binary* incompatibilities and usage of non-API code between plug-ins (where API code must be tagged as such). However they do not detect source incompatible changes as presented here.

Instead of defining the syntactic compatibility relation directly on package implementations, as it was done here, it is also possible to derive (syntactic) package signatures from the implementations and then define compatibility based on these signatures (like SML [MTH90] signatures and signature subtyping). This is a two-step process which might lead to better module/package designs. Currently, the package signature is hidden in the definitions of compatibility. A possible application for this is modular typechecking at the package level, e.g., the compiler may not need to know about non-public types to type-check other packages. We have investigated this signature-based type-checking for a subset of the Java language [DPW12].

2.4.3 Package-local Refactoring

There exists a lot of work in the refactoring community to support API evolution. Many solutions focus on creating compatibility layers for the libraries or adapting the client programs (in binary and source setting), for example [BTF05; CN96; Dig+08; Fre06; HD05], but this addresses a different issue. The question, what the API of a library actually is, is left unanswered or only partially answered for a fixed set of clients. Most semantic-preserving refactoring techniques assume that the complete program is available (the typical closed-world view). They allow to reason about semantic preservation of refactorings for one single context, the given program. This might fit well for developers of a single program, but not at all for library or component developers. While most of the existing work which tries to address this issue, e.g., [BTF05; CN96; Dig+08; Fre06; HD05], track modifications of the library and creates compatibility layers or adapts the clients, we consider a setting where no such tracking is needed.

Let us consider for example the *Rename Variable* refactoring as described by Schäfer, Ekman, and Moor [SEM08]. The refactoring should work in such a way that all bindings are preserved, i.e., all accesses to a declaration should be preserved by the renaming. In a complete program, all accesses to a declaration are known. In the setting of refactoring a library, however, this assumption does not hold. However, our syntactic compatibility conditions provide an abstraction for all such accesses from outside the package.

We see two ways to realize package-local refactoring. One way would be to do the refactoring and then check if the new library implementation is (syntactically) compatible with the old one. Another way would be to statically prove that a certain class of refactorings guarantee (syntactic) compatibility. One could also restrict the set of possible contexts by describing acceptable contexts syntactically in the signature (contract) of the package, which would be a prerequisite for true *modular* refactoring.

2.5 Related Work

In this section, we present related work not covered so far. Dmitriev [Dmi02] investigated `make` technology for the Java language, in particular how a change to a class *may* affect other classes. Source incompatible changes (at the class, not package level) are listed in a semi-formal way, but neither proved necessary nor sufficient. To our knowledge there is no other work for object-oriented languages that makes source compatibility the focus of investigation. In the following, we discuss work on behavioral equivalence of object-oriented components that also consider source compatibility, work on binary compatibility, separate compilation, object-oriented module systems, language design aspects, and refactoring techniques.

Behavioral equivalence. Two classes, two packages, or generally two components are called behaviorally equivalent if they have the same interface behavior. Source compatibility is a prerequisite for behavioral equivalence: If two components are not source compatible, there is a context that compiles with one, but not with the other component and thus the components are not equivalent. Koutavas and Wand [KW07] present proof techniques to show that two classes are equivalent, but source compatibility is trivial in the language subset they consider, without packages and with only very restricted use of access modifiers. Closely related to our work is the notion of compatibility by Jeffrey and Rathke [JR05b], which presented a fully abstract trace semantics for a Java-like core language with a package construct. The syntactic characterization of (source) compatibility[8] ([JR05b, §3]) is a prerequisite to a fully abstract semantics of packages. Their paper, however, neither gives a formal definition of the type system nor a proof for source compatibility.

Binary compatibility. Chapter 13 of the Java Language Specification [Gos+05] defines properties for binary compatibility: a set of changes that developers are permitted to make to their packages, classes, or interfaces. This set must guarantee that preexisting class files which linked with the previous (package, class or interface) implementations still link with the current implementations. As mentioned in the JLS [Gos+05, §13.2], binary compatibility is different from source compatibility. For example, introducing a new field, with the same name

[8]Their work provided one of the starting points of our studies.

as an existing field in a subclass of the class containing the existing field declaration, does not break binary compatibility with preexisting binaries. However, at the source code level, this may lead to source incompatibility (typing error). A new declaration is added, changing the meaning of a name in an unchanged part of the source code, while the preexisting binary for that unchanged part of the source code retains the fully qualified, previous meaning of the name.

Binary compatibility gives weaker guarantees to clients than source compatibility. If we consider the case that a library developer has made binary compatible changes to his code, a client developer may not be able to recompile his ongoing project with the new implementation of the library (e.g., if he wants to make some fixes to his client code). Another important issue with compatibility as defined by the JLS is that only (supposedly) sufficient conditions are given (e.g., [Gos+05, §13.3]). Different encapsulation boundaries are also considered by the JLS (e.g., packages, classes and interfaces) which makes it quite a complex chapter in the language specification.

Forman et al. [For+95] have investigated binary compatibility for IBM's System Object Model. They provide a set of transformations which should guarantee compatibility, but do not provide any proofs. Drossopoulou, Wragg, and Eisenbach [DWE98] analyzed binary compatibility as it is defined in the JLS, show that some of the transformations described in the JLS do not guarantee successful linking, and prove their own binary compatibility criteria correct for a Java subset. However, they do not consider whether the criteria they give are necessary conditions for binary compatibility.

Separate compilation. Ancona and Zucca [AZ04] give principal typings for a Java subset without access modifiers. Ancona et al. [Anc+05] also propose a compositional compilation scheme for open codebases, e.g., which do not contain all used types. Although this work has goals different to ours, one of the main common aspects is that they have to find a representation for all possible contexts.

Lagorio [Lag04] investigates how to extract dependency information from Java sources to deal with dependencies.

Module systems and language design. Many module systems [SSP07; JSR277; AZ01; MFH01; Cor+03; Zen05] have been proposed for Java. We focus on the (currently) most popular ones. The OSGi Alliance provides a module system [OSGI] for Java which focuses on the run-time module environment. However, as the module system is not tightly integrated with the Java language, the compile-time module environment may differ from the run-time module environment. Project Jigsaw [Jigsaw] aims at providing a simple, low-level module system to modularize the JDK. It was initially planned for Java 7 but is currently deferred to Java 9.

Most of the aforementioned module systems focus more on visibility issues than on accessibility (as explained at the beginning of Section 2.3). The Java Specification Request (JSR) 294 [JSR294] defines a standard for module accessibility but does not fix the module boundaries. This allows module systems such as [OSGI] to fix module boundaries on top of it.

The existing module systems do not really solve the question what the API of a module is. Very often, this is defined as the aggregation of the API of a set of packages or types. However, it remains unclear what the actual API of a package or type is. With the presented compatibility conditions we aim to initiate further research on alternative definitions of modules and their interplay with compatibility.

The following work studies the Java accessibility modifiers. Müller and Poetzsch-Heffter [MP98] identify the changes that access modifiers in a program can have on the program semantics. Schirmer [Sch04] gives a formalization of the access modifiers and shows interesting runtime properties with respect to access integrity.

3 Trace-based Semantics

[...] operational semantics [...] forces people to think about programs in terms of computational behaviours, based on an underlying computational model. This is bad, because operational reasoning is a tremendous waste of mental effort.

— E.W. Dijkstra, On the cruelty of really teaching computing science

In this chapter, we develop a trace-based semantics for LPJAVA. A trace-based semantics characterizes the behavior of a library implementation with program contexts in terms of sequences of interactions. The crucial advantage of the semantic model is that it abstracts from the complex runtime configurations of an operational model, such as heap and stack.

In the following, we illustrate the trace-based semantics using the example library of the introductory chapter. The traces which are generated by the program that consists of the program context and the utility library in Figures 1.1 and 1.2 on page 4 are of the form

$$\mathsf{call}\ o_1.\mathsf{addObserver}(o_2)\boxed{\downarrow} \cdot \mathsf{rtrn}\ \_\boxed{\uparrow} \cdot \mathsf{call}\ o_1.\mathsf{notifyObservers}(o_3)\boxed{\downarrow} \cdot \\ \mathsf{call}\ o_2.\mathsf{update}(o_3)\boxed{\uparrow} \cdot \mathsf{rtrn}\ \_\boxed{\downarrow} \cdot \mathsf{rtrn}\ \_\boxed{\uparrow}$$

where o_1, o_2, o_3 are arbitrary but distinct object identifiers.[1] In the following, we describe how this trace is constructed.

Program execution starts at line 7 (beginning of the main method) of the program context in Figure 1.2. As LPJAVA only has default constructors, the first interaction happens at line 10, where the method addObserver is called. Execution jumps to the beginning of the body of this method, which is defined

[1]To simplify the presentation for this example, we have omitted some of the information in the trace regarding types.

in the library implementation at line 6 in Figure 1.1. Due to the change in control from the program context to the library, the interaction is marked with the input label call o_1.addObserver(o_2)📥. It contains the information that we have a method call of the method addObserver, that two distinct objects o_1 and o_2 are callee and parameter of the method call and that the direction of the change in control is from the program context to the library, which is denoted by 📥 for input. As library implementations are the focus of our work, we take a library-centric view and use the words input and output from the perspective of the library. Within the body of the addObserver method (which is not shown), the observer is added to the list of observers and the method returns. When the method returns, we have again a change in control, but this time in the reverse direction. The output label rtrn _📤 is thus recorded, which contains the information that the change in control is due to a method return, that no values are passed (void method) and 📤 denotes that this is an output label. We are now back executing in the program context and execute the next statement in line 11 of Figure 1.2 by calling the method notifyObservers. This leads to the label call o_1.notifyObservers(o_3)📥, which contains the information that the method is called on the same object o_1 that we called addObserver before. In the body of the notifyObservers method at line 9 in Figure 1.1, the program iterates over the (singleton) list of registered observers and calls the update method. This leads again to a change in control as the update method for this particular observer has been defined in the program context. This is recorded by the label call o_2.update(o_3)📤 that shows exactly which objects are involved in this call. Finally the update method returns (rtrn _📥), then the notifyObservers method returns (rtrn _📤) and the main method terminates.

The rest of this chapter is structured as follows. We first present in Section 3.1 a standard operational semantics enriched in such a way that the interactions between a library implementation and its program contexts become explicit and call it the *enhanced semantics*. Based on the enhanced semantics, we characterize in Section 3.2 the behavior of a library implementation X in terms of its possible interaction traces with program contexts and define the interaction traces of X with program contexts. Section 3.3 presents the invariants on the program configurations that hold for arbitrary LPJava programs in each state of the enhanced semantics. Section 3.4 gives a formal definition of observable behavior and formalizes backward compatibility in terms of a contextual preorder. Section 3.5 presents an equivalent definition of backward compatibility in terms of the trace-based semantics and states the properties needed to prove said equivalence. Section 3.6 provides a discussion.

3.1 Operational Model

The small-step operational semantics of LPJava is presented in the style of FeatherweightJava [IPW01] and ClassicJava [FKF99]. A configuration has the form $\mathcal{S} \overset{\text{def}}{=} KX, \mathcal{O}, \overline{\mathcal{F}}$ where KX is the program consisting of the program context K and the library implementation X that is run, $\mathcal{O}$ denotes the heap and $\overline{\mathcal{F}}$ the stack. In order to generate the traces and realize the most general context, the configurations are augmented with additional information. They are for example aware of which part of the code belongs to the program context and which part to the library implementation. The additional information does, however, not change the standard operational behavior.[2] The syntax of configurations is given in Figure 3.1, where the augmented information is highlighted.

$\mathcal{S} ::= KX, \mathcal{O}, \overline{\mathcal{F}}$	configuration
$v ::= o \mid \textbf{null}$	value
$\mathcal{O} ::= o \mapsto (V, L, p.c, \mathcal{G})$	heap
$V ::= \mathsf{exposed} \mid \mathsf{internal}$	exposure flag
$L ::= \mathsf{ctxt} \mid \mathsf{lib}$	origin location
$\mathcal{G} ::= (p.c, f) \mapsto v$	field mapping
$\mathcal{F} ::= E_L{:}p.t \mid \mathcal{E}_L{:}p.t$	stack slice
$E ::= \ldots \mid v$	extended expressions

Figure 3.1: Semantic entities for LPJava, where $o \in$ object identifiers

Heap. Values occurring in the runtime configurations, denoted by v, can either be object identifiers o or the special value **null**. The heap $\mathcal{O}$ is a map from object identifiers to heap entries. Heap entries consist of the special flags L and V, the runtime class type $p.c$ of the object and field values $\mathcal{G}$. The flag L ranges over $\{\mathsf{ctxt}, \mathsf{lib}\}$ and indicates whether the object has been created by code of the program context K or the library implementation X. The flag V ranges over $\{\mathsf{exposed}, \mathsf{internal}\}$ and is used in heap entries to denote whether an object created in the context has been exposed (i.e., made known) to the library or vice versa. The field mapping $\mathcal{G}$ maps pairs of field name and class name to

[2]We do not formally postulate this claim in the thesis.

$$\begin{aligned}\mathcal{E} ::= & \lfloor\rfloor \mid \mathcal{E}.f \mid \mathcal{E}.f = E \mid v.f = \mathcal{E} \mid \textbf{let } p.t\ x = \mathcal{E} \textbf{ in } E \mid (p.t)\mathcal{E} \textbf{ err } E \mid \mathcal{E}.m(\overline{E}) \\ \mid\ & v.m(\overline{v} \cdot \mathcal{E} \cdot \overline{E}) \mid (\mathcal{E} == E\ ?\ E : E) \mid (v == \mathcal{E}\ ?\ E : E)\end{aligned}$$

Figure 3.2: Evaluation contexts for LPJAVA

values. The class name determines the class where the field was defined and is used to distinguish fields with same name that occur in different classes of the subtype hierarchy.

Stack. The stack is represented as a sequence of stack slices $\overline{\mathcal{F}}$. This partitioning of the stack into stack slices allows marking parts of the stack as belonging to the program context or the library. A stack slice consists of a typed expression or evaluation context and flag L. Expressions that appear in stack slices can, in contrast to the expressions of the previous chapter, contain values v. An evaluation context $\mathcal{E}$ (see [WF94]) is an expression with a *hole* $\lfloor\rfloor$ somewhere inside the expression. We write $\mathcal{E}\lfloor E\rfloor$ to mean that the hole in $\mathcal{E}$ is replaced by expression E. A hole in $\mathcal{E}$ can only appear at certain positions, as defined in Figure 3.2. The topmost stack slice in the stack contains an expression E and all other stack slices contain an evaluation context $\mathcal{E}$. Stack slices are associated with either the library implementation X or the context K using the flag L. It is used in stack slices to mark if the code (E or $\mathcal{E}$) that is part of this stack slice originates from X or K.

3.2 Traces

The operational rules for a program are based on a labeled small-step reduction judgment of the form $\mathcal{S} \overset{\gamma}{\rightsquigarrow} \mathcal{S}'$, defined in Figures 3.3 and 3.5. We say that X *controls execution* if code of X is executed; otherwise K controls execution. The function exec($\mathcal{S}$) from Figure 3.7 is used to determine who controls execution in a particular configuration $\mathcal{S}$. An interaction is a change of control. Labels indicate whether a change of control happened in a particular step. The syntax of labels is given in Figure 3.4. Interaction is considered from the viewpoint of the library. Input labels (marked by ⏬) express a change of control from the context to the library; output labels (marked by ⏫) express a change of control from the library to the context. Transitions which do not express a control flow change are marked as silent transitions with the label τ. In our language, changes of control can only happen via method calls or returns. To indicate this, there are input and output labels for method invocation and return. The labels for method invocation and return include the parameter and result values together with their *abstracted types*, the rationale for which is given later.

Local steps. We give a short overview of the rules in Figure 3.3. We use the helper relation $\rightsquigarrow^{L}_{KX}$, defined by rule R-INTERNAL-STEP, to denote τ steps that are local to an evaluation context in a stack slice. We first describe the rules that encompass such local steps. Object allocation is given by rule R-NEW. A new object identifier is non-deterministically chosen and a heap entry for it is created. The new object is marked as $\mathsf{internal}$ and its origin L is set depending on who controls execution. The fields of the object are initialized with default values using the function $\mathsf{initf}_{KX}(p.c)$ defined in Figure 3.7. The cast operator is described using the rule R-CAST and uses the function $\mathsf{type}_{\mathcal{O}}(v)$ defined in Figure 3.7 to get the dynamic type of the value v. The rule R-LET shows how the values of local variables are substituted in the follow-up expression. Field access and writes are formalized using the rules R-GET and R-SET. Here, we assume that the class name of the class which defines the field was attached to the field name in the type-checking phase (e.g., $f_{p.c}$). The condition operator, described by R-IF, checks whether the two values are equal to select one of the sub-expressions. The rule R-CALL-INTERN considers method calls where control of execution stays within the library. The function $\mathsf{select}_{KX}(L)$, defined in Figure 3.7, yields the program context K or the library implementation X, depending on the value of L. The crucial part of the rule is that it requires that the body of the method m is defined in a class $p.c$ that is part of the program context K, if K controls execution, or part of the library implementation X, if X controls execution. The function $\mathsf{body}_{KX}(p.c, m)$ yields the names of the formal parameters and the body of the method m defined in the class $p.c$.

Interactions. The last two rules, presented in Figure 3.5, handle changes of control, i.e., interactions. The rule R-CALL-BOUNDARY considers changes of control that are caused by a method call, contrasted to the previous rule R-CALL-INTERN with the negation $\neg L$ in the rule antecedent $p.c \in \mathcal{C}_{\mathsf{select}_{KX}(\neg L)}$. Negation, defined in Figure 3.7, simply yields the *other* value in a two-valued domain. Interactions allocate or deallocate stack slices in contrast to method calls within the context or the library implementation that are handled within the same stack slice (see R-CALL-INTERN). In case of a method call, a new stack slice $\mathcal{F}'$ is

R-INTERNAL-STEP
$$\frac{\mathcal{O}, E \rightsquigarrow^{L}_{KX} \mathcal{O}', E'}{KX, \mathcal{O}, \mathcal{E}\lfloor E \rfloor_{L}{:}p.t \cdot \overline{\mathcal{F}} \overset{\tau}{\rightsquigarrow} KX, \mathcal{O}', \mathcal{E}\lfloor E' \rfloor_{L}{:}p.t \cdot \overline{\mathcal{F}}}$$

R-NEW
$$\frac{o \notin \mathsf{dom}(\mathcal{O}) \qquad \mathcal{G} = \mathsf{initf}_{KX}(p.c) \qquad \mathcal{O}' = \mathcal{O}[o \mapsto (\mathsf{internal}, L, p.c, \mathcal{G})]}{\mathcal{O}, \textbf{new}\ p.c \rightsquigarrow^{L}_{KX} \mathcal{O}', o}$$

R-IF
$$\frac{E' \overset{\text{def}}{=} \begin{cases} E_1 & \text{if } v_1 = v_2 \\ E_2 & \text{otherwise} \end{cases}}{\mathcal{O}, (v_1 == v_2\ ?\ E_1 : E_2) \rightsquigarrow^{L}_{KX} \mathcal{O}, E'}$$

R-CAST
$$\frac{E' \overset{\text{def}}{=} \begin{cases} v & \text{if } \mathsf{type}_{\mathcal{O}}(v) \leq_{KX} p.t \\ E & \text{otherwise} \end{cases}}{\mathcal{O}, (p.t)v\ \textbf{err}\ E \rightsquigarrow^{L}_{KX} \mathcal{O}, E'}$$

R-LET
$$\mathcal{O}, \textbf{let}\ p.t\ x = v\ \textbf{in}\ E \rightsquigarrow^{L}_{KX} \mathcal{O}, E[v/x]$$

R-GET
$$\frac{\mathcal{O}(o) = (\_,\_,\_,\mathcal{G})}{\mathcal{O}, o.f_{p.c} \rightsquigarrow^{L}_{KX} \mathcal{O}, \mathcal{G}(p.c, f)}$$

R-SET
$$\frac{\mathcal{O}(o) = (V, L', p'.c', \mathcal{G}) \qquad \mathcal{G}' = \mathcal{G}[(p.c, f) \mapsto v]}{\mathcal{O}, o.f_{p.c} = v \rightsquigarrow^{L}_{KX} \mathcal{O}[o \mapsto (V, L', p'.c', \mathcal{G}')], v}$$

R-CALL-INTERN
$$\frac{\begin{array}{c} \langle m, \_, p.c \rangle \in_{KX} \mathsf{type}_{\mathcal{O}}(o) \\ p.c \in \mathcal{C}_{\mathsf{select}_{KX}(L)} \qquad \mathsf{body}_{KX}(p.c, m) = (\overline{x}, E) \end{array}}{\mathcal{O}, o.m(\overline{v}) \rightsquigarrow^{L}_{KX} \mathcal{O}, E[o/\textbf{this}, \overline{v}/\overline{x}]}$$

Figure 3.3: Rules for the enhanced small-step semantics (local steps)

$$\begin{array}{rcll}
\gamma & ::= & \mu\text{⍐} \mid \mu\text{⍗} \mid \tau & \text{label} \\
\mu & ::= & \mathsf{call}\ o^{\alpha}.m(\overline{v^{\alpha}}) & \text{call message} \\
 & \mid & \mathsf{rtrn}\ v^{\alpha} & \text{return message} \\
o^{\alpha} & ::= & o{:}T^{\alpha} & \text{abstracted object} \\
v^{\alpha} & ::= & o^{\alpha} \mid \mathbf{null} & \text{abstracted value} \\
T^{\alpha} & ::= & \langle \widehat{p.t}, \widehat{m} \rangle & \text{abstracted type}
\end{array}$$

Figure 3.4: Syntax of traces, i.e., sequences of labels $\overline{\gamma}$

created that contains the body of the called method. The stack slice is typed, containing the return type $p'.t'$ of the method that was invoked. At this part of the presentation, the type provides no additional use, but will be useful to model the most general context. The interaction created by the method call generates a label γ using the function $\mathsf{mcall}_{KX}(o, m, \overline{v}, \mathcal{O}, L)$ that is defined as

$$\mathsf{call}\ \mathsf{valabs}_{KX}(o, \mathcal{O}).m(\mathsf{valabs}_{KX}(\overline{v}, \mathcal{O}))\ \mathsf{from}(L)$$

using helper functions in Figure 3.7. The choice as to whether an input or output label is to be generated is done using the extra information L tagged to the current stack slice. The function $\mathsf{from}(L)$ yields an input label if the program context was executing, and an output label otherwise. The label also contains the receiver and parameter values of the method call together with their abstracted types. The function $\mathsf{valabs}_{KX}(v, \mathcal{O})$ yields an abstracted value, which amounts to adding an abstracted type to the value, if it is an object identifier. The functions $\mathsf{typeabs}_{KX}(p.c)$ and $\mathsf{expose}(v, \mathcal{O})$ are presented in the next paragraphs, and followed by an explanation of the rule **R-Return-Boundary**.

Type abstraction. The idea behind the type abstraction is to allow the possibility to compare labels of different library implementations in a way that is independent from program contexts. Thus, we abstract from types declared in the context (e.g., IntObs in Figure 1.2). Similarly, local types do not appear in the labels, as they cannot be observed by program contexts. This allows library implementations to use different local types, i.e, renaming the ObsIter class in Figure 1.1 should not matter to program contexts. Types in labels are *abstracted* (see $\mathsf{typeabs}_{KX}(p.c)$ in Figure 3.7) to a representation which only preserves the information (1) which public supertypes of $p.c$ belong to the library and (2) which of their methods are not overridden by the context. In our example, the type of ObsIter objects is abstracted

R-CALL-BOUNDARY

$$\dfrac{\begin{array}{ll}\langle m, p'.t' \cdot \overline{T}, p.c\rangle \in_{KX} \mathsf{type}_{\mathcal{O}}(o) & p.c \in \mathcal{C}_{\mathsf{select}_{KX}(\neg L)} \\ \mathsf{body}_{KX}(p.c, m) = (\overline{x}, E) & \gamma = \mathsf{mcall}_{KX}(o, m, \overline{v}, \mathcal{O}, L) \\ \mathcal{O}' = \mathsf{expose}(o \cdot \overline{v}, \mathcal{O}) & \mathcal{F}' = E[o/\mathbf{this}, \overline{v}/\overline{x}]_{\neg L}{:}p'.t'\end{array}}{KX, \mathcal{O}, \mathcal{E}\lfloor o.m(\overline{v})\rfloor_L{:}p.t \cdot \overline{\mathcal{F}} \overset{\gamma}{\rightsquigarrow} KX, \mathcal{O}', \mathcal{F}' \cdot \mathcal{E}_L{:}p.t \cdot \overline{\mathcal{F}}}$$

R-RETURN-BOUNDARY

$$\dfrac{\gamma = \mathsf{mrtrn}_{KX}(v, \mathcal{O}, L) \qquad \mathcal{O}' = \mathsf{expose}(v, \mathcal{O})}{KX, \mathcal{O}, v_L{:}p'.t' \cdot \mathcal{E}_{\neg L}{:}p.t \cdot \overline{\mathcal{F}} \overset{\gamma}{\rightsquigarrow} KX, \mathcal{O}', \mathcal{E}\lfloor v\rfloor_{\neg L}{:}p.t \cdot \overline{\mathcal{F}}}$$

Figure 3.5: Rules for the enhanced small-step semantics (interactions)

to ⟨{lang.Object, Iterator}, {hasNext, next}⟩ and the type of IntObs objects is abstracted to ⟨{lang.Object, Observer}, •⟩. The reason for (1) is that these are the types of X that can be used in cast expressions in the context. Based on the label, it becomes thus clear which cast expressions will succeed and which not. The reason for (2) is that, based on the label, we know the methods that are or are not overridden by the context and, if invoked on the receiver object, thus lead to changes of control. As X defines a finite set of types (denoted by $\mathcal{T}_X$), there are only a finite set of abstracted types that can occur in traces with X. This set is denoted by $\mathcal{T}_X^\alpha$ and can be constructed from X.

Information hiding. Finally, the values that are passed from the library implementation to the program context or vice versa are exposed, denoted using the function $\mathsf{expose}(o \cdot \overline{v}, \mathcal{O})$. This means that the semantics tracks exactly which objects of the library implementation are known to the program context and vice versa. Remember that objects are always created internally.

The rule R-RETURN-BOUNDARY shows the situation where control of execution is caused by a method return. In that case, a label is created using the function $\mathsf{mrtrn}_{KX}(v, \mathcal{O}, L)$ that is defined as

$$\mathsf{rtrn}\ \mathsf{valabs}_{KX}(v, \mathcal{O})\ \mathsf{from}(L)$$

and the value is exposed in the same way as for method calls. Finally the stack slice is deallocated and the result inserted into the evaluation context of the stack slice below.

$$\textsc{L-Step}\quad \frac{\overbrace{\mathcal{S} \overset{\tau}{\rightsquigarrow} \ldots \overset{\tau}{\rightsquigarrow}}^{i \text{ times, } i \in \mathbb{N}} \mathcal{S}' \qquad \mathcal{S}' \overset{\gamma}{\rightsquigarrow} \mathcal{S}'' \qquad \gamma \neq \tau}{\mathcal{S} \xrightarrow{\gamma} \mathcal{S}''} \qquad \textsc{L-Empty}\quad \frac{}{\mathcal{S} \xrightarrow{\bullet} \mathcal{S}} \qquad \textsc{L-Accum}\quad \frac{\mathcal{S} \xrightarrow{\overline{\gamma}} \mathcal{S}' \qquad \mathcal{S}' \xrightarrow{\gamma} \mathcal{S}''}{\mathcal{S} \xrightarrow{\overline{\gamma}\cdot\gamma} \mathcal{S}''}$$

Figure 3.6: Rules for large-step semantics

Large-step semantics. In the following, we consider (interaction) traces as sequences of labels $\overline{\gamma}$ which are generated by steps of the enhanced operational semantics. To abstract from silent τ steps of a computation, we provide a large step version of the enhanced semantics (denoted $\xrightarrow{\overline{\gamma}}$) that is inductively defined in Figure 3.6. Every large step (L-Step) represents a finite number of τ steps followed by a non-τ step. Note that τ does not appear in labels of large steps. Large steps represent the execution of small steps up to the state right after the next non-τ label has been generated. The large step semantics then accumulates the non-τ labels into a sequence (L-Empty and L-Accum).

As can be seen from the transition rules, evaluation is deterministic (up to object naming). In order to deal with the non-deterministic choice of fresh object identifiers in the traces, we introduce (object) renamings.

Definition 3.1 (Renaming ρ)
A *renaming* is a bijective relation on object identifiers. We write ρ for such a relation. ◇

We can then consider traces equivalent modulo a renaming. We have equivalent traces $\overline{\gamma_1} \equiv^{\rho} \overline{\gamma_2}$ iff the object identifiers appearing at the same positions in the traces are related under the renaming ρ and the types appearing at the same position are equal. In the following, we use the straightforward generalization of this definition of *equivalence modulo a renaming* ($\equiv^{\rho}$) to arbitrary syntactic terms.

Definition 3.2 (Equivalence on terms ($\equiv^{\rho}$))
Two syntactic terms are equivalent modulo a renaming (written $\equiv^{\rho}$) if the object identifiers appearing at the same positions in the terms are related under the renaming ρ and the remaining parts of the terms are syntactically equal. If we are not interested in a particular ρ, we omit it for brevity. ◇

Initial configuration. As described in Definition 2.4, a program context is a codebase that has a main class $p.c$ with a main method lang.Object main(), where the class $p.c$ is also called a *startup class*. It is executed by calling main. In the following we assume without loss of generality that the startup class has the name main.Main and is defined in the context K.

Definition 3.3 (Initial configuration $\mathcal{S}^{\textbf{init}}_{KX}$)
The initial configuration $\mathcal{S}^{\textbf{init}}_{KX}$ is defined as $KX, \mathcal{O}, \mathcal{F} \cdot \bullet$ where

- $\mathcal{O} \overset{\text{def}}{=} \varnothing[o \mapsto (\mathsf{internal}, \mathsf{ctxt}, \mathsf{main.Main}, \mathcal{G})]$,
- $\mathcal{G} \overset{\text{def}}{=} \mathsf{initf}_{KX}(\mathsf{main.Main})$,
- $\mathcal{F} \overset{\text{def}}{=} E[o/\textbf{this}]_{\mathsf{ctxt}}\text{:}\mathsf{lang.Object}$,
- o is an arbitrary object identifier, and
- $(\_, E) \overset{\text{def}}{=} \mathsf{body}_{KX}(\mathsf{main.Main}, \mathsf{main})$. ◇

Our goal in this subsection was to characterize the behavior of a library implementation X in terms of its possible interaction traces with program contexts, which we achieve by the following definition.

Definition 3.4 (Trace semantics)
The *traces* of a library implementation X with a program context K are given by $\mathsf{traces}(KX) \overset{\text{def}}{=} \{\overline{\gamma} \mid \exists \mathcal{S} : \mathcal{S}^{\textbf{init}}_{KX} \xrightarrow{\overline{\gamma}} \mathcal{S}\}$ ◇

Note that $\mathsf{traces}(KX)$ is closed with respect to renaming, i.e., if $\overline{\gamma_1} \in \mathsf{traces}(KX)$ and $\overline{\gamma_1} \equiv \overline{\gamma_2}$, then also $\overline{\gamma_2} \in \mathsf{traces}(KX)$. Furthermore, $\mathsf{traces}(KX)$ is prefix-closed and only refers to public types in X.

3.3 Well-formed Configurations

Well-formedness conditions on runtime configurations tell us about the invariants that hold during program runs (i.e., how we expect runtime configurations to look like). For example, absence of dangling pointers or well-typedness of references are standard well-formedness conditions. Before giving the definition of well-formed runtime configuration (which also subsumes type soundness), we first present a few helper functions (formally defined in Figure 3.7). The function $\mathsf{stackabs}_L(\overline{\mathcal{F}})$ yields all the L-tagged stack slices of $\overline{\mathcal{F}}$ and $\mathsf{fields}^L_{KX}(\mathcal{G})$ restricts $\mathcal{G}$ to fields that are defined in classes of L. The function $\mathsf{filter}(\mathcal{O}, L)$ returns all object identifiers of objects in $\mathcal{O}$ that are tagged as L (similar for V).

$$
\begin{array}{ll}
\neg L & \overset{\text{def}}{=} L' \text{ where } L \neq L' \text{ (similar for } \neg V\text{)} \\
\mathsf{abs}_X(\langle \widehat{p.t}, \widehat{m} \rangle) & \overset{\text{def}}{=} \langle \widehat{p.t}', \widehat{m}' \rangle \text{ where} \\
& \qquad \widehat{p.t}' \overset{\text{def}}{=} \widehat{p.t} \cap \{p.t \mid p.t \in \mathcal{T}_X \wedge \mathsf{public}_X(p.t)\} \text{ and} \\
& \qquad \widehat{m}' \overset{\text{def}}{=} \widehat{m} \cap \{m \mid \langle m, \_, \_ \rangle \in_X p.t \wedge p.t \in \widehat{p.t}'\} \\
\mathsf{available}(\mathcal{O}, L) & \overset{\text{def}}{=} \mathsf{filter}(\mathcal{O}, \mathsf{exposed}) \cup \mathsf{filter}(\mathcal{O}, \mathsf{internal}, L) \\
\mathsf{body}_{KX}(p.c, m) & \overset{\text{def}}{=} (\overline{x}, E) \text{ where } \overline{x} \text{ are the formal parameters} \\
& \qquad\qquad \text{and } E \text{ is method body of } m \text{ in } p.c \\
\mathsf{expose}(v, \mathcal{O}) & \overset{\text{def}}{=} \begin{cases} \mathcal{O} & \text{if } v = \mathbf{null} \\ \mathcal{O}[v \mapsto (\mathsf{exposed}, L, p.c, \mathcal{G})] & \text{if } \mathcal{O}(v) = (\_, L, p.c, \mathcal{G}) \end{cases} \\
\mathsf{exec}(KX, \mathcal{O}, \overline{\mathcal{F}}) & \overset{\text{def}}{=} L \quad \text{if } \overline{\mathcal{F}} = E_L{:}p.t \cdot \overline{\mathcal{F}'} \\
\mathsf{fields}^L_{KX}(\mathcal{G}) & \overset{\text{def}}{=} \{(p.c, f) \mapsto v \mid ((p.c, f) \mapsto v) \in \mathcal{G} \text{ and } p.c \in \mathcal{C}_{\mathsf{select}_{KX}(L)}\} \\
\mathsf{filter}(\mathcal{O}, V) & \overset{\text{def}}{=} \{o \in \mathcal{O} \mid \mathcal{O}(o) = (V, \_, \_, \_)\} \\
\mathsf{filter}(\mathcal{O}, L) & \overset{\text{def}}{=} \{o \in \mathcal{O} \mid \mathcal{O}(o) = (\_, L, \_, \_)\} \\
\mathsf{filter}(\mathcal{O}, V, L) & \overset{\text{def}}{=} \mathsf{filter}(\mathcal{O}, V) \cap \mathsf{filter}(\mathcal{O}, L) \\
\mathsf{from}(L) & \overset{\text{def}}{=} \begin{cases} \boxed{\downarrow} & \text{if } L = \mathsf{ctxt} \\ \boxed{\uparrow} & \text{if } L = \mathsf{lib} \end{cases} \\
\mathsf{initf}_{KX}(p.c) & \overset{\text{def}}{=} \{(p_0.c_0, f) \mapsto \mathbf{null} \mid \langle f, \_ \rangle \in_{KX} p_0.c_0 \wedge p.c \leq_{KX} p_0.c_0\} \\
\mathsf{objectrefs}(\_) & \overset{\text{def}}{=} \text{yields all object identifiers contained in the} \\
& \qquad\qquad\qquad\qquad \text{syntactic element } \_ \\
\mathsf{select}_{KX}(L) & \overset{\text{def}}{=} \begin{cases} K & \text{if } L = \mathsf{ctxt} \\ X & \text{if } L = \mathsf{lib} \end{cases} \\
\mathsf{stackabs}_L(\overline{\mathcal{F}}) & \overset{\text{def}}{=} \begin{cases} \bullet & \text{if } \overline{\mathcal{F}} = \bullet \\ \mathcal{F} \cdot \mathsf{stackabs}_L(\overline{\mathcal{F}}') & \text{if } \overline{\mathcal{F}} = \mathcal{F} \cdot \overline{\mathcal{F}}' \text{ and } \mathcal{F} = \_{}_L{:}p.t \\ \mathsf{stackabs}_L(\overline{\mathcal{F}}') & \text{if } \overline{\mathcal{F}} = \mathcal{F} \cdot \overline{\mathcal{F}}' \text{ and } \mathcal{F} = \_{}_{\neg L}{:}p.t \end{cases} \\
\mathsf{type}_{\mathcal{O}}(v) & \overset{\text{def}}{=} \begin{cases} \bot & \text{if } v = \mathbf{null} \\ p.c & \text{if } \mathcal{O}(v) = (\_, \_, p.c, \_) \end{cases} \\
\mathsf{typeabs}_{KX}(p.c) & \overset{\text{def}}{=} \langle \widehat{p.t}, \widehat{m} \rangle \text{ where} \\
& \qquad \widehat{p.t} \overset{\text{def}}{=} \{p_0.t_0 \mid p.c \leq_{KX} p_0.t_0 \wedge \mathsf{public}_X(p_0.t_0)\} \text{ and} \\
& \qquad \widehat{m} \overset{\text{def}}{=} \{m \mid \langle m, \_, p_0.c_0 \rangle \in_{KX} p.c \wedge p_0.c_0 \in \mathcal{C}_X\} \\
\mathsf{valabs}_{KX}(v, \mathcal{O}) & \overset{\text{def}}{=} \begin{cases} \mathbf{null} & \text{if } v = \mathbf{null} \\ o{:}\mathsf{typeabs}_{KX}(\mathsf{type}_{\mathcal{O}}(o)) & \text{if } v = o \end{cases}
\end{array}
$$

Figure 3.7: Semantic helper functions (alphabetically sorted)

The function $\mathsf{available}(\mathcal{O}, L)$ returns all object identifiers of objects in $\mathcal{O}$ that are either tagged as $\mathsf{exposed}$ or tagged as $\mathsf{internal}$ to L. These represent the objects that are potentially "known" to the library implementation or program context, depending on the parameter L. The function $\mathsf{objectrefs}(\dots)$ yields all object identifiers contained in a syntactic element.

Definition 3.5 (Well-formed runtime configuration)
A runtime configuration $\mathcal{S} = KX, \mathcal{O}, \overline{\mathcal{F}}$ is well-formed if

- $\mathcal{S}$ is well-typed (standard definition, not detailed further here)
- The top of the stack $\overline{\mathcal{F}}$ is an expression of the form $\mathcal{E}\lfloor E \rfloor$, the rest are evaluation contexts of the form $\mathcal{E}$
- Stack frames in $\overline{\mathcal{F}}$ are alternatively from lib and from ctxt and the lowest stack frame is from ctxt
- Stack consistency and separation (Object identifiers used in L-tagged slices of the stack are L-available in the heap): $\forall L : \mathsf{objectrefs}(\mathsf{stackabs}_L(\overline{\mathcal{F}})) \subseteq \mathsf{available}(\mathcal{O}, L)$
- Store consistency: $\mathsf{objectrefs}(\mathsf{rng}(\mathcal{O})) \subseteq \mathsf{dom}(\mathcal{O})$
- Store separation (Only L-available objects can be accessed from L-available objects): $\forall o \in \mathsf{available}(\mathcal{O}, L)$ such that $\mathcal{O}(o) = (\_, \_, p.c, \mathcal{G})$ follows that $\mathsf{rng}(\mathsf{fields}^L_{KX}(\mathcal{G})) \subseteq \mathsf{available}(\mathcal{O}, L)$
- Objects created by code of X are of a type of X: $\forall o \in \mathsf{filter}(\mathcal{O}, \mathsf{lib}) : \mathsf{type}_{\mathcal{O}}(o) \in \mathcal{C}_X \cup \{\mathsf{lang.Object}\}$
- Internal objects of K have their X fields **null**: $\forall (\mathsf{internal}, \mathsf{ctxt}, p.c, \mathcal{G}) \in \mathsf{rng}(\mathcal{O}) : \mathsf{rng}(\mathsf{fields}^{\mathsf{lib}}_{KX}(\mathcal{G})) = \{\mathbf{null}\}$. The reason for this is that changes to the X fields of an object created by K can only be made by X (as all fields are private) and hence the object must first have been exposed to X. ◇

In a similar fashion to the preservation lemma in type soundness proofs [WF94], we state in the following that runtime configurations of programs are always well-formed. Initial program states are well-formed (Lemma 3.1) and well-formedness is preserved by small operational steps (Lemma 3.2).

Lemma 3.1 (Initial state is well-formed)
The initial state $\mathcal{S}^{\mathbf{init}}_{KX}$ for a program KX is well-formed. ◇

Proof: Trivial. □

Lemma 3.2 (Preservation of well-formedness)
Consider a well-formed configuration $\mathcal{S}$. If $\mathcal{S} \overset{\gamma}{\rightsquigarrow} \mathcal{S}'$, then $\mathcal{S}'$ is well-formed as well. ◇

$$E ::= \ldots \mid \mathsf{success}$$
$$\gamma ::= \ldots \mid \mathsf{succ}$$

T-SUCCESS
$$X, p.c, \Gamma \vdash \mathsf{success} : \bot$$

R-SUCCESS
$$\frac{\overline{\mathcal{F}} = \mathcal{E}\lfloor \mathsf{success} \rfloor_{\mathsf{ctxt}}{:}p.t \cdot \overline{\mathcal{F}'} \cdot \mathcal{E}'_{\mathsf{ctxt}}{:}p'.t'}{KX, \mathcal{O}, \overline{\mathcal{F}} \overset{\mathsf{succ}}{\rightsquigarrow} KX, \mathcal{O}, \mathbf{null}_{\mathsf{ctxt}}{:}p'.t'}$$

Figure 3.8: Observability for LPJAVA

PROOF: By case distinction on operational rule used. □

In the following, we present the formal definition of backward compatibility that the results are based on in this thesis.

3.4 Observable Behavior

A standard way to compare two program parts is to use termination behavior [Mor68] or reachability of a certain state [Hen88] (e.g., a program point [Ste06; JR05a]) as the observation result. In this thesis, we use a formal model of state reachability by introducing a special expression success which we add to our surface syntax of LPJAVA in Figure 3.8. The expression has no influence on source compatibility as it is typed in the same way as **null** (T-SUCCESS). If, during a program run, such an expression (R-SUCCESS) is reached, then the execution of the running program is stopped (as we are not interested in the further execution of the program, we only want to make the observation) and the special label succ is emitted to represent the observation in the trace. We put no restrictions on how often the success expression is used in programs. However, following other works [Ste06; JR05a], we additionally assume that only the observer (i.e., program context) can produce the succ label, so that succ cannot be fabricated by the library. From this definition, we can derive the shape of traces generated by the enhanced semantics.

Lemma 3.3 (Well-formed traces)
Let K be a program context of the library implementation X. If $\overline{\gamma} \in \mathsf{traces}(KX)$, then $\overline{\gamma}$ is of the form $(\mu{\downarrow} \cdot \mu{\uparrow})^* \cdot (\mathsf{succ} \mid \mu{\downarrow})^?$. This means that traces consist of an alternating sequence of input and output labels and potentially end with the label succ, if it was not preceded by an input label (i.e., success was caused

by the program context). ◇

PROOF: Follows directly from the operational rules. □

We introduce abbreviating notations to denote whether a program run leads to an observation or not. Note that the success expression is reached in the program run if and only if the label succ occurs in the trace.

Definition 3.6 (Successful (✓) and unsuccessful (✗) programs)
We write $\mathcal{S}$✓ if there exists a configuration $\mathcal{S}'$ such that $\mathcal{S} \xrightarrow{\overline{\gamma}} \mathcal{S}'$ and $\mathsf{last}(\overline{\gamma}) =$ succ. We write $\mathcal{S}$✗ in the other cases. ◇

We use ✓ and ✗ only for programs with deterministic program contexts. We use the wording program and program run interchangeably in this case, as a program can only have a single program run modulo renaming of object identifiers.

Using the formalized notion of observation, we can then define the standard notion of contextual compatibility, namely that any program context that can make an observation with the first library implementation must be able to do the observation with the second library implementation.

Definition 3.7 (Contextual compatibility)
A library implementation Y is *contextually compatible* with a library implementation X if Y is source compatible with X and for any deterministic program context K of X: $\mathcal{S}^{\mathbf{init}}_{KX}$✓ implies $\mathcal{S}^{\mathbf{init}}_{KY}$✓. ◇

3.5 Properties

The definition of contextual compatibility quantifies over all possible program contexts and does not support inductive proofs. However, using the definitions of traces, we can compare the behavior of two library implementations at the level of the traces, providing direct support for inductive proofs.

Definition 3.8 (Contextual trace compatibility)
A library implementation Y is *contextually trace compatible* with a library implementation X if Y is source compatible with X and for any deterministic program context K of X: $\mathsf{traces}(KX) \subseteq \mathsf{abs}_X(\mathsf{traces}(KY))$. ◇

We cannot simply state trace inclusion, as Y may have more public types than X (see Definition 2.6 and $\mathsf{S2}_{X,Y}$). Our solution is to abstract from these additional types in the traces with the function $\mathsf{abs}_X(T^\alpha)$, defined in Figure 3.7, which restricts types occurring in the abstracted type T^α to public types occurring in X and restricts the methods to those that appear in these types. The resulting type is always non-empty as $\mathsf{lang.Object}$ appears in each abstracted type. We simplify the notation by lifting the function $\mathsf{abs}_X(T^\alpha)$ from abstracted types to traces, which corresponds to applying it point-wise to each abstracted type occurring in the trace.

Finally we can state that the formalized notions of backward compatibility of Definitions 3.7 and 3.8 coincide.

Theorem 2 (Contextual compatibility iff contextual trace compatibility)
Consider two library implementations X and Y. Then Y is contextually compatible with X iff Y is contextually trace compatible with X. $\diamond$

PROOF: The proof is given at the end of this section. The left-to-right direction is given by Lemma 3.8 and the right-to-left direction by Lemma 3.9. $\square$

The theorem essentially states that our definition of trace semantics is well-chosen with respect to the notion of observation that we gave in the previous section. The following lemmas reveal the core properties of the trace semantics and form the constituents needed to prove the previous theorem. All of these lemmas are proven using specialized simulation relations. The simulation relations and the proofs of these lemmas are presented in more detail in Section 5.3.

The first two lemmas show that libraries and contexts compute the next label only based on the trace history.

Lemma 3.4 (Library independence)
Consider two program contexts K_1 and K_2 for X such that $\overline{\gamma} \cdot \gamma \in \mathsf{traces}(K_1X)$ and $\overline{\gamma} \in \mathsf{traces}(K_2X)$ and $\mathsf{last}(\overline{\gamma}) = \mu\boxed{\downarrow}$. Then $\overline{\gamma} \cdot \gamma \in \mathsf{traces}(K_2X)$. $\diamond$

PROOF: Given in Section 5.3.1. $\square$

In short, the lemma states that the next label generated by the library implementation is independent of a specific program context, it only depends on the trace behavior of the program context. We give a similar lemma for source compatible library implementations.

Lemma 3.5 (Context independence)
Let Y be source compatible with X, K be a program context for X and Y, and $\overline{\gamma} \cdot \gamma \in \mathsf{traces}(KX)$ and $\overline{\gamma} \in \mathsf{abs}_X(\mathsf{traces}(KY))$ and $\overline{\gamma} = \bullet$ or $\mathsf{last}(\overline{\gamma}) = \mu\boxed{\uparrow}$. Then, $\overline{\gamma} \cdot \gamma \in \mathsf{abs}_X(\mathsf{traces}(KY))$. $\diamond$

PROOF: Given in Section 5.3.1. □

Similarly to the previous lemma, context independence states that the next label generated by a program context is independent of a specific library implementation that this context runs with, it only depends on the trace behavior of the library implementation. Both independence lemmas show that the trace does not contain too much information.

On the flip side, the following lemma provides us with the information that the trace contains all relevant information to distinguish two library implementations. It states that whenever we have two library implementations that respond in a different way when run with a deterministic program context, we can construct a deterministic program context that can observe this difference.

Lemma 3.6 (Differentiating context)
Let Y be source compatible with X and K be a deterministic program context for X and Y such that $\overline{\gamma} \cdot \gamma \in \mathsf{traces}(KX)$ and $\overline{\gamma} \in \mathsf{abs}_X(\mathsf{traces}(KY))$ but $\overline{\gamma} \cdot \gamma \notin \mathsf{abs}_X(\mathsf{traces}(KY))$. Then there is a deterministic program context K' such that $\mathcal{S}^{\mathbf{init}}_{K'X}✓$ and $\mathcal{S}^{\mathbf{init}}_{K'Y}✗$. ◇

PROOF: Given in Section 5.3.5. □

In the following, we provide the proof of Theorem 2. We show both directions. The proofs are done at the level of the traces and rely on the lemmas previously defined (but not yet proven). In order to keep the presentation slick, the proofs do not account for source compatibility. However, all the theorems and lemmas explicitly state source compatibility as requirements. We first give a small helper lemma.

Lemma 3.7 (Library causes distinctive behavior)
Let Y be source compatible with X and K be a program context for X and Y such that $\overline{\gamma} \cdot \gamma \in \mathsf{traces}(KX)$ and $\overline{\gamma} \in \mathsf{abs}_X(\mathsf{traces}(KY))$ but $\overline{\gamma} \cdot \gamma \notin \mathsf{abs}_X(\mathsf{traces}(KY))$. Then, $\mathsf{last}(\overline{\gamma}) = \mu$⍗. ◇

PROOF: We repeat the proof goal by numbering the assumptions:
ASSUME: There is a program context K and trace $\overline{\gamma} \cdot \gamma$ such that
a) $\overline{\gamma} \cdot \gamma \in \mathsf{traces}(KX)$
b) $\overline{\gamma} \in \mathsf{abs}_X(\mathsf{traces}(KY))$
c) $\overline{\gamma} \cdot \gamma \notin \mathsf{abs}_X(\mathsf{traces}(KY))$
PROVE: $\mathsf{last}(\overline{\gamma}) = \mu$⍗
The proof goes by contradiction:
ASSUME: $\mathsf{last}(\overline{\gamma}) \neq \mu$⍗
PROVE: Contradiction

We distinguish two cases:
⟨1⟩1. CASE: $\overline{\gamma} = \bullet$ or $\mathsf{last}(\overline{\gamma}) = \mu⏏$
  ⟨2⟩1. $\overline{\gamma} \cdot \gamma \in \mathsf{abs}_X(\mathsf{traces}(KY))$
    From Lemma 3.5 by assumptions (a) and (b)
  ⟨2⟩2. Q.E.D.
    Contradiction by assumption (c) and step ⟨2⟩1
⟨1⟩2. CASE: $\mathsf{last}(\overline{\gamma}) = \mathsf{succ}$
  Contradiction by Lemma 3.3 and assumption (a)
⟨1⟩3. Q.E.D.
  Cases exhaustive due to Lemma 3.3 □

Finally we prove both directions of Theorem 2.

Lemma 3.8 (Contextual compatibility implies contextual trace compatibility) Consider two library implementations X and Y. If Y is contextually compatible with X then Y is contextually trace compatible with X. ◇

PROOF: By unfolding Definitions 3.7 and 3.8:
ASSUME: For any deterministic program context K of X: $\mathcal{S}^{\mathbf{init}}_{KX}✓$ implies $\mathcal{S}^{\mathbf{init}}_{KY}✓$
PROVE: For any deterministic program context K of X, we have: $\mathsf{traces}(KX) \subseteq \mathsf{abs}_X(\mathsf{traces}(KY))$
The proof goes by contraposition:
ASSUME: There is a deterministic program context K of X such that $\mathsf{traces}(KX) \not\subseteq \mathsf{abs}_X(\mathsf{traces}(KY))$
PROVE: There is a deterministic program context K of X such that $\mathcal{S}^{\mathbf{init}}_{KX}✓$ and $\mathcal{S}^{\mathbf{init}}_{KY}✗$
This is equivalent to the following formulation, as the empty trace is in both sets and they are prefix-closed:
ASSUME: There is a deterministic program context K of X and a trace $\overline{\gamma} \cdot \gamma$ such that
  a) $\overline{\gamma} \cdot \gamma \in \mathsf{traces}(KX)$
  b) $\overline{\gamma} \in \mathsf{abs}_X(\mathsf{traces}(KY))$
  c) $\overline{\gamma} \cdot \gamma \notin \mathsf{abs}_X(\mathsf{traces}(KY))$
PROVE: There is a deterministic program context K of X such that $\mathcal{S}^{\mathbf{init}}_{KX}✓$ and $\mathcal{S}^{\mathbf{init}}_{KY}✗$
We first prove that $\overline{\gamma}$ ends in an input label, i.e., that the library implementations cause the distinctive behavior. The proof then follows directly by Lemma 3.6:
⟨1⟩1. $\mathsf{last}(\overline{\gamma}) = \mu⏏$
  From Lemma 3.7 by assumptions (a), (b) and (c)
⟨1⟩2. Q.E.D.

From Lemma 3.6 by assumptions (a), (b), (c) and step ⟨1⟩1 □

Lemma 3.9 (Contextual trace compatibility implies contextual compatibility) Consider two library implementations X and Y. If Y is contextually trace compatible with X then Y is contextually compatible with X. ◇

PROOF: By unfolding Definitions 3.7 and 3.8:
ASSUME: For any deterministic program context K of X, we have: $\mathsf{traces}(KX) \subseteq \mathsf{abs}_X(\mathsf{traces}(KY))$
PROVE: For any deterministic program context K of X: $\mathcal{S}^{\mathbf{init}}_{KX}\checkmark$ implies $\mathcal{S}^{\mathbf{init}}_{KY}\checkmark$
The claim then follows directly by Definitions 3.4 and 3.6 □

3.6 Discussion

The definition of contextual compatibility is based on the notion of successful program, which again relies on the large-step semantics and the shape of the last label of the trace being succ. At first, it might seem odd that the presented notion of observation directly relies on the definition of the traces. We can give, however, an equivalent definition of successful programs in terms of the small-step semantics and the expression $\mathsf{success}$. The advantage of the definition as stated originally allows for an easier treatment in the proofs.

In an earlier version of the semantics [WP11], we used termination behavior as the notion of observation, which lead to a more complicated presentation of the semantics and proofs. We believe that both notions of observation lead to the same results for the given language. However, if we generalize the approach to a more complex setting, for example with concurrency, only the notion of observation based on reachability is a useful one [Hen88].

Another thing to note is that Lemmas 3.4, 3.5 and 3.7 were not only stated for deterministic program contexts, but are also applicable for most general contexts (which we will consider in the next chapter). This means that these properties are preserved by the introduction of most general contexts. The proofs of the lemmas, which are deferred to a later chapter, (obviously) account for both kind of contexts.

4 Most General Context

An abstraction is one thing that represents several real things equally well.

— Edsger W. Dijkstra

The characterization of the behavior of a library with a program context in terms of traces is independent of the program configurations, i.e., heaps and stacks. Although the traces abstract from types and steps in the context, we still have to consider the traces of all possible program contexts in order to describe the full behavior of a library. In this chapter, we introduce a most general context (MGC) that enables all interactions that a standard program context can engage in. Compared to a standard program context, the most general context abstracts over types, objects and operational steps.

We recapitulate the capabilities of standard program contexts before we describe the abstraction realized by the MGC. In terms of program code, program contexts can define classes that rely on the library, either by calling code of the library or by extending classes and implementing interfaces of the library (e.g., class IntObs in Figure 1.2). In terms of operational steps, program contexts can

- create objects of classes that are defined in the program context (e.g., IntObs in line 7 of Figure 1.2) or objects of classes that are defined in the library and accessible to the program context (e.g., Subject in line 8, but not ObsIter).
- perform steps that do not lead to a change in control, e.g., access and write fields and method calls/returns that are dispatched to code of the program context (e.g., line 4 in Figure 1.2). The MGC abstracts over these steps.
- call methods that are defined in the library (e.g., line 10 in Figure 1.2) or return to code that is defined in the library (e.g., line 5 in Figure 1.2).

The only relevant actions which should be done by the MGC are those that lead to a change in control. In short, the MGC can call methods using available objects or simply return to method invocations from the library. Available objects are either those that are created by the MGC or the ones that have been exposed by the library.

The rest of this chapter is structured as follows. Section 4.1 introduces the construction of the most general context for a library implementation X that represents all standard program contexts of X. Section 4.2 presents another definition of backward compatibility in terms of the trace-based semantics and the most general context. This definition is independent of the quantification over all possible program contexts. Section 4.3 concludes.

4.1 Construction

In this section we construct a most general context $\mathsf{mgc}(X)$ based on the library implementation X that enables all possible interactions that X can engage in. The program context $\mathsf{mgc}(X)$ finitely represents exactly all program contexts that X can have. Compared to a standard program context, $\mathsf{mgc}(X)$ abstracts over types, objects, and operational steps. To express $\mathsf{mgc}(X)$, we add a non-deterministic expression nde to LPJava.

Program context representation. The most general context of X is constructed from the library implementation X which we denote by the construction function $\mathsf{mgc}(X)$. Let $p_{\mathbf{mgc}}$ be a package name not occurring in X. For each abstracted type $T^\alpha = \langle \widehat{p.t}, \widehat{m} \rangle \in \mathcal{T}_X^\alpha$, we construct a class of the following form in the package $p_{\mathbf{mgc}}$:

$\textbf{public class } c \textbf{ extends } p_0.c_0 \textbf{ implements } \overline{p.i} \; \{ \; \widehat{M^c} \; \}$

where

- c is a class name that is unique for each abstracted type T^α,
- $p_0.c_0$ is the smallest class in $\widehat{p.t}$,
- $\overline{p.i}$ are the interfaces in $\widehat{p.t}$,
- $\widehat{M^c} \overset{\text{def}}{=} \{ p_0.t_0 \; m(\overline{p_1.t_1 \; x}) \; \{ \; \mathsf{nde} \; \} \mid \langle m, p_0.t_0 \cdot \overline{p_1.t_1}, \_ \rangle \in_{KX} p.t \wedge p.t \in \widehat{p.t} \wedge m \notin \widehat{m} \}$. The body of the method (nde) is explained later.

The idea behind this construction is to create a class for each possible abstracted type T^α of the context that can occur in labels. This class has the shape such that the abstracted type of objects of the class is exactly that of the abstracted type from which the class was constructed. Formally stated, this means that $\mathsf{typeabs}_{\mathsf{mgc}(X)X}(p_{\mathbf{mgc}}.c) = T^\alpha$. For a library implementation consisting of a single class with two methods, the most general context would consist of four classes subclassing the class, overriding either none or one of the methods or both, and a class subclassing Object with no methods (if the Object class has no methods either).

The most general context $\mathsf{mgc}(X)$ also has an additional class Main, which is the startup class:

```
package main; public class Main { lang.Object main() { nde } }
```

The body of the methods that are defined in the most general context cannot be represented by LPJava, which is why we extend LPJava with the non-deterministic expression nde and introduce corresponding reduction rules (Figure 4.1). The nde expression has no influence on source compatibility as it is typed in the same way as **null** (T-Nde). We know that if Y is source compatible with X, then every program context of X is also a program context of Y. In particular, this property is preserved for most general contexts. This means that if we have $\vdash \mathsf{mgc}(X)X$ and Y is source compatible with X, then also $\vdash \mathsf{mgc}(X)Y$. In particular, running the most general context of X with a library implementation Y should simulate all possible runs of program contexts of X with Y.

Semantics of nde. The nde expression is only allowed in most general contexts and has the following semantics. Reducing a non-deterministic expression can lead to no effect at all (MGC-Skip).[1] It can lead to a successful observation (MGC-Prepare-Success). It can lead to the creation of new objects of accessible types (MGC-Prepare-New). If we have a most general context representing all program contexts for X and run it with a library implementation Y ($\rightsquigarrow^{\mathsf{ctxt}}_{\mathsf{mgc}(X)Y}$ in the consequent of the rule), then the most general context can only create objects of an accessible type that is known to X. As all classes in a most general context are public, only public classes of $\mathsf{mgc}(X)X$ are considered. Reducing a non-deterministic expression can also lead to a change in control with a well-formed method call or return using MGC-Prepare-Call or MGC-Prepare-Return. These rules rely on the extra information V and L tagged to objects to

[1]Note that MGC-Skip is only introduced to simplify the simulation proofs.

$E ::= \ldots \mid \mathsf{nde}$

T-NDE

$X, p.c, \Gamma \vdash \mathsf{nde} : \bot$

MGC-SKIP

$\mathcal{O}, \mathsf{nde} \rightsquigarrow^{\mathsf{ctxt}}_{KX} \mathcal{O}, \mathsf{nde}$

MGC-PREPARE-SUCCESS

$\mathcal{O}, \mathsf{nde} \rightsquigarrow^{\mathsf{ctxt}}_{KX} \mathcal{O}, \mathsf{success}$

MGC-PREPARE-NEW

$$\frac{p.c \in \mathcal{T}_{\mathsf{mgc}(X)X} \qquad \mathsf{public}_{\mathsf{mgc}(X)X}(p.c) \qquad x \text{ fresh}}{\mathcal{O}, \mathsf{nde} \rightsquigarrow^{\mathsf{ctxt}}_{\mathsf{mgc}(X)Y} \mathcal{O}, \mathbf{let}\ \mathsf{lang.Object}\ x = \mathbf{new}\ p.c\ \mathbf{in}\ \mathsf{nde}}$$

MGC-PREPARE-CALL

$$\frac{\begin{array}{c} o \cdot \overline{v} \subseteq \mathsf{available}(\mathcal{O}, \mathsf{ctxt}) \cup \{\mathbf{null}\} \\ \langle m, \_, p.c \rangle \in_{\mathsf{mgc}(X)Y} \mathsf{type}_{\mathcal{O}}(o) \qquad p.c \in \mathcal{C}_Y \\ \mathsf{type}_{\mathcal{O}}(o \cdot \overline{v}) \leq_{\mathsf{mgc}(X)Y} p_0.t_0 \cdot \overline{p_2.t_2} \qquad \langle m, p_1.t_1 \cdot \overline{p_2.t_2}, \_ \rangle \in_{\mathsf{mgc}(X)X} p_0.t_0 \\ x', x, \overline{x} \text{ fresh} \qquad E = \mathbf{let}\ p_0.t_0\ x = o\ \mathbf{in}\ \mathbf{let}\ p_2.t_2\ x = v\ \mathbf{in}\ x.m(\overline{x}) \end{array}}{\mathcal{O}, \mathsf{nde} \rightsquigarrow^{\mathsf{ctxt}}_{\mathsf{mgc}(X)Y} \mathcal{O}, \mathbf{let}\ \mathsf{lang.Object}\ x' = E\ \mathbf{in}\ \mathsf{nde}}$$

MGC-PREPARE-RETURN

$$\frac{v \in \mathsf{available}(\mathcal{O}, \mathsf{ctxt}) \cup \{\mathbf{null}\} \qquad \mathsf{type}_{\mathcal{O}}(v) \leq_{KX} p.t}{KX, \mathcal{O}, \mathsf{nde}_{\mathsf{ctxt}}{:}p.t \cdot \overline{\mathcal{F}} \overset{\tau}{\rightsquigarrow} KX, \mathcal{O}, v_{\mathsf{ctxt}}{:}p.t \cdot \overline{\mathcal{F}}}$$

Figure 4.1: Syntax extension, typing and rules for the most general context

determine what objects are known to the context ($\mathsf{available}(\mathcal{O}, \mathsf{ctxt})$). The function $\mathsf{available}(\mathcal{O}, L)$, defined in Figure 3.7, returns the object identifiers of all exposed objects and objects internal to L. Using the rule MGC-PREPARE-CALL, a most general context for X can create a method call $x.m(\overline{x})$ that is well-typed in $\mathsf{mgc}(X)X$, i.e., $\langle m, p_1.t_1 \cdot \overline{p_2.t_2}, \_ \rangle \in_{\mathsf{mgc}(X)X} p_0.t_0$. The values used in the method call have to be known to the context ($o \cdot \overline{v} \subseteq \mathsf{available}(\mathcal{O}, \mathsf{ctxt}) \cup \{\mathbf{null}\}$) and have the right type ($\mathsf{type}_{\mathcal{O}}(o \cdot \overline{v}) \leq_{\mathsf{mgc}(X)Y} p_0.t_0 \cdot \overline{p_2.t_2}$). Finally, the method call must be a boundary method call ($\langle m, \_, p.c \rangle \in_{\mathsf{mgc}(X)Y} \mathsf{type}_{\mathcal{O}}(o)$ and $p.c \in \mathcal{C}_Y$). The rule MGC-PREPARE-RETURN, preparing a change in control with a method return, relies on the extra type information $p.t$ tagged to the stack slices to choose a type-correct return value.

4.2 Properties

We confirm that the properties stated in the previous chapters are preserved by the introduction of the most general context and that $\mathsf{mgc}(X)X$ is a program (i.e., well-formed). We refrain from stating these properties again in this chapter. Using the definitions of traces and most general context, we can give the denotation of a library implementation without quantifying over all possible program contexts.

Definition 4.1 (Trace behavior of a library implementation)
Let Y be a library implementation that is source compatible with a library implementation X. Then the trace-based behavior of Y with all possible program contexts of X is defined as $\mathsf{abs}_X(\mathsf{traces}(\mathsf{mgc}(X)Y))$. $\diamond$

Based on this definition, we can give a more convenient formalization of backward compatibility which we call *trace compatibility*.

Definition 4.2 (Trace compatibility)
A library implementation Y is *trace compatible* with a library implementation X if Y is source compatible with X and $\mathsf{traces}(\mathsf{mgc}(X)X) \subseteq \mathsf{abs}_X(\mathsf{traces}(\mathsf{mgc}(X)Y))$. $\diamond$

In the following theorem, we state that the formalized notions of backward compatibility of Definitions 3.8 and 4.2 coincide.[2] The theorem essentially states that our definition of most general context is well-chosen with respect to the trace semantics.

Theorem 3 (Contextual trace compatibility iff trace compatibility)
Consider two library implementations X and Y. Then Y is contextually trace compatible with X iff Y is trace compatible with X. $\diamond$

PROOF: The proof is given at the end of this section. The left-to-right direction is given by Lemma 4.4 and the right-to-left direction by Lemma 4.5. $\square$

Similarly as in the previous chapter, we present in the following the lemmas that reveal the core properties of the most general context and form the constituents needed to prove the previous theorem. All of these lemmas are proven

[2]In our earlier SBMF paper [WP11], we defined trace compatibility wrongly as $\mathsf{traces}(\mathsf{mgc}(X)X) \subseteq \mathsf{abs}_X(\mathsf{traces}(\mathsf{mgc}(Y)Y))$, a property that follows from the current definition of trace compatibility, but is too weak to prove that Definitions 3.8 and 4.2 coincide.

using specialized simulation relations. The simulation relations and the proofs of these lemmas are presented in more detail in Section 5.3.

The following two lemmas state that the most general context for a library implementation X simulates exactly all possible program contexts for X.

Lemma 4.1 (MGC abstraction is sound)
Let Y be source compatible with X and K be a deterministic program context of X. Then, $\mathsf{traces}(KY) \subseteq \mathsf{traces}(\mathsf{mgc}(X)Y)$. $\diamond$

PROOF: Given in Section 5.3.3. $\square$

Lemma 4.2 (MGC abstraction is complete)
Let Y be source compatible with X and $\overline{\gamma} \in \mathsf{traces}(\mathsf{mgc}(X)Y)$. Then, there is a deterministic program context K of X with $\overline{\gamma} \in \mathsf{traces}(KY)$. $\diamond$

PROOF: Given in Section 5.3.4. $\square$

The last lemma in this section provides a property for program contexts that make only use of a subset of the API provided by a library. The lemma considers that types of Y can appear in traces of program contexts of X with the library implementation Y. However, these additional types cannot be observed by the program contexts of X, i.e., no distinctions based on these types can be made.

Lemma 4.3 (Additional types in Y not observable for program contexts of X)
Let Y be source compatible with X and K_1 and K_2 be deterministic program contexts of X such that $\overline{\gamma}_1 \in \mathsf{traces}(K_1Y)$ and $\overline{\gamma}_2 \in \mathsf{traces}(K_2Y)$ and $\mathsf{abs}_X(\overline{\gamma}_1) = \mathsf{abs}_X(\overline{\gamma}_2)$. Then $\overline{\gamma}_1 = \overline{\gamma}_2$. $\diamond$

PROOF: Given in Section 5.3.2. $\square$

In the following, we prove both directions of Theorem 3.

Lemma 4.4 (Contextual trace compatibility implies trace compatibility)
Consider two library implementations X and Y. If Y is contextually trace compatible with X then Y is trace compatible with X. $\diamond$

PROOF: By unfolding Definitions 3.8 and 4.2:
ASSUME: a) For any deterministic program context K of X, we have: $\mathsf{traces}(KX) \subseteq \mathsf{abs}_X(\mathsf{traces}(KY))$
b) $\overline{\gamma} \in \mathsf{traces}(\mathsf{mgc}(X)X)$
PROVE: $\overline{\gamma} \in \mathsf{abs}_X(\mathsf{traces}(\mathsf{mgc}(X)Y))$
The proof goes in five steps:
$\langle 1 \rangle 1.$ There is a deterministic program context K' of X such that $\overline{\gamma} \in \mathsf{traces}(K'X)$

From Lemma 4.2 by assumption (b)
$\langle 1 \rangle 2.$ $\overline{\gamma} \in \mathsf{abs}_X(\mathsf{traces}(K'Y))$
By step $\langle 1 \rangle 1$ and assumption (a)
$\langle 1 \rangle 3.$ $\overline{\gamma}' \in \mathsf{traces}(K'Y)$ where $\mathsf{abs}_X(\overline{\gamma}') = \overline{\gamma}$
By definition of $\mathsf{abs}()$
$\langle 1 \rangle 4.$ $\overline{\gamma}' \in \mathsf{traces}(\mathsf{mgc}(X)Y)$
From Lemma 4.1 by step $\langle 1 \rangle 3$
$\langle 1 \rangle 5.$ Q.E.D.
By definition of $\mathsf{abs}()$ and steps $\langle 1 \rangle 3$ and $\langle 1 \rangle 4$ □

Lemma 4.5 (Trace compatibility implies contextual trace compatibility) Consider two library implementations X and Y. If Y is trace compatible with X then Y is contextually trace compatible with X. ◇

PROOF: By unfolding Definitions 3.8 and 4.2:
ASSUME: $\mathsf{traces}(\mathsf{mgc}(X)X) \subseteq \mathsf{abs}_X(\mathsf{traces}(\mathsf{mgc}(X)Y))$
PROVE: For any deterministic program context K of X, we have: $\mathsf{traces}(KX) \subseteq \mathsf{abs}_X(\mathsf{traces}(KY))$
The proof goes by contraposition:
ASSUME: There is a deterministic program context K of X such that $\mathsf{traces}(KX) \nsubseteq \mathsf{abs}_X(\mathsf{traces}(KY))$
PROVE: $\mathsf{traces}(\mathsf{mgc}(X)X) \nsubseteq \mathsf{abs}_X(\mathsf{traces}(\mathsf{mgc}(X)Y))$
This is equivalent to the following formulation, as the empty trace is in both sets and they are prefix-closed:
ASSUME: There is a deterministic program context K of X and a trace $\overline{\gamma} \cdot \gamma$ such that
a) $\overline{\gamma} \cdot \gamma \in \mathsf{traces}(KX)$
b) $\overline{\gamma} \in \mathsf{abs}_X(\mathsf{traces}(KY))$
c) $\overline{\gamma} \cdot \gamma \notin \mathsf{abs}_X(\mathsf{traces}(KY))$
PROVE: $\overline{\gamma} \cdot \gamma \in \mathsf{traces}(\mathsf{mgc}(X)X)$ and $\overline{\gamma} \cdot \gamma \notin \mathsf{abs}_X(\mathsf{traces}(\mathsf{mgc}(X)Y))$
We prove that the trace $\overline{\gamma} \cdot \gamma$ is included in the first but not the second set of traces:
$\langle 1 \rangle 1.$ $\overline{\gamma} \cdot \gamma \in \mathsf{traces}(\mathsf{mgc}(X)X)$
From Lemma 4.1 by assumption (a)
$\langle 1 \rangle 2.$ $\overline{\gamma} \cdot \gamma \notin \mathsf{abs}_X(\mathsf{traces}(\mathsf{mgc}(X)Y))$
We first prove that $\overline{\gamma}$ ends in an input label, i.e., that the library implementations cause the distinctive behavior.
$\langle 2 \rangle 1.$ $\mathsf{last}(\overline{\gamma}) = \mu ⏏$
From Lemma 3.7 by assumptions (a), (b) and (c)
The proof then goes by contradiction:

ASSUME: $\overline{\gamma} \cdot \gamma \in \mathsf{abs}_X(\mathsf{traces}(\mathsf{mgc}(X)Y))$
PROVE: Contradiction
⟨2⟩2. There is a trace $\overline{\gamma}' \cdot \gamma'$ such that $\overline{\gamma}' \cdot \gamma' \in \mathsf{traces}(\mathsf{mgc}(X)Y)$ and $\mathsf{abs}_X(\overline{\gamma}' \cdot \gamma') = \overline{\gamma} \cdot \gamma$ and $\mathsf{last}(\overline{\gamma}') = \mu'\boxed{\downarrow}$
By definition of abs() and step ⟨2⟩1
⟨2⟩3. There is a deterministic program context K' of X such that $\overline{\gamma}' \cdot \gamma' \in \mathsf{traces}(K'Y)$ and $\mathsf{abs}_X(\overline{\gamma}' \cdot \gamma') = \overline{\gamma} \cdot \gamma$ and $\mathsf{last}(\overline{\gamma}') = \mu'\boxed{\downarrow}$
From Lemma 4.2 by step ⟨2⟩2
⟨2⟩4. There is a trace $\overline{\gamma}''$ such that $\overline{\gamma}'' \in \mathsf{traces}(KY)$ and $\mathsf{abs}_X(\overline{\gamma}'') = \overline{\gamma}$
By definition of abs() and assumption (b)
⟨2⟩5. $\overline{\gamma}' = \overline{\gamma}''$
From Lemma 4.3 by steps ⟨2⟩3 and ⟨2⟩4 and prefixed-closedness of traces
⟨2⟩6. $\overline{\gamma}' \cdot \gamma' \in \mathsf{traces}(KY)$
From Lemma 3.4 by steps ⟨2⟩3, ⟨2⟩4 and ⟨2⟩5
⟨2⟩7. $\overline{\gamma} \cdot \gamma \in \mathsf{abs}_X(\mathsf{traces}(KY))$
By steps ⟨2⟩6 and ⟨2⟩2
⟨2⟩8. Q.E.D.
Contradiction by step ⟨2⟩7 and assumption (c)
⟨1⟩3. Q.E.D.
By steps ⟨1⟩1 and ⟨1⟩2 □

4.3 Conclusion

The most general context presented in this chapter has a finite representation that is structurally very close to standard program contexts. As shown, the class table, except for method bodies, corresponds exactly to that of a standard program context.

Whereas in other works, the most general context is implicit in the semantics (e.g., [JR05b; Ábr+04]), we separate the most general context from the trace semantics and give it a program representation that is as close a possible to a normal program context. To represent the MGC, the core language only needs to be extended by a single new non-deterministic expression. The advantage of this explicit representation is that it gives a simple syntactic characterization of all possible program contexts. It also allows for a simpler definition of a more restricted set of contexts, for example by giving a smaller class table or putting additional antecedents to the operational rules.

5 Full Abstraction

All models are wrong; some models are useful.

— G. Box

In this chapter, we state the main theorem of this thesis, namely that the developed trace-based semantics with the most general context is *fully abstract*:

Theorem 4 (Full abstraction)
Consider two library implementations X and Y. Then Y is trace compatible with X iff Y is contextually compatible with X. ◊

PROOF: Immediate from Theorems 2 and 3. □

The theorem states that Definition 4.1, which describes the behavior of a library implementation in terms of the traces with the most general context, is an adequate model to study the (observable) behavior of library implementations.

All compatibility notions are preorder relations, i.e., reflexive and transitive. As consequence, the definitions and properties stated so far can directly be transferred to a setting which studies the equivalence of library implementations, as equivalence is just compatibility in both directions.

Definition 5.1 (Source, contextual and trace equivalence)
Library implementations X and Y are ψ *equivalent*, $\psi \in \{$source, contextually, trace$\}$, if X is ψ compatible with Y and Y is ψ compatible with X. ◊

The remaining part of this chapter is structured as follows. In Section 5.1, we discuss the fully abstract semantics. In Section 5.2, we present the related work. Section 5.3 finally provides the proofs that have been promised for the lemmas in the previous two chapters. Section 5.4 concludes.

5.1 Discussion

From a refinement point of view, the definition of trace compatibility might seem unusual. Typically, trace inclusion for *contextual refinement* is stated the other way around: The set of traces of Y is included in the set of traces of X. Usually, refinement considers the removal of non-determinism (i.e., removing uncertainty or underspecification) as well as adding behavior in undefined situations. As our language is deterministic up to object allocation, and the set of traces closed under renaming and thus object allocation, our setting only focuses on the second aspect. In fact, the new library implementation Y can only exhibit additional behavior if the old library implementation X fails. These include traces of runs that get either stuck at some point (e.g., **null**.f) or diverge locally by executing a infinite number of consecutive τ steps. Under these restrictions, an alternative, equivalent definition to trace compatibility can be given where the inclusion goes in the other direction. For this, the following must be observed. A set of traces T can be stated as a relation $\mathcal{R}(T) \overset{\text{def}}{=} \{(\overline{\gamma}, \gamma) \mid \overline{\gamma} \cdot \gamma \in T\}$. Trace compatibility can then be restated as $\mathcal{R}(\mathsf{traces}(\mathsf{mgc}(X)X)) \subseteq \mathcal{R}(\mathsf{abs}_X(\mathsf{traces}(\mathsf{mgc}(X)Y)))$. The relation can be totalized, as described for *data refinement* in [WD96, Chapter 16]; such a totalization is given the dotted notation $\dot{\mathcal{R}}(T)$. It can then be shown that there is an equivalent definition of trace compatibility where the inclusion goes the other way around, namely $\dot{\mathcal{R}}(\mathsf{traces}(\mathsf{mgc}(X)X)) \supseteq \dot{\mathcal{R}}(\mathsf{abs}_X(\mathsf{traces}(\mathsf{mgc}(X)Y)))$. This definition is useful to prove that an arbitrarily chosen property that holds for all behaviors of X also holds for Y as the behaviors of Y are a subset of the behaviors of X. However, for the purposes of this thesis (i.e., the formalized property *backward compatibility*), the definition of trace compatibility as stated in Definition 4.2 gives a more direct connection to the operational model of LPJAVA and the simulation-based reasoning approach that is presented in Chapter 6.

5.2 Related Work

Probably closest to our work is the fully abstract trace semantics for Java Jr., a Java subset with a package-like construct, developed by Jeffrey and

Rathke [JR05b]. The package system of Java Jr. is quite different from standard Java packages. In particular, class types are always package-local and interfaces public. Thus, code in one package cannot directly create objects in another package. Such cross-boundary object creations have to be simulated by using statically created named objects[1] and factory methods. The more severe restriction of Java Jr. is that cross-boundary inheritance is not allowed; in particular, a user of a library cannot inherit from library classes which is often done in practice, e.g., to specialize widgets of GUI frameworks. The fully abstract semantics of LPJava overcomes these restrictions and additionally supports downcasts. Thus, the LPJava semantics covers more features of existing OO programming languages. On the other hand, the restrictions of the Java Jr. package system allow for a more symmetric relationship between components and their contexts.

Steffen [Ste06] and Ábrahám et al. [Ábr+04] give a fully abstract trace semantics for a concurrent class-based language, i.e., a language without inheritance and subtyping. Similarly to Java Jr., the full abstraction proof strongly relies on the property that the traces can be decomposed into complementary traces (and recomposed) due to the duality of the component and context. An interesting technical difference between [JR05b; Ste06] and our semantics is that they use configurations in which component and context each have their own stack and heap, whereas our technique enriches normal small-step operational semantics. We developed this different approach to achieve a closer relationship to existing programming logics.

In the setting of concurrent data structures, Gotsman and Yang [GY11] use traces and a most general client to check whether a library linearizes another. Filipovic et al. [Fil+10a] use a trace abstraction to study whether sequential consistency or linearizability implies observational refinement in the setting of (abstract) object systems.

Additional related work on denotational semantics that are not formulated in terms of traces, more general refinement and simulation-based proof techniques are covered in the next chapter (Section 6.3).

[1]A particularity of Java Jr. that is not available in Java.

5.3 Simulation-based Proofs

In this section, we prove the open lemmas from the previous two chapters using specialized simulation relations. Before we can proceed, we need to extend our terminology by the notions of *minimal* renaming and *consistency* between renamings. Remember that a renaming is simply a bijective relation on object identifiers. Union, subset and composition operations on renamings have their usual mathematical interpretation for binary relations. It is important to note, however, that the union of two bijective relations does not always yield a bijective relation, as conflicting pairs of values might occur in the resulting relation.

Definition 5.2 (Minimality and consistency of renamings)
A renaming ρ is *minimal* for a property P if ρ satisfies P and no renaming ρ' exists which is a strict subset ($\rho' \subset \rho$) and which satisfies P. Two renamings are *consistent* if the union of both relations yields a renaming again, i.e., they agree on the common value pairs. ◇

Note that consistency is an equivalence relation. Composition ($\circ$) of two renamings yields a renaming again. Union ($\cup$) of two renamings that are consistent with each other yields a renaming again.

5.3.1 Context and Library Independence

We introduce the preorder relations $\preccurlyeq_{\mathsf{lib}}$ and $\preccurlyeq_{\mathsf{ctxt}}$ that relate two well-formed runtime configurations if their lib or ctxt part is the same and the other part behaves in a similar way. This allows us for example to relate runtime configurations when programs only differ either in the context or library, as in Lemmas 3.4 to 4.2, but generate the same traces. We give their definition in the following. We then show that they have simulation properties on the small step relations and extend this to large step relations.

We present the definitions of the relations $\preccurlyeq_{\mathsf{lib}}$ and $\preccurlyeq_{\mathsf{ctxt}}$ in a single definition $\preccurlyeq^{\rho}_{L}$, and rely on helper functions that were given in Figure 3.7. An informal explanation is given after the definition. Similar to the equivalence relation on terms ($\equiv^{\rho}$), the preorder relations $\preccurlyeq_{\mathsf{lib}}$ and $\preccurlyeq_{\mathsf{ctxt}}$ relate two configurations with objects relying on a non-deterministic allocator. Hence, the relations are parameterized in the renaming ρ (e.g., $\preccurlyeq^{\rho}_{\mathsf{lib}}$), which we omit when the particular ρ is not important (however, some ρ that satisfies the relation must still exist).

Definition 5.3 (Preorder relations $\preccurlyeq_L$)
Consider two well-formed configurations of the form $\mathcal{S}_1 \overset{\text{def}}{=} K_1X_1, \mathcal{O}_1, \overline{\mathcal{F}_1}$ and $\mathcal{S}_2 \overset{\text{def}}{=} K_2X_2, \mathcal{O}_2, \overline{\mathcal{F}_2}$ such that X_2 is source compatible with X_1. We write $\mathcal{S}_1 \preccurlyeq_L^{\rho_e} \mathcal{S}_2$ if ρ_e is a renaming from $\mathsf{filter}(\mathcal{O}_1, \mathsf{exposed})$ to $\mathsf{filter}(\mathcal{O}_2, \mathsf{exposed})$ and there is a renaming ρ_i from $\mathsf{filter}(\mathcal{O}_1, \mathsf{internal}, L)$ to $\mathsf{filter}(\mathcal{O}_2, \mathsf{internal}, L)$ and $\rho = \rho_e \cup \rho_i$ such that

- if $L = \mathsf{ctxt}$ then $K_1 = K_2$ else $X_1 = X_2$
- $\mathsf{exec}(\mathcal{S}_1) = \mathsf{exec}(\mathcal{S}_2)$
- $\mathsf{stackabs}_L(\overline{\mathcal{F}_1}) \equiv^\rho \mathsf{stackabs}_L(\overline{\mathcal{F}_2})$
- if $o_1 \equiv^\rho o_2$ with $\mathcal{O}_1(o_1) = (V_1, L_1, p_1.c_1, \mathcal{G}_1)$ and $\mathcal{O}_2(o_2) = (V_2, L_2, p_2.c_2, \mathcal{G}_2)$, then
  - $V_1 = V_2$ and $L_1 = L_2$
  - $\mathsf{fields}^L_{K_1X_1}(\mathcal{G}_1) \equiv^\rho \mathsf{fields}^L_{K_2X_2}(\mathcal{G}_2)$
  - if $L_1 = L$
    then $p_1.c_1 = p_2.c_2$
    else $\mathsf{typeabs}_{K_1X_1}(p_1.c_1) = \mathsf{abs}_{X_1}(\mathsf{typeabs}_{K_2X_2}(p_2.c_2))$ $\diamond$

In the following, we explain the definition. We first require that there is a renaming from the exposed objects of the first configuration to the exposed objects of the second. As the exposed objects are exactly those that have so far appeared in the traces, this means that the same amount of distinct objects have appeared in the traces for both programs. As the L parts of both configurations are the same, we also have the property that there is a renaming between the (internal) objects that are created by L. In contrast to the renaming of the internal objects, the renaming of the exposed objects is reified in the parameter of the relation ($\mathcal{S}_1 \preccurlyeq_L^{\rho_e} \mathcal{S}_2$) as we later describe how it is affected by operational steps. The definition also requires that for related configurations the execution is at the same place (either in code of the library or the context). Furthermore, we require the parts of the stack that consist of code from L to be equivalent under the object renaming. For related objects, the heap entries must also match in the following way. The exposure and location flags must be the same. The values of fields that are defined in L must be equivalent under the object renaming. At last, the dynamic type of related objects must be equal if they are created by L. Otherwise, they must have the same abstracted types.

We confirm in the following lemmas that initial states are related under $\preccurlyeq_L$.

Lemma 5.1 (Initial states are related under $\preccurlyeq_{\mathsf{lib}}$)
Consider two program contexts K_1 and K_2 and a library implementation X such that $\vdash K_1X$ and $\vdash K_2X$. Then $\mathcal{S}^{\mathbf{init}}_{K_1X} \preccurlyeq_{\mathsf{lib}} \mathcal{S}^{\mathbf{init}}_{K_2X}$. ◊

Proof: Trivial. □

Lemma 5.2 (Initial states are related under $\preccurlyeq_{\mathsf{ctxt}}$)
Consider two library implementations X_1 and X_2 such that X_2 is source compatible with X_1 and a program context K of X_1. Then $\mathcal{S}^{\mathbf{init}}_{KX_1} \preccurlyeq_{\mathsf{ctxt}} \mathcal{S}^{\mathbf{init}}_{KX_2}$. ◊

Proof: Trivial. □

We first present the connection between the relations $\preccurlyeq_L$ and steps of the small-step operational semantics and later consider large steps.

Small-step Semantics

To study the simulation properties of the relations $\preccurlyeq_L$ at the level of the small-step operational semantics, we consider three different cases. We distinguish whether the steps are initiated from L or from $\neg L$. For steps initiated from $\neg L$, we also distinguish whether the steps are labeled by τ or another label. For illustration purposes, we depict the situations graphically.

Lemma 5.3 ($\preccurlyeq_L$ simulates small step from L)
If $\mathcal{S}_1 \preccurlyeq^{\rho}_L \mathcal{S}_2$ and $\mathcal{S}_1 \overset{\gamma_1}{\rightsquigarrow} \mathcal{S}'_1$ and $\mathsf{exec}(\mathcal{S}_1) = L$, then $\mathcal{S}_2 \overset{\gamma_2}{\rightsquigarrow} \mathcal{S}'_2$ and $\gamma_1 \equiv^{\rho_\gamma} \mathsf{abs}_{X_1}(\gamma_2)$ and ρ_γ minimal and consistent with ρ and $\mathcal{S}'_1 \preccurlyeq^{\rho\cup\rho_\gamma}_L \mathcal{S}'_2$. ◊

$$\begin{tikzcd}
\mathsf{exec}(\mathcal{S}_1)=L \arrow[dr, dash] & & & & & \\
& \mathcal{S}_1 \arrow[r, dotted, dash, "\preccurlyeq_L"] \arrow[dd, squiggly, "\gamma_1"] & \mathcal{S}_2 & & \mathcal{S}_1 \arrow[r, dotted, dash, "\preccurlyeq_L"] \arrow[dd, squiggly, "\gamma_1"'] & \mathcal{S}_2 \arrow[dd, squiggly, "\gamma_2"] \\
\text{If} & & & \text{then} & {} \arrow[r, dotted, dash, "\equiv"] & {} \\
& \mathcal{S}'_1 & & & \mathcal{S}'_1 \arrow[r, dotted, dash, "\preccurlyeq_L"] & \mathcal{S}'_2
\end{tikzcd}$$

Proof: By case distinction on reduction rule used. □

Lemma 5.4 (τ steps in $\neg L$ preserve $\preccurlyeq_L$)
Assume that $\mathcal{S}_1 \preccurlyeq^{\rho}_L \mathcal{S}_2$ and $\mathsf{exec}(\mathcal{S}_1) = \neg L$. If $\mathcal{S}_1 \overset{\tau}{\rightsquigarrow} \mathcal{S}'_1$ then $\mathcal{S}'_1 \preccurlyeq^{\rho}_L \mathcal{S}_2$. Similarly, if $\mathcal{S}_2 \overset{\tau}{\rightsquigarrow} \mathcal{S}'_2$ then $\mathcal{S}_1 \preccurlyeq^{\rho}_L \mathcal{S}'_2$. ◊

$\mathsf{exec}(\mathcal{S}_1) = \neg L$

If
$$\begin{tikzcd} \mathcal{S}_1 \arrow[r, dash, dotted, "\preccurlyeq_L"] \arrow[d, squiggly, "\tau"] & \mathcal{S}_2 \\ \mathcal{S}_1' & \end{tikzcd}$$
then
$$\begin{tikzcd} \mathcal{S}_1 \arrow[r, dash, dotted, "\preccurlyeq_L"] \arrow[d, squiggly, "\tau"] & \mathcal{S}_2 \arrow[dl, dash, dotted, "\preccurlyeq_L"] \\ \mathcal{S}_1' & \end{tikzcd}$$

PROOF: By case distinction on reduction rule used. □

Lemma 5.5 (Similar labels from $\neg L$ preserve $\preccurlyeq_L$)
If $\mathcal{S}_1 \preccurlyeq_L^{\rho} \mathcal{S}_2$ and $\mathcal{S}_1 \overset{\gamma_1}{\rightsquigarrow} \mathcal{S}_1'$ and $\mathsf{exec}(\mathcal{S}_1) = \neg L$ and $\mathcal{S}_2 \overset{\gamma_2}{\rightsquigarrow} \mathcal{S}_2'$ and $\gamma_1 \equiv^{\rho_\gamma} \mathsf{abs}_{X_1}(\gamma_2) \neq \tau$ and ρ_γ minimal and consistent with ρ, then $\mathcal{S}_1' \preccurlyeq_L^{\rho\cup\rho_\gamma} \mathcal{S}_2'$. ◇

$\mathsf{exec}(\mathcal{S}_1) = \neg L$

If
$$\begin{tikzcd} \mathcal{S}_1 \arrow[r, dash, dotted, "\preccurlyeq_L"] \arrow[d, squiggly, "\gamma_1 \neq \tau"] & \mathcal{S}_2 \arrow[d, squiggly, "\gamma_2"] \\ \mathcal{S}_1' & \mathcal{S}_2' \end{tikzcd}$$
with $\equiv$ between the arrows γ_1 and γ_2;
then
$$\begin{tikzcd} \mathcal{S}_1 \arrow[r, dash, dotted, "\preccurlyeq_L"] \arrow[d, squiggly, "\gamma_1"] & \mathcal{S}_2 \arrow[d, squiggly, "\gamma_2"] \\ \mathcal{S}_1' \arrow[r, dash, dotted, "\preccurlyeq_L"] & \mathcal{S}_2' \end{tikzcd}$$
with $\equiv$ between the arrows γ_1 and γ_2.

PROOF: By case distinction on reduction rule used. □

Large-step Semantics

The three lemmas of the previous subsection can be extended to single large steps and then to many large steps, i.e., (partial) program runs.

Lemma 5.6 ($\preccurlyeq_L$ simulates single large step from L)
If $\mathcal{S}_1 \preccurlyeq_L^{\rho} \mathcal{S}_2$ and $\mathcal{S}_1 \xrightarrow{\gamma_1} \mathcal{S}_1'$ and $\mathsf{exec}(\mathcal{S}_1) = L$, then $\mathcal{S}_2 \xrightarrow{\gamma_2} \mathcal{S}_2'$ and $\gamma_1 \equiv^{\rho_\gamma} \mathsf{abs}_{X_1}(\gamma_2)$ and ρ_γ minimal and consistent with ρ and $\mathcal{S}_1' \preccurlyeq_L^{\rho\cup\rho_\gamma} \mathcal{S}_2'$. ◇

PROOF: By induction on the number of small steps and Lemma 5.3. □

Lemma 5.7 (Similar single large step from $\neg L$ preserves $\preccurlyeq_L$)
If $\mathcal{S}_1 \preccurlyeq_L^{\rho} \mathcal{S}_2$ and $\mathcal{S}_1 \xrightarrow{\gamma_1} \mathcal{S}_1'$ and $\mathsf{exec}(\mathcal{S}_1) = \neg L$ and $\mathcal{S}_2 \xrightarrow{\gamma_2} \mathcal{S}_2'$ and $\gamma_1 \equiv^{\rho_\gamma} \mathsf{abs}_{X_1}(\gamma_2)$ and ρ_γ minimal and consistent with ρ, then $\mathcal{S}_1' \preccurlyeq_L^{\rho\cup\rho_\gamma} \mathcal{S}_2'$. ◇

PROOF: By induction on the number of τ steps and Lemma 5.4 and Lemma 5.5. □

We then relate many large steps. For deterministic contexts, we can state that if we have a run starting from a state and another run starting from a related state which emits a trace equivalent to the first one, then the end states are related. We generalize this in the following corollary, where we also consider non-deterministic (i.e., most general) contexts.

Lemma 5.8 ($\preccurlyeq_L$ simulates multiple large steps)
If $\mathcal{S}_1 \preccurlyeq_L^{\rho} \mathcal{S}_2$ and $\mathcal{S}_1 \xrightarrow{\overline{\gamma}_1} \mathcal{S}_1'$ and $\mathcal{S}_2 \xrightarrow{\overline{\gamma}_2} \mathcal{S}_2'$ and $\overline{\gamma}_1 \equiv^{\rho_{\overline{\gamma}}^{12}} \mathsf{abs}_{X_1}(\overline{\gamma}_2)$ and $\rho_{\overline{\gamma}}^{12}$ minimal and consistent with ρ, then $\exists \mathcal{S}_3'$ such that $\mathcal{S}_2 \xrightarrow{\overline{\gamma}_3} \mathcal{S}_3'$ and $\overline{\gamma}_1 \equiv^{\rho_{\overline{\gamma}}^{13}} \mathsf{abs}_{X_1}(\overline{\gamma}_3)$ and $\rho_{\overline{\gamma}}^{13}$ minimal and consistent with ρ and $\mathcal{S}_1' \preccurlyeq_L^{\rho \cup \rho_{\overline{\gamma}}^{13}} \mathcal{S}_3'$. ◇

Proof: The (quite technical) proof goes by induction on the length of trace $\overline{\gamma}_1$ and uses Lemmas 5.6 and 5.7. For the induction to work, a stronger consequent relating $\mathcal{S}_2'$ and $\mathcal{S}_3'$ is needed, namely that $\mathcal{S}_2' \preccurlyeq_{\neg L}^{\rho_{\mathcal{S}_2}^{\mathbf{id}} \cup (\rho_{\overline{\gamma}}^{13} \circ \rho_{\overline{\gamma}}^{12^{-1}})} \mathcal{S}_3'$. □

The states $\mathcal{S}_1'$ and $\mathcal{S}_2'$ might not be related, as during the runs $\mathcal{S}_1 \xrightarrow{\gamma_1} \mathcal{S}_1'$ and $\mathcal{S}_2 \xrightarrow{\gamma_2} \mathcal{S}_2'$, the same most general context might have chosen different executions as it is non-deterministic. For example, it may create more objects that are internal to it in one execution than in another, but still generate an equivalent trace. For deterministic contexts, however, $\mathcal{S}_1' \preccurlyeq_L \mathcal{S}_2'$.

Proofs of Independence Lemmas

We restate Lemmas 3.4 and 3.5 and prove them using the previous lemmas.

Lemma 3.4 (Library independence)
Consider two program contexts K_1 and K_2 for X such that $\overline{\gamma} \cdot \gamma \in \mathsf{traces}(K_1 X)$ and $\overline{\gamma} \in \mathsf{traces}(K_2 X)$ and $\mathsf{last}(\overline{\gamma}) = \mu$⊕̲. Then $\overline{\gamma} \cdot \gamma \in \mathsf{traces}(K_2 X)$. ◇

Proof: The initial states of $K_1 X$ and $K_2 X$ are related by $\preccurlyeq_{\mathsf{lib}}$ due to Lemma 5.1. By Lemma 5.8, the states right after the trace $\overline{\gamma}$ are related by $\preccurlyeq_{\mathsf{lib}}$ as well. By Lemma 5.6, the library implementations then give similar outputs. □

Lemma 3.5 (Context independence)
Let Y be source compatible with X, K be a program context for X and Y, and $\overline{\gamma} \cdot \gamma \in \mathsf{traces}(KX)$ and $\overline{\gamma} \in \mathsf{abs}_X(\mathsf{traces}(KY))$ and $\overline{\gamma} = \bullet$ or $\mathsf{last}(\overline{\gamma}) = \mu$⏏. Then, $\overline{\gamma} \cdot \gamma \in \mathsf{abs}_X(\mathsf{traces}(KY))$. ◇

Proof: Follows similarly to the proof of the previous lemma by the relation $\preccurlyeq_{\mathsf{ctxt}}$, Lemmas 5.2, 5.6 and 5.8. □

5.3.2 Restricted Program Contexts

We restate Lemma 4.3 and give a proof outline.

Lemma 4.3 (Additional types in Y not observable for program contexts of X) Let Y be source compatible with X and K_1 and K_2 be deterministic program contexts of X such that $\overline{\gamma}_1 \in \mathsf{traces}(K_1Y)$ and $\overline{\gamma}_2 \in \mathsf{traces}(K_2Y)$ and $\mathsf{abs}_X(\overline{\gamma}_1) = \mathsf{abs}_X(\overline{\gamma}_2)$. Then $\overline{\gamma}_1 = \overline{\gamma}_2$. ◇

PROOF: The proof relies on the deterministic nature of the operational semantics and a simulation relation between K_1Y and K_2Y. This relation is based on $\preccurlyeq_{\mathsf{lib}}$ and additionally states that the types of corresponding context objects have the property that if $\mathsf{abs}_X(\mathsf{typeabs}_{K_1Y}(p_1.c_1)) = \mathsf{abs}_X(\mathsf{typeabs}_{K_2Y}(p_2.c_2))$, then $\mathsf{typeabs}_{K_1Y}(p_1.c_1) = \mathsf{typeabs}_{K_2Y}(p_2.c_2)$. □

5.3.3 Sound MGC Abstraction

In this section we define the simulation relation to prove Lemma 4.1. The goal is to prove that $\mathsf{traces}(KY) \subseteq \mathsf{traces}(\mathsf{mgc}(X)Y)$ for any program context K of X. We thus give a simulation relation on runtime configurations that relates the configurations of KY and $\mathsf{mgc}(X)Y$ such that whenever for related states the first configuration can make a step, the second one can make the same step (plus a few τ steps). This relation is defined as the intersection between two relations $\preccurlyeq_{\mathsf{lib}} \cap \lll$, the first of which was defined in an earlier section.

We first give a function that abstracts stack slices of deterministic program contexts to their MGC counterpart.

Definition 5.4 (MGC stack abstraction $\mathsf{nondet}(\overline{\mathcal{F}})$) Let $\mathcal{E}' \overset{\text{def}}{=} \mathbf{let}\ \mathsf{lang.Object}\ x = \lfloor\rfloor\ \mathbf{in}\ \mathsf{nde}_{\mathsf{ctxt}}{:}p.t$. Then

$$\mathsf{nondet}(\overline{\mathcal{F}}) \overset{\text{def}}{=} \begin{cases} \bullet & \text{if } \overline{\mathcal{F}} = \bullet \\ \mathsf{nde}_{\mathsf{ctxt}}{:}p.t \cdot \mathsf{nondet}(\overline{\mathcal{F}}') & \text{if } \overline{\mathcal{F}} = E_{\mathsf{ctxt}}{:}p.t \cdot \overline{\mathcal{F}}' \\ \mathcal{E}' \cdot \mathsf{nondet}(\overline{\mathcal{F}}') & \text{if } \overline{\mathcal{F}} = \mathcal{E}_{\mathsf{ctxt}}{:}p.t \cdot \overline{\mathcal{F}}' \\ \overline{\mathcal{F}} & \text{if } \overline{\mathcal{F}} = v_{\mathsf{ctxt}}{:}p.t \end{cases}$$

The topmost stack slice, which is an expression E, is abstracted to the nde expression. Lower stack slices, which are evaluation contexts $\mathcal{E}$, are abstracted to evaluation contexts of the form $\mathcal{E}'$. The evaluation context $\mathcal{E}'$ directly reflects the shape that stack slices of the most general context take when a boundary method is called (cf. MGC-PREPARE-CALL and R-CALL-BOUNDARY). Note that the choice of the identifier x can be normalized. We omit the details for simplicity. The last case considers terminal configurations (e.g., after R-SUCCESS), which are left unchanged.

We define the relation $\lll^{\rho}$, which relates the deterministic program context to the MGC, as follows:

Definition 5.5 (Preorder relation $\lll$)
Let $\mathcal{S}_1 = KY, \mathcal{O}_1, \overline{\mathcal{F}_1}$ and $\mathcal{S}_2 = \mathsf{mgc}(X)Y, \mathcal{O}_2, \overline{\mathcal{F}_2}$ be two well-formed configurations such that Y is source compatible with X and K is a deterministic program context of X. We write $\mathcal{S}_1 \lll^{\rho_e} \mathcal{S}_2$ if ρ_e is a bijective renaming from $\mathsf{filter}(\mathcal{O}_1, \mathsf{exposed})$ to $\mathsf{filter}(\mathcal{O}_2, \mathsf{exposed})$ and ρ_i is a bijective renaming from $\mathsf{filter}(\mathcal{O}_1, \mathsf{internal}, \mathsf{ctxt})$ to $\mathsf{filter}(\mathcal{O}_2, \mathsf{internal}, \mathsf{ctxt})$ and $\rho = \rho_e \cup \rho_i$ such that

- $\mathsf{nondet}(\mathsf{stackabs}_{\mathsf{ctxt}}(\overline{\mathcal{F}_1})) \equiv^{\rho} \mathsf{stackabs}_{\mathsf{ctxt}}(\overline{\mathcal{F}_2})$
- if $o_1 \equiv^{\rho} o_2$ and $\mathcal{O}_1(o_1) = (V_1, L_1, p_1.c_1, \mathcal{G}_1)$ and $\mathcal{O}_2(o_2) = (V_2, L_2, p_2.c_2, \mathcal{G}_2)$, then:
  - $V_1 = V_2$ and $L_1 = L_2$
  - $\mathsf{typeabs}_{KY}(p_1.c_1) = \mathsf{typeabs}_{\mathsf{mgc}(X)Y}(p_2.c_2)$ ◇

The main requirements of the relation $\lll$ are that the stack of the MGC corresponds to the abstracted stack of the deterministic program context and that related objects have the same abstracted type. We confirm that the relation $\preccurlyeq_{\mathsf{lib}} \cap \lll$ has the simulation property.

Lemma 5.9 (Initial states are related under $\preccurlyeq_{\mathsf{lib}} \cap \lll$)
Consider two library implementations such that Y is source compatible with X and a deterministic program context K of X. Then $\mathcal{S}_{KY}^{\mathsf{init}} \preccurlyeq_{\mathsf{lib}} \cap \lll \mathcal{S}_{\mathsf{mgc}(X)Y}^{\mathsf{init}}$. ◇

PROOF: Trivial. □

Note that sometimes two operational steps are needed by the MGC to simulate one operational step of the deterministic program context. Steps by the deterministic context using the rule R-NEW are simulated by the MGC using the rules MGC-PREPARE-NEW, R-NEW and R-LET. Steps by the context using the rule R-CALL-BOUNDARY are simulated by applying the rules MGC-PREPARE-CALL, R-LET and R-CALL-BOUNDARY. Steps by the context using the rule R-RETURN-BOUNDARY are simulated using the rules MGC-PREPARE-RETURN and R-RETURN-BOUNDARY. Finally, all other steps by the context are simulated using the rule MGC-SKIP. Steps by the library using any rule are simulated using the same rule (except R-RETURN-BOUNDARY which is simulated by R-RETURN-BOUNDARY and R-LET, as stack slices of the MGC are of the form **let** lang.Object $x = v$ **in** nde due to MGC-PREPARE-CALL).

Lemma 5.10 ($\preccurlyeq_{\mathsf{lib}} \cap \lll$ simulates small steps)
If $\mathcal{S}_1 \preccurlyeq^{\rho}_{\mathsf{lib}} \cap \lll^{\rho} \mathcal{S}_2$ and $\mathcal{S}_1 \overset{\gamma_1}{\rightsquigarrow} \mathcal{S}_1'$, then either

- $\exists i \in \mathbb{N} : \mathcal{S}_2 \overbrace{\overset{\tau}{\rightsquigarrow} \ldots \overset{\tau}{\rightsquigarrow}}^{i \text{ times}} \_ \overset{\gamma_2}{\rightsquigarrow} \mathcal{S}_2'$ or
- $\exists i \in \mathbb{N} : \mathcal{S}_2 \overset{\gamma_2}{\rightsquigarrow} \_ \overbrace{\overset{\tau}{\rightsquigarrow} \ldots \overset{\tau}{\rightsquigarrow}}^{i \text{ times}} \mathcal{S}_2'$

and $\gamma_1 \equiv^{\rho_\gamma} \gamma_2$, ρ_γ minimal and consistent with ρ and $\mathcal{S}_1' \preccurlyeq^{\rho \cup \rho_\gamma}_{\mathsf{lib}} \cap \lll^{\rho \cup \rho_\gamma} \mathcal{S}_2'$.◇

PROOF: By case distinction on reduction rule used. Lemmas 5.3 to 5.5 are also used. □

We restate Lemma 4.1:

Lemma 4.1 (MGC abstraction is sound)
Let Y be source compatible with X and K be a deterministic program context of X. Then, $\mathsf{traces}(KY) \subseteq \mathsf{traces}(\mathsf{mgc}(X)Y)$. ◇

PROOF: By using the simulation relation $\preccurlyeq_{\mathsf{lib}} \cap \lll$ between the configurations of a run of KY and the configurations of a run of $\mathsf{mgc}(X)Y$ (Lemmas 5.9 and 5.10). □

5.3.4 Complete MGC Abstraction

In this section, we give the construction of a deterministic program context to simulate a run of the most general context and define the simulation relation $\ggg$ to prove Lemma 4.2. The story goes as follows: Let Y be source compatible with X and $\overline{\gamma} \in \mathsf{traces}(\mathsf{mgc}(X)Y)$. We consider a run of $\mathsf{mgc}(X)Y$ which simulates this $\overline{\gamma}$ and show that we can construct a deterministic program context K of X which simulates this run and leads to the same trace. The simulation relation is defined as $\preccurlyeq_{\mathsf{lib}} \cap \ggg$.

Construction of Deterministic Program Context

We construct a deterministic program context based on a (partial) run of the most general context. Running the constructed program context should lead to an equivalent trace. The high level idea behind the construction is the following. The class structure for the constructed context K of X is nearly the same as

```
public class $c$ extends $p_0.c_0$ implements $\overline{p.i}$ {
  main.Main $f_{\mathsf{main}}$;
  lang.Object setMain(main.Main $x$){ this.$f_{\mathsf{main}}$ = $x$ }
  $\widehat{M^{c}}$
}
```

Figure 5.1: Classes of constructed context

for mgc(X) (except method bodies and a few extra fields and methods). The method bodies however now contain expressions that simulate the choices made by the most general context (when executing the non-deterministic expression nde). As the choices may differ for different method incarnations, the construction needs to account for this and distinguish the different incarnations. This is done by having a bookkeeping object (which every object of a type of the context refers to) that globally counts method incarnations of methods defined in the context. To enable access to all available objects (MGC-PREPARE-CALL and MGC-PREPARE-RETURN), the bookkeeping object has extra fields to store references to all objects that have been created so far by the program context or that have been exposed by the library.

We start by constructing the class structure of K which is similar to the one for the construction of the most general context in Section 4.1. For each class $p_{\mathbf{mgc}}.c$ in mgc(X) except main.Main, we construct a class such as given in Figure 5.1 with same name and header (**extends** and **implements** clause), where (1) f_{main} is a field name to refer to the initial object of class main.Main, (2) setMain is a method name not occurring anywhere in X, (3) $\widehat{M^{c}}$ is the set of methods corresponding to the ones of the class $p_{\mathbf{mgc}}.c$ in mgc(X), but with different method bodies. The method bodies are explained after the description of the class main.Main.

The class main.Main (see Figure 5.2) is the startup class and plays the bookkeeping role. It has field declarations $p_j.c_j\ f_j^{\mathsf{ctxt}}$; for every object that is created in the (partial) run by the most general context, where $p_j.c_j$ is the dynamic class type of the object. For every object of the library that is exposed in the run, we declare a field f_j^{lib} of type lang.Object. We cannot provide a more specific type to these fields, because the dynamic type of these objects might be local to a package of the library and thus not accessible in K. However, it may have public supertypes that are accessible in K. When we use the objects for method calls or returns, we cast to an appropriate super-type, however. For example, assume an object of a dynamic class type that is local to a package of the library but implementing a public interface i_1 with a method m_1 and a public interface i_2 with a method m_2. Depending on whether we want to invoke m_1 or m_2, we have to cast the object to i_1 or i_2. We provide getter and setter methods for the f_j^{ctxt} and f_j^{lib} fields.

To count method incarnations, we use objects of an additional Number class which we define in the package main. Each number is represented by a different Number object. We add a field main.Number f_j^N; for each number that refers to the corresponding Number object. The field f_{currNum} stores a reference to the object representing the number of the current method incarnation. The method incrNum sets the f_{currNum} field to the object representing the next number (it is implemented as a huge if-else cascade). We also provide getter methods for the f_j^N fields and f_{currNum}. In the main method, we initialize the Number fields.

Using the given definition of the main.Main class, we can now implement the method bodies of $\widehat{M^{\mathsf{c}}}$. They are illustrated in Figure 5.3, where we use if-else syntax for better readability. For each invocation of a method in the context, we increase the method incarnation counter. Then we choose the right code NODE_j to execute based on which method incarnation we currently are simulating. Each NODE_j will only be executed once and its construction is based on the steps done by the MGC when reducing the expression nde in the corresponding method incarnation. For each step, an expression is created, which thus leads to a sequence of expressions (delimited by ;). In the following we explain the construction of NODE_j.

First, NODE_j starts by storing the transmitted values (parameters of the method call for this method incarnation j) if necessary, i.e., if these values have not been stored already by an earlier NODE. We call these actions $\mathsf{STORE}(x)$ and they have the following form: $\mathsf{STORE}(x)$ is **null** if we do not care about the value pointed to by x, i.e., if this value was already stored, or if the value is **null**. We use the expression **this**.f_{main}.set$f_j^{\mathsf{lib}}(x)$ for library objects that have not been previously stored and the expression **this**.f_{main}.set$f_j^{\mathsf{ctxt}}((p.c)x$ **err null**) for context objects of dynamic type $p.c$ that have not been previously stored. Note that j is the index to the right field we use to store the object. Note that for each object that occurs in available($\mathcal{O}$, ctxt) in the program run, there is exactly one STORE instruction in the constructed program context K. Also note that we need to put STORE(**this**) at the beginning of NODE_0.

After the storage, we now construct for each step by the MGC an expression which simulates the step. We thus get a sequence of expressions in NODE_j. We start with the simple cases:

```
package main;

public class Main {
  main.Main $f_{\text{main}}$;
  $p_0.c_0\ f_0^{\text{ctxt}}$; ... $p_{nx}.c_{nx}\ f_{nx}^{\text{ctxt}}$;
  lang.Object $f_0^{\text{lib}}$; ... lang.Object $f_{nm}^{\text{lib}}$;
  main.Number $f_0^N$; ... main.Number $f_z^N$;
  main.Number $f_{\text{currNum}}$;

  getter−methods for all $f_j^N$ and $f_{\text{currNum}}$

  getter− and setter−methods for all $f_j^{\text{ctxt}}$ and $f_j^{\text{lib}}$

  lang.Object incrNum() { ... }

  lang.Object main() {
    this.$f_{\text{main}}$ = this;
    initialize $f_j^N$ fields with distinct Number objects
    this.$f_{\text{currNum}}$ = this.$f_0^N$;
    NODE$_0$
  }
}

public class Number {}
```

Figure 5.2: main package of constructed context

```
this.f_main.incrNum();
if (this.f_main.getf_currNum() == this.f_main.getf^N_{j_1}()) { NODE_{j_1} }
else if ... { ... }
else if (this.f_main.getf_currNum() == this.f_main.getf^N_{j_n}()) { NODE_{j_n} }
else { null } // never reached
```

Figure 5.3: Method bodies of constructed context

- For a step with MGC-PREPARE-NEW, we use the expression

$$\textbf{let } p.c\ x = \textbf{new } p.c \textbf{ in } (x.\mathsf{setMain}(\textbf{this}.f_{\mathsf{main}});\ \mathsf{STORE}(x))$$

  where x is a fresh variable name not appearing elsewhere in the construction.
- For MGC-SKIP, we use the expression $(\mathsf{main.Main})\textbf{null err null}$ (arbitrarily chosen expression which does not have any observable effect).
- For MGC-PREPARE-SUCCESS, we use the expression success.

For the cases MGC-PREPARE-CALL and MGC-PREPARE-RETURN, we need to access appropriate values in $\mathsf{available}(\mathcal{O}, \mathsf{ctxt})$. This is done by accessing the corresponding field on the bookkeeping object, which we denote by $v_{\mathsf{corr}} \overset{\mathsf{def}}{=} \textbf{this}.f_{\mathsf{main}}.\mathsf{get}f_j^{\mathsf{ctxt}}()$, $\textbf{this}.f_{\mathsf{main}}.\mathsf{get}f_j^{\mathsf{lib}}()$ or $\textbf{null}$.

- For MGC-PREPARE-RETURN, we use $(p.t)v_{\mathsf{corr}}$ **err null** where $p.t$ is the return type of the enclosing method.
- For MGC-PREPARE-CALL, we use

$$\textbf{let } \mathsf{lang.Object}\ x = E \textbf{ in } \mathsf{STORE}(x)$$

  where

$$E \overset{\mathrm{def}}{=} ((p_0.t_0)v_{\mathsf{corr}}\ \textbf{err null}).m(\overline{(p_2.t_2)v_{\mathsf{corr}}\ \textbf{err null}})$$

  and $p_0.t_0$ and $\overline{p_2.t_2}$ are the types occurring in the rule and x is a fresh variable name not appearing elsewhere.

Simulation

Using the previous construction, we can then define the simulation relation $\ggg^{\rho}$ between the most general and the constructed program context.

Definition 5.6 (Preorder relation $\ggg$)
Let $\mathcal{S}_1 = \mathsf{mgc}(X)Y, \mathcal{O}_1, \overline{\mathcal{F}_1}$ and $\mathcal{S}_2 = KY, \mathcal{O}_2, \overline{\mathcal{F}_2}$ be two well-formed configurations where Y is source compatible with X and K is a program context of X derived using the construction described in Section 5.3.4. We write $\mathcal{S}_1 \ggg^{\rho_e} \mathcal{S}_2$ if ρ_e is a bijective renaming from $\mathsf{filter}(\mathcal{O}_1, \mathsf{exposed})$ to $\mathsf{filter}(\mathcal{O}_2, \mathsf{exposed})$ and ρ_i is a bijective renaming from $\mathsf{filter}(\mathcal{O}_1, \mathsf{internal}, \mathsf{ctxt})$ to the subset of $\mathsf{filter}(\mathcal{O}_2, \mathsf{internal}, \mathsf{ctxt})$ with class type different to $\mathsf{main.Number}$ and $\rho = \rho_e \cup \rho_i$ such that

- $\mathsf{stackabs}_{\mathsf{ctxt}}(\overline{\mathcal{F}_1}) \equiv^{\rho} \mathsf{nondet}(\mathsf{stackabs}_{\mathsf{ctxt}}(\overline{\mathcal{F}_2}))$
- If $o_1 \equiv^{\rho} o_2$ and $\mathcal{O}_1(o_1) = (V_1, L_1, p_1.c_1, \mathcal{G}_1)$ and $\mathcal{O}_2(o_2) = (V_2, L_2, p_2.c_2, \mathcal{G}_2)$:
  - $V_1 = V_2$ and $L_1 = L_2$
  - $p_1.c_1 = p_2.c_2$
- $\exists o_{\mathsf{main}}$ such that $\mathcal{O}_2(o_{\mathsf{main}}) = (\mathsf{internal}, \mathsf{ctxt}, \mathsf{main.Main}, \mathcal{G})$ and
  - $\mathsf{objectrefs}(\mathcal{G}(\overline{\mathsf{main.Main}, f^{\mathsf{ctxt}}})) \equiv^{\rho} \mathsf{filter}(\mathcal{O}_1, \mathsf{ctxt})$
  - $\mathsf{objectrefs}(\mathcal{G}(\overline{\mathsf{main.Main}, f^{\mathsf{lib}}})) \equiv^{\rho} \mathsf{filter}(\mathcal{O}_1, \mathsf{exposed}, \mathsf{lib})$
  - $\mathcal{G}(\overline{f^N})$ holds the $\mathsf{main.Number}$ objects, which are distinct each. The value $\mathcal{G}(\overline{f_{\mathsf{currNum}}})$ represents the current node and holds an object from $\mathcal{G}(\overline{f^N})$.
  - $\forall o \in \mathsf{dom}(\mathcal{O}_2)$ with $\mathcal{O}_2(o) = (\_, \mathsf{ctxt}, p.c, \mathcal{G}_2)$ and $p.c \in \mathcal{C}_K$ and $p.c \neq \mathsf{main.Number}$, we have $\mathcal{G}_2(p.c, f_{\mathsf{main}}) = o_{\mathsf{main}}$. ◇

On one hand, we have a coupling that is stronger than $\lll$, as the constructed context uses exactly the same classes as the abstract context, just with other method bodies. On the other hand, the coupling is weaker as it does not relate all internal objects of ctxt (i.e., the helper objects representing numbering). To show the simulation property of $\preccurlyeq_{\mathsf{lib}} \cap \ggg$, similarly to Section 5.3.3, we first show that initial states are related and that the relation is preserved by small steps. We assume that the steps that are done by the MGC are those upon which we based the construction of the deterministic program context.

The constructed program context needs to initialize first (i.e., execute the statements that appear before NODE_0 in the main method) before it can simulate the MGC, shown in the following lemma by multiple τ steps:

Lemma 5.11 (Initial states are related under $\preccurlyeq_{\mathsf{lib}} \cap \ggg$ after a few steps)
Consider two library implementations such that Y is source compatible with X and a deterministic program context K constructed from a run of $\mathsf{mgc}(X)Y$.

Then $\exists i \in \mathbb{N} : \mathcal{S}^{\mathbf{init}}_{KY} \overbrace{\overset{\tau}{\rightsquigarrow} \dots \overset{\tau}{\rightsquigarrow}}^{i \text{ times}} \mathcal{S}$ and $\mathcal{S}^{\mathbf{init}}_{\mathsf{mgc}(X)Y} \preccurlyeq_{\mathsf{lib}} \cap \ggg \mathcal{S}$. ◇

Proof: Trivial. □

Often, many steps are needed by the constructed context to simulate a step of the MGC. A technical challenge is that there are steps by the MGC for which the relation $\lll$ cannot be established, e.g., MGC-PREPARE-CALL yields a stack slice that is not in the range of nondet(). In that case, we know that there is always a unique next step for which the relation can be established. Steps that are guaranteed to occur after each other are MGC-PREPARE-CALL and R-CALL-BOUNDARY, MGC-PREPARE-SUCCESS and R-SUCCESS, or R-RETURN-BOUNDARY and R-LET if it is a return to the MGC. In the following simulation lemma we distinguish three disjoint cases (which are exhaustive).

Lemma 5.12 ($\preccurlyeq_{\mathsf{lib}} \cap \ggg$ simulates small steps)
Consider $\mathcal{S}_1 \preccurlyeq^{\rho}_{\mathsf{lib}} \cap \ggg^{\rho} \mathcal{S}_2$.

- If $\mathcal{S}_1 \overset{\tau}{\rightsquigarrow} \mathcal{S}_1'$ is a step using the rule MGC-PREPARE-CALL or MGC-PREPARE-SUCCESS, then there are unique $\mathcal{S}_1''$, γ_1 such that $\mathcal{S}_1' \overset{\gamma_1}{\rightsquigarrow} \mathcal{S}_1''$ and $\Big(\exists i,j \in \mathbb{N} : \mathcal{S}_2 \overbrace{\overset{\tau}{\rightsquigarrow} \ldots \overset{\tau}{\rightsquigarrow}}^{i \text{ times}} \_ \overset{\gamma_2}{\rightsquigarrow} \_ \overbrace{\overset{\tau}{\rightsquigarrow} \ldots \overset{\tau}{\rightsquigarrow}}^{j \text{ times}} \mathcal{S}_2'$ and $\gamma_1 \equiv^{\rho_\gamma} \gamma_2$ and ρ_γ minimal and consistent with ρ and $\mathcal{S}_1'' \preccurlyeq^{\rho \cup \rho_\gamma}_{\mathsf{lib}} \cap \ggg^{\rho \cup \rho_\gamma} \mathcal{S}_2' \Big)$ $(\star)$.

- If $\mathcal{S}_1 \overset{\gamma_1}{\rightsquigarrow} \mathcal{S}_1'$ where γ_1 is an return output label (i.e., $\mathsf{rtrn}\ \_\boxed{\uparrow}$), then there are unique $\mathcal{S}_1''$, γ such that $\mathcal{S}_1' \overset{\gamma}{\rightsquigarrow} \mathcal{S}_1''$ and $\gamma = \tau$ and $\star$.

- If $\mathcal{S}_1 \overset{\gamma_1}{\rightsquigarrow} \mathcal{S}_1''$ and none of the previous cases apply, then $\star$. ◇

PROOF: By case distinction on reduction rule used. Lemmas 5.3 to 5.5 are also used. □

We restate Lemma 4.2:

Lemma 4.2 (MGC abstraction is complete)
Let Y be source compatible with X and $\overline{\gamma} \in \mathsf{traces}(\mathsf{mgc}(X)Y)$. Then, there is a deterministic program context K of X with $\overline{\gamma} \in \mathsf{traces}(KY)$. ◇

PROOF: By constructing a deterministic program context from a run of $\mathsf{mgc}(X)Y$ (see Section 5.3.4) and using the specialized simulation relation $\preccurlyeq_{\mathsf{lib}} \cap \ggg$ (Lemmas 5.11 and 5.12). □

5.3.5 Differentiating Context

We restate Lemma 3.6:

Lemma 3.6 (Differentiating context)
Let Y be source compatible with X and K be a deterministic program context for X and Y such that $\overline{\gamma} \cdot \gamma \in \mathsf{traces}(KX)$ and $\overline{\gamma} \in \mathsf{abs}_X(\mathsf{traces}(KY))$ but $\overline{\gamma} \cdot \gamma \notin \mathsf{abs}_X(\mathsf{traces}(KY))$. Then there is a deterministic program context K' such that $\mathcal{S}^{\textbf{init}}_{K'X}\checkmark$ and $\mathcal{S}^{\textbf{init}}_{K'Y}\text{✗}$. ◇

Proof: The proof follows directly from the following construction:
In order to distinguish two library implementations that generate different labels at some point in the trace, we construct a deterministic program context that makes the first configuration succeed and prevents the second one to do so. To construct such a context, we have to generate program code that can distinguish the situations. By Lemma 3.7, we know that the library implementations cause the distinctive behavior, i.e., that $\mathsf{last}(\overline{\gamma}) = \mu ⏏$.
By Lemma 4.1, we have $\overline{\gamma} \cdot \gamma \in \mathsf{traces}(\mathsf{mgc}(X)X)$. We then use the same construction as in Section 5.3.4. However, we also add a method lang.Object loop() { **this**.loop(); } to the main.Main class. To prevent a program to succeed, we then just call this diverging method by **this**.f_{main}.loop() in a node where we distinguish the situations.
As our constructed context is deterministic (up to object naming), the constructed program context reaches the same nodes (NODE_j) for equivalent prefixes of the trace. The construction then depends on the last label in which the two traces differ. If there is no such label for the second implementation ($\nexists \gamma' : \overline{\gamma} \cdot \gamma' \in \mathsf{traces}(KY)$), then a node is reached in the first implementation whereas the second implementation is stuck or diverges and does not reach any node NODE_j. We then put success in the previous node. If different nodes are reached (which is e.g., the case if one label is a call and the other is a return or if they are both calls but with different method names) we put in the first node success and in the second one we call **this**.f_{main}.loop(). If the same node is reached, we have to compare the abstract values that occur at the same position in the labels. Without loss of generality we assume that they differ at a certain place and that this value is referred to by variable x: if one value is **null** and the other one not, we use the expression (x == **null** ? success : **this**.f_{main}.loop()).
We assume in the following that both are abstracted objects and consider the two remaining cases:

- At least one object has occurred earlier in the trace. Then we compare the objects to the corresponding object stored in the f^{ctxt} or f^{lib} field: (x == **this**.f_{main}.getf_j^{lib}() ? success : **this**.f_{main}.loop()).

- Both objects are created by the library and have not occurred earlier in the trace: Then the abstracted types ${T^\alpha}_1$ of v_1^α and ${T^\alpha}_2$ of v_2^α have to be different.[2] Without loss of generality let $p.t$ be a public type in ${T^\alpha}_1$, but not in ${T^\alpha}_2$. Thus, we can distinguish the values by casting to this type: $(((p.t)x$ **err null**$) ==$ **null** ? success : **this**.f_{main}.loop()). □

5.4 Conclusion

This chapter has shown how particular preorder relations, that have simulation properties on the enhanced operational semantics, can be used to prove the full abstraction result. The presented trace-based semantics can be used to study different (functional) behavioral properties of library implementations. For example, it can be used to prove that a library implementation conforms to a certain specification. In the remaining part of the thesis, however, we use the model to study the presented formalized notion of backward compatibility, a property that relates two library implementations and is central to the full abstraction result.

[2]Note that for objects created by the library, their dynamic class type is always defined in the library implementation (Definition 3.5), and thus the set $\widehat{m}$ of the abstracted type $T^\alpha \overset{\text{def}}{=} \langle \widehat{p.t}, \widehat{m} \rangle$ of the object is uniquely determined by $\widehat{p.t}$.

6 Simulation-based Reasoning

All things are difficult before they are easy.

— Dr. Thomas Fuller

Building on the formal foundations of the previous sections, we outline in the following a method for reasoning about backward compatibility of libraries. The reasoning method, presented in Section 6.1, is based on the idea of directly connecting the representations of both library implementations using a so-called *coupling relation* [Hoa72; Mor94; BAW98]. As reasoning is done in terms of code, we discuss the relation between traces and program code in Section 6.2. Section 6.3 discusses the related work. Section 6.4 recapitulates the formal model in simpler terms and generalizes it to a larger subset of Java. Section 6.4 gives a short introduction to a specification language that supports the presented reasoning approach.

6.1 Reasoning about Backward Compatibility

Before describing the reasoning method, we recapitulate the proof obligations that are needed in order to prove two library implementations compatible. We also describe how the direct connection of the trace semantics to the operational semantics can be exploited to prove compatibility.

In order to prove that a library implementation Y is backward compatible with a library implementation X using the trace-based definition of compatibility, the following steps are necessary. First, Y must be proven source compatible with X. This can be directly done by the checks detailed in Section 2.3. The more difficult part is to prove, as per Definition 4.2, that $\mathsf{traces}(\mathsf{mgc}(X)X) \subseteq \mathsf{abs}_X(\mathsf{traces}(\mathsf{mgc}(X)Y))$.

In the following, we present an approach for proving backward compatibility based on the specialized simulation relations introduced earlier. Similarly to the full abstraction proof, we use specialized simulations for reasoning about backward compatibility. From Lemmas 5.2, 5.3 and 5.8 we get the property that there is a relation $\preccurlyeq^{\rho}_{\mathsf{ctxt}}$ between ctxt parts of corresponding states of the two library implementations whenever these implementations are trace compatible. The relation ensures that large steps from the context are simulated properly (Lemma 5.6). In order to prove trace inclusion, we then need to prove that large steps from related states in the library implementations are also simulated properly. For this, we need to relate also the parts of the configurations which belong to the respective library implementations. Such a relation is called a *coupling relation*. The relation can rely on the following properties of $\preccurlyeq^{\rho}_{\mathsf{ctxt}}$ (see Definition 5.3). There is a renaming ρ from the exposed objects of the first configuration to the exposed objects of the second (i.e., the objects occurring in the trace). We call this renaming a *correspondence relation*. This relation between objects of the different runtime configurations can be exploited to relate two implementations of a library, namely, we can talk about corresponding objects, which are those, that appear at the same positions in both traces.

To prove trace compatibility, a coupling relation needs to be provided. The coupling relation between the states of both library implementations can be described using the correspondence relation ρ. We then need to prove for coupled states that the next labels of small steps in the context or large steps in the library are also related and the states coupled and that the coupling holds for the initial states of the programs. We introduce the notion of *adequate* coupling relation to denote coupling relations having this simulation property.

Definition 6.1 (Adequate coupling)
$\preccurlyeq_{\mathbf{inv}}$ is an *adequate* coupling relation for two source compatible library implementations X and Y if

- $\mathcal{S}^{\mathbf{init}}_{\mathsf{mgc}(X)X} \preccurlyeq_{\mathsf{ctxt}} \cap \preccurlyeq_{\mathbf{inv}} \mathcal{S}^{\mathbf{init}}_{\mathsf{mgc}(X)Y}$, and
- If $\mathcal{S}_1 \preccurlyeq^{\rho}_{\mathsf{ctxt}} \cap \preccurlyeq^{\rho}_{\mathbf{inv}} \mathcal{S}_2$ and $\mathsf{exec}(\mathcal{S}_1) = \mathsf{ctxt}$ and $\mathcal{S}_1 \overset{\gamma_1}{\rightsquigarrow} \mathcal{S}_1'$, then $\mathcal{S}_2 \overset{\gamma_2}{\rightsquigarrow} \mathcal{S}_2'$ and $\gamma_1 \equiv^{\rho_\gamma} \mathsf{abs}_X(\gamma_2)$ and ρ_γ minimal and consistent with ρ and $\mathcal{S}_1' \preccurlyeq^{\rho\cup\rho_\gamma}_{\mathsf{ctxt}} \cap \preccurlyeq^{\rho\cup\rho_\gamma}_{\mathbf{inv}} \mathcal{S}_2'$, and
- If $\mathcal{S}_1 \preccurlyeq^{\rho}_{\mathsf{ctxt}} \cap \preccurlyeq^{\rho}_{\mathbf{inv}} \mathcal{S}_2$ and $\mathsf{exec}(\mathcal{S}_1) = \mathsf{lib}$ and $\mathcal{S}_1 \xrightarrow{\gamma_1} \mathcal{S}_1'$, then $\mathcal{S}_2 \xrightarrow{\gamma_2} \mathcal{S}_2'$ and $\gamma_1 \equiv^{\rho_\gamma} \mathsf{abs}_X(\gamma_2)$ and ρ_γ minimal and consistent with ρ and $\mathcal{S}_1' \preccurlyeq^{\rho\cup\rho_\gamma}_{\mathsf{ctxt}} \cap \preccurlyeq^{\rho\cup\rho_\gamma}_{\mathbf{inv}} \mathcal{S}_2'$. ◇

```
public class Cell { // old impl.
  private Object c;
  public void set(Object o) {
    c = o;
  }
  public Object get() {
    return c;
  }
}
```

```
public class Cell { // new impl.
  private Object c1, c2;
  private boolean f;
  public void set(Object o) {
    f = !f;
    if (f) c1 = o; else c2 = o;
  }
  public Object get() {
    if (f) return c1; else return c2;
  }
}
```

Figure 6.1: Cell example

Please note that different definitions of adequacy are possible. For example, the second condition can be defined in terms of large steps instead of small steps. A result of this are smaller coupling relations that are adequate. In general, a coupling relation only needs to talk about the library part of the configuration. If the relation only relates the lib parts, then we also have the guarantee that it is preserved by the context, which allows us to disregard steps in the (most general) context in the proof.

We can show that a coupling relation always exists if two library implementations are backward compatible. This completeness result comes basically for free from our full abstraction proof using specialized simulations.

Theorem 5 (Soundness and completeness of adequate couplings) Consider two library implementations X and Y. Then Y is trace compatible with X iff there exists an adequate coupling relation for X and Y. ◇

PROOF: Soundness follows directly from Definitions 3.4 and 4.2. Completeness follows by constructing such a relation using Lemmas 5.2, 5.3 and 5.8.

Cell example. In the following, we illustrate coupling relations using the Cell example in Figure 6.1, which provides a Cell class to store and retrieve object references. In a more refined version of the Cell library on the right of Figure 6.1, a library developer might now want the possibility to not only retrieve the last value that was stored, but also the previous value. In the new implementation of the class, he therefore introduces two fields to store values and a boolean flag to determine which of the two fields stores the last value that has been set. This second representation allows to add a method to retrieve not only the last value that was stored, but also the previous one, for example **public** T getPrevious() { **if**(f) **return** c2; **else return** c1; }.

The developer might now wonder whether the old version of the library can be safely replaced with the new version, i.e., whether the new version of the Cell library still retains the behavior of the old version when used in program contexts of the old version. The definition of a coupling relation is also called a *coupling invariant*. Intuitively, the developer might argue in the following way why he believes that both libraries are equivalent: If the boolean flag in the second library version is true, then the value that is stored in the field c1 corresponds to the value that is stored in the field c in the first library version. Similarly, if the boolean flag is false, then the value that is stored in c2 corresponds to the value that is stored in c. Using the correspondence relation ρ, we can formally talk about corresponding objects in both program runs. For all corresponding objects $(o_1, o_2) \in \rho$ that have the dynamic type Cell or a subtype thereof and where the value of the field o_2.f is true, the values that are stored in the fields o_1.c and o_2.c1 are either both null or corresponding objects, i.e., $(o_1.\mathsf{c}, o_2.\mathsf{c1}) \in \rho$. Similarly, if the value of the field o_2.f is false, then $(o_1.\mathsf{c}, o_2.\mathsf{c2}) \in \rho$ or these fields are both null. We can formalize this as follows. We write $\mathcal{O}(o, p.c, f) \overset{\text{def}}{=} \mathcal{G}(p.c, f)$ if $\mathcal{O}(o) = (\_,\_,\_,\mathcal{G})$ and then define $\preccurlyeq^{\rho}_{\mathbf{inv}}$ as $\{(\mathcal{S}_1, \mathcal{S}_2) \mid \forall (o_1, o_2) \in \rho : \mathsf{type}_{\mathcal{O}_1}(o_1) \leq_{\mathsf{mgc}(X_1)X_1} \mathsf{Cell} \Rightarrow \text{if } \mathcal{O}_1(o_1, \mathsf{Cell}, \mathsf{f})$ $\text{then } \mathcal{O}_1(o_1, \mathsf{Cell}, \mathsf{c}) \equiv^{\rho} \mathcal{O}_2(o_2, \mathsf{Cell}, \mathsf{c1}) \text{ else } \mathcal{O}_1(o_1, \mathsf{Cell}, \mathsf{c}) \equiv^{\rho} \mathcal{O}_2(o_2, \mathsf{Cell}, \mathsf{c2})\}$.

We then show that $\preccurlyeq^{\rho}_{\mathbf{inv}}$ is an adequate coupling relation. For that, we need to know the shape of labels occurring in the trace. A superset of the shapes (i.e., types and method names) of all possible input labels can be derived from the code of the library implementation, e.g., the public methods. We discuss this in more detail in Section 6.2. For this particular example, the input labels are of the form call o^{α}.get()⍗ and call o^{α}.set(v^{α})⍗. Showing that the simulation property is preserved by small steps from the context is done by case distinction over the reduction rule used (the reduction rules which are prefixed by MGC). As the coupling invariant talks only about the lib part of the configurations, this proof is trivial. The only interesting case is where related objects which have been created by the most general context with the type Cell or a subtype thereof are exposed to the library. In this case, we know that the fields of both objects are null (see Definition 3.5) and thus the coupling is preserved.

The main proof obligation is to show that the simulation property is preserved by large steps from the library. We consider the states after related inputs (e.g., call o_1^{α}.get()⍗ and call o_2^{α}.get()⍗ if $(o_1, o_2) \in \rho$) in states which are coupled (i.e., where the coupling invariant holds). We then have to prove that the states right after the next change in control are also coupled and the generated (output) labels are related. In the following, we denote the first implementation of the Cell library by X and the second one by Y. We suffix elements of the configurations of X by 1 and of Y by 2. We only consider the states right after labels of the form call $o^{\alpha}.\mathsf{get}()\text{⍗}$, i.e., we have configurations of the form

- $\mathcal{S}_1 \overset{\text{def}}{=} \mathsf{mgc}(X)X, \mathcal{O}_1, \mathsf{body}_{\mathsf{mgc}(X)X}(\mathsf{Cell}, \mathsf{get})[o_1/\mathbf{this}]_{\mathsf{lib}}{:}\mathsf{lang.Object} \cdot \overline{\mathcal{F}}_1$ and
- $\mathcal{S}_2 \overset{\text{def}}{=} \mathsf{mgc}(X)Y, \mathcal{O}_2, \mathsf{body}_{\mathsf{mgc}(X)Y}(\mathsf{Cell}, \mathsf{get})[o_2/\mathbf{this}]_{\mathsf{lib}}{:}\mathsf{lang.Object} \cdot \overline{\mathcal{F}}_2$

such that $\mathcal{S}_1 \preccurlyeq^{\rho}_{\mathsf{ctxt}} \cap \preccurlyeq^{\rho}_{\mathbf{inv}} \mathcal{S}_2$. After large steps, we then have

- $\mathcal{S}_1 \xrightarrow{\mathsf{rtrn}\ v_1^{\alpha}\text{⍐}} \mathsf{mgc}(X)X, \mathcal{O}_1, \overline{\mathcal{F}}'_1$
  where $v_1^{\alpha} = \mathsf{valabs}_{\mathsf{mgc}(X)X}(\mathcal{O}_1(o_1, \mathsf{Cell}, \mathsf{c}), \mathcal{O}_1)$, and
- $\mathcal{S}_2 \xrightarrow{\mathsf{rtrn}\ v_2^{\alpha}\text{⍐}} \mathsf{mgc}(X)Y, \mathcal{O}_2, \overline{\mathcal{F}}'_2$
  where $v_2^{\alpha} = \begin{cases} \mathsf{valabs}_{\mathsf{mgc}(X)Y}(\mathcal{O}_2(o_2, \mathsf{Cell}, \mathsf{c1}), \mathcal{O}_2) & \text{if } \mathcal{O}_2(o_2, \mathsf{Cell}, \mathsf{f}) \\ \mathsf{valabs}_{\mathsf{mgc}(X)Y}(\mathcal{O}_2(o_2, \mathsf{Cell}, \mathsf{c2}), \mathcal{O}_2) & \text{otherwise} \end{cases}$

From $\mathcal{S}_1 \preccurlyeq^{\rho}_{\mathbf{inv}} \mathcal{S}_2$, we get the property then that $\mathsf{rtrn}\ v_1^{\alpha}\text{⍐} \equiv^{\rho} \mathsf{abs}_X(\mathsf{rtrn}\ v_2^{\alpha}\text{⍐})$. As we also had $\mathcal{S}_1 \preccurlyeq^{\rho}_{\mathsf{ctxt}} \mathcal{S}_2$, the successor states are related as well by $\preccurlyeq^{\rho}_{\mathsf{ctxt}} \cap \preccurlyeq^{\rho}_{\mathbf{inv}}$.

6.2 From Traces to Program Code

An important part of the reasoning method is to establish a relation between the traces and the program code. If we have a call input label, we need to find out what the possible targets of such a call are, i.e., the method bodies in the library implementation resulting from a method dispatch that leads to such a label. Similarly, if we have a return input label, we need to find the places in the library implementation where we can return to. In general, these places cannot statically be determined, as we illustrate in the following example. Consider a public class C and a local class D as part of a library implementation where the method m in D overrides the method m in C.

```
public class C { public void m() { BODY1 } }
class D extends C { public void m() { BODY2 } }
```

Also consider a program context from which the class D is not accessible (i.e., defined in another package). If a D object is exposed under the C type, then a change in control can reach BODY2. This is the case, e.g., if there is a method of the following form in the library implementation:

```
public class Factory { public static C instance() { return new D(); } }
```

If an input label of the form call $o^{\alpha}.m()$⍗ occurs where the abstracted type of o^{α} is $\langle\{\mathsf{C}\}, \bullet\rangle$, then the program could, depending on the actual runtime type, either lead to a dispatch of the method m defined in C or in D (i.e., BODY1 or BODY2). The places which are targets of a change in control can be approximated based on the types that appear in the input/output labels. As the labels are based on the runtime type information, we can give an inverse of the abstraction function $\mathsf{typeabs}_{KX}(p.c)$ that computes the abstracted types of the labels. This inverse is a relation, i.e., it associates multiple places in the library implementation with a certain shape of message.

A more precise static analysis in the likes of [GP11] can be used to statically determine the shape of messages in most cases. Remaining (open) cases have to be formulated as part of a program invariant. For example, the invariant may specify the property that there are no exposed objects of dynamic type D. In that case, input labels could never contain D objects and thus never dispatch to a method in the class D.

6.3 Related Work

We first discuss papers where program equivalence is achieved using denotational state-transformer semantics and specific confinement notions. In the second part, we relate our approach to work on refinement and simulation-based proof techniques that are directly based on a standard operational semantics.

Denotational semantics and confinement. Classical denotational methods have been successfully used to investigate properties of object-oriented programs [Coo89]. The denotational semantics according to these methods map program parts (e.g., classes) to state transformers describing how the program part modifies the stack and heap. However, these denotations are often not fully abstract, i.e., they differentiate between classes that cannot be distinguished by a context. Banerjee and Naumann [BN05a] presented a method to reason about whole-program equivalence in a Java subset. Under a notion of confinement for class tables, they prove equivalence between different implementations of a class by relating their (classical, fixpoint-based) denotations by simulations. More recently, Naumann, Sampaio, and Silva extended this technique to multiple classes [NSS12].

With these works, we share the goal to verify that different implementations of program parts are equivalent or in a preorder relation. However, the approaches differ with respect to the semantical methods (see above) and the encapsulation techniques. Whereas we focus here on language-defined accessibility control, the previously mentioned works use ownership confinement that partitions the heap into disjoint groups of objects, so-called *islands*, dominated each by an owner object of the same type. This encapsulation boundary allows specifying the correspondence between an island of the old and new implementation in terms of a local coupling relation between islands. The proof obligation is to show that the local coupling relation has the simulation property. Confinement then guarantees that local couplings induce coupling relations on the whole program configurations. The advantage of ownership confinement is that it allows reasoning in terms of small, local couplings, having an inherent notion of modularity. A drawback is that the confinement conditions restrict the specific structures of object graphs that are supported by the method. To remedy this problem, Banerjee and Naumann use in subsequent work [BN05b] a discipline using assertions and ghost fields to specify heap encapsulation (by ownership techniques). This *inv/own* discipline, adapted from [LM04], provides an explicit representation of when object invariants and couplings are known to hold. The advantage of this approach is that the ownership structure of the heap is not fixed but can be manipulated using assertions, enabling ownership transfer between islands as well as from and to the client. In our setting, invariants must simply hold in observable states.

Ownership confinement can be understood as part of a verification technique for programs following this discipline. The goal of this thesis is a more generic semantics framework that only takes language-defined encapsulation mechanisms into account. It can be combined with ownership confinement, but it can also be used together with more flexible confinement disciplines that relax "owners as dominators" [CPN98] and allow different boundary objects to share their representation (e.g., a list with iterators).

Refinement. The formalized notion of backward compatibility can be seen as a form of data refinement [Mor94], restricted to a deterministic setting. Class refinement has been studied for various extensions of the Z specification language [WD96], however focusing on class specifications and not implementations. Exceptions to this are the works of Mikhajlova and Sekerinski [MS97] and Back, Mikhajlova, and Wright [BMW00], which consider refinement relations between a single class and its subclass, omitting more complex object structures. Filipovic et al. present a powerful theory of data refinement based on simulations in the setting of a programming model with dynamic storage and pointer arithmetic [Fil+10b]. The setting of low-level programs, which they consider, only offers weak encapsulation guarantees. Bridging the gap between higher-level and leaky lower-level abstractions motivate that refinement is not considered under all possible contexts, but only those that do not interfere with the considered module. This approach is coined by them as *blaming the client*. Leino and Yessenov develop a system for refinement of single classes with automated tool support [LY12]. Following the approach of blaming the client, contexts are restricted using a model of memory permissions. Blaming the client requires that specifications outside the programming model are available that formalize what the clients can and cannot do. In this thesis, we strived for a simple, comprehensive model that works under all possible program contexts.

Proof techniques. Bisimulations were first used by Hennessy and Milner [HM80] to reason about concurrent programs. Sumii and Pierce used bisimulations which are sound and complete with respect to contextual equivalence in a language with dynamic sealing [SP07a] and in a language with type abstraction and recursion [SP07b]. Koutavas and Wand, building on their earlier work [KW06] and the work of Sumii and Pierce, used bisimulations to reason about the equivalence of *single* classes [KW07] in different Java subsets. The subset they considered includes inheritance and down-casting but neither interfaces nor accessibility of types. Showing equivalence between two class implementations in their setting amounts to constructing adequate relations (which are sound and complete with respect to contextual equivalence). These adequate relations are similar[1] to the specialized simulation relations outlined in Section 6.1 (the auxiliary relation R^l in their work to talk about related objects corresponds to our correspondence relation ρ). As a proof method, the authors provide conditions that are sufficient for proving adequacy of a given relation. In the setting of class libraries, deriving such conditions is a lot more complicated, as it is for example not statically clear which objects or method bodies need to be related (see Section 6.2). Keeping track of the exposed objects, our simulation relations can be considered as *environmental (bi-)simulations* [SKS07]: In contrast to (standard) applicative bisimulations, environmental bisimulations have beside the tested terms an additional environment component that keeps track of additional knowledge.

[1]Whereas their proof goes by induction on the complexity of possible expressions (where complexity denotes the number of steps it takes to reduce the expression to a value), our proof goes by induction on the length of the traces of the library with the most general context.

6.4 The Formal Model Revisited

The examples in the upcoming evaluation chapter cover a larger subset of Java than the formal model. In this section, we recapitulate the formal model presented so far in simpler terms and generalize it to a larger subset of Java.

Proving backward compatibility of two library implementations relies on a particular kind of simulation relation. The library developer specifies the relation using a *coupling invariant* which describes how the old library implementation is related to the new implementation. The developer (or an automatic tool such as the developed BCVERIFIER) then proves that the relation induced by the coupling invariant has the simulation property. The relation has to hold in the states where control of execution is in code that is not part of the library, i.e., where code of the *program context* is executing. These are the states where a program (context) can *observe* if two implementations behave differently. As libraries can make internal method calls and also call back into client code using dynamic dispatch (see example in Section 7.3), the observable states are not statically bound to program points such as start and end of library methods.

The relation only equates observable states where the behavior of the two library implementations is indistinguishable. Checking that the relation induced by the coupling invariant is a proper simulation between programs with the old library implementation and programs with the new library implementation consists of ensuring that 1) the initial (observable) states are in the relation, and that 2) computational steps between consecutive observable states are properly simulated. Assuming a single-threaded setting[2], two cases can occur: 2a) Computational steps where the next state is an observable state again, i.e., control of execution stays in code of the program context. 2b) Computational steps where the next state is not an observable state. This means that control of execution goes to code of the library and returns at some later point to the program context. This is the case in which the library code, that differs in the two implementations, gets to execute.

If the old library implementation does not reach an observable state, for example by diverging or crashing, the behavior of the new library implementation is not relevant. This means that the new library implementation has the liberty to add additional behavior. To define whether or not observable states of two

[2]A generalization of the theory to a setting with concurrency is considered as future work.

library implementations are indistinguishable, it is necessary to know what part of the state results from code of the library implementation and which part results from the program context.

6.4.1 Characterization of Library State

In the single-threaded setting, a program state usually consists of a single stack and a heap. A *stack* is a sequence of stack frames. Stack frames are created by method invocations. If the body of the invoked method is defined in the library, then we say that the stack frame belongs to the library; otherwise it is part of the program context. We group *consecutive* stack frames that belong either all to the library or the program context into *stack slices*. A well-formed stack then consists of an alternating sequence of stack slices that belong to the library and stack slices that belong to the program context, i.e., the stack corresponds to a zipper with alternating teeth. The stack slice at the bottom of the stack belongs to the program context as execution starts in the program context, usually with a main method.

Separating the *heap* into a part that belongs to the library and a part that belongs to the program context is a bit more difficult. With inheritance, some code parts of an object can belong to the context and other parts to the library. We differentiate for fields whether they have been defined in classes of the program context or the library. For simplicity, we assume that code outside the library does not directly access fields that are defined in the library, which is usually considered bad practice anyhow. For the libraries to be indistinguishable, the heap state reachable from stack slices of the program context must be similar. To better characterize the objects which are potentially reachable by the program context, we distinguish (1) which objects have been *created by code of the library* or by code of the program context, and (2) which objects created by the program context have been *exposed* (i.e., made known) to the library or vice versa.

As all possible program contexts have to be considered, we assume that every object which has been made known to the program context at some point in time can later be used again by the program context. The objects which can appear in stack slices of the program context are then only objects which have been created by the program context or those which have been created by the library and which have been exposed.

6.4.2 Indistinguishable States

In the following, we call program states for programs with the old library implementation *old program states* and the program states for programs with the new library implementation *new program states*. The simulation relation equates program states which have the same number of stack slices and where the stack slices of the program context are similar. This allows stack slices of the library implementations to be completely different. Due to the non-deterministic choice of object identifiers (i.e., heap locations) during object allocation, the stack slices of the program context can only be identical modulo a renaming between objects identifiers appearing in the old and object identifiers appearing in the new state. The renaming tracks which new objects take the place of the old objects and must be a bijective relation in order to guarantee indistinguishability, as otherwise an identity check from the program context using the == operator would yield true for one library implementation and false for the other. The simulation relation thus equates program states for which there is a renaming between the exposed objects of the old program state and the exposed objects of the new program state. We call this the *correspondence relation* and talk about corresponding objects, modeled in the formal model with the equivalence $\equiv^{\rho}$.

The two library implementations might however still create *different* objects as long as these are not exposed to the program context. Corresponding objects can have different dynamic types but must have the same public super types that are defined in the old library implementation as the context can only use the public types to distinguish them. As we assume that code outside of the library does not directly access fields that are defined in the library, we can abstract from the fields of classes that are defined in the library. Similarly to the correspondence relation, there must be a renaming between the internal objects created by the (same) program context of the old and new program state. Here the runtime types of the objects are exactly the same, as these objects have been created by the same expression in the program context (e.g., `new C()`).

6.4.3 Proof Obligations

As most important step to prove backward compatibility, we need to show that computational steps between observable states are properly simulated. If control of execution goes from code of the program context to library code, this can only be due to a method call, method return or a constructor call.

We thus have to consider calls of all available (public or protected) methods and constructors. Similarly, we have to consider all possible return points in code of the library where a method was called that could potentially lead to code of the program context to be executed. We assume for the pre-states that they were related, which means that they are indistinguishable and satisfy the coupling invariant. As the observable pre-states were indistinguishable and we had a method call, this means that the receiver/parameters of the call were corresponding and a method with the same name and source compatible signature was called. Similarly, if we had a method return, then the return values were corresponding. For the post-states (if they exist), we must prove that they are related again. This means that we need to prove that the coupling invariant still holds for the post-states. In order to satisfy indistinguishability, we need to check again in case of a method call whether a method with the same name and similar signature was called and the receiver/parameters are indistinguishable, or for a method return whether the return values are indistinguishable.

Constructor calls from the program context present a slightly more complicated situation. The receiver object of such a constructor call, at first internal to the context, is exposed as soon as the first library constructor in the class hierarchy executes.[3] As library implementations are supposed to be definition-complete (i.e., contain all dependencies and can be typechecked/compiled in isolation), we can safely assume that all parameters that are passed to the constructor are exposed. We can safely deal with callbacks that occur during construction, because as soon as the first library constructor is invoked, the object is marked as exposed. This means that the object becomes part of the observable state as soon as the control flow returns to code of the program context, independently whether it is caused by a method call to code of the program context that appears within the body of the constructor or by termination of the constructor.

6.5 Specification Language

To facilitate the specification of coupling invariants, a specification language called ISL (*Invariant Specification Language*) is introduced. Specifications in

[3]Here, we additionally consider constructors with expression bodies and not only default constructors as in the formalized model.

ISL describe properties about the two program configurations as well as their relation. To this means, ISL provides facilities to access the (abstracted) heap and stack of the enhanced runtime configurations. For example, the coupling invariant that was mathematically stated in Section 6.1 for the Cell example of Figure 6.1 can be stated in ISL in the following way.

```
invariant forall old Cell o1, new Cell o2 ::
  o1 ∼ o2 ⇒ if o2.f then o1.c ∼ o2.c1 else o1.c ∼ o2.c2;
```

The coupling invariant, which has to hold in properly simulated observable program states, states that the field values must correspond between objects o2 of the new type Cell (or subtype thereof) and the Cell objects o1 of the old library implementation for which they act as a substitute. The specification language uses the operator ∼ to denote the correspondence between old and new reference values (i.e., $\equiv^{\rho}$ in the mathematical description). The **forall** quantification in ISL only ranges over objects that are not internal to the program context, i.e., objects which are exposed or created by the library. The reason for this is that objects which are internal to the program context are not relevant for the behavior of the library. We then need to prove the simulation property, which amounts to three proof cases:

- The initial state must satisfy the coupling invariant, which is trivial to prove as no Cell objects are allocated in this state.
- The coupling must be preserved by steps in the program context. As the invariant only talks about the library state of exposed Cell objects (due to o1 ∼ o2), which remains untouched in steps of the program context, the proof is trivial.
- The coupling must be preserved by interactions with the library. Program contexts can either call the (empty) Cell constructor or the methods get and set and the libraries react by returning from the constructor or method by returning a value or not (void). The property can be established in the same way as was done in Section 6.1.

Specifications in ISL are typed, which allows to catch errors in specifications with a type checker. For example, expressions have to be properly typed. The conditional expression **if** … **then** … **else** … only makes sense if the **if** clause is an expression of boolean type. In a similar way, field access on a certain variable is only reasonable if a field with that name is defined for values stored by this variable (e.g., o1.c). More semantic well-formedness conditions (e.g., disallowing field access on a **null** value), that are highly context-dependent, are not covered by the type-based discipline and instead turned into additional proof obligations. The different features of ISL are introduced on a per-need basis in the following chapter.

7 Experience and Evaluation

Even rats learn from experience.

— George Skarbek

This chapter validates the presented theory using a number of classical examples from the literature. Each section addresses typical challenges that arise in the setting of proving backward compatibility of object-oriented libraries and illustrates them with examples. Section 7.1 discusses type abstraction, Section 7.2 information hiding, Section 7.3 callbacks and Section 7.4 more elaborate forms of control flow. Hand in hand with the various challenges, we provide features of the ISL specification language for dealing with the different forms of data and control flow abstraction that appear in libraries. A more exhaustive overview of ISL is deferred until the next chapter.

7.1 Type Abstraction

A typical feature of object-oriented programming are interfaces, which allow library developers to hide implementation details from clients. In particular, it allows the library developer to provide different class implementations that implement the same (Java) interface. In the setting of library evolution, this gives the library developer the additional choice to replace implementations. As the choice of implementation can be based on input values provided by the program context, a static verifier has to account for all possible replacements. To reduce all possible replacements to the ones that can really occur, valid replacements can be specified in the coupling invariant, thereby ruling out illegal combinations as they become part of the proof assumptions and obligations.

```
public interface Fruit {} // old and new impl.
public class Apple implements Fruit {}
public class Banana implements Fruit {}
```

```
public class Factory { // old impl.
  public static Fruit
    giveMeAFruit(boolean b) {
    if (b)
      return new Apple();
    else
      return new Banana();
  }
}
```

```
public class Factory { // new impl.
  public static Fruit
    giveMeAFruit(boolean b) {
    if (b)
      return new Apple();
    else
      return new BabyBanana();
  }
}
```

```
public class BabyBanana
  extends Banana {} // new impl.
```

Figure 7.1: Subtypes example

As an example, consider the library implementation in Figure 7.1 that has a public interface Fruit with implementing classes Apple and Banana. In the new implementation, the library developer decides to deliver objects of a new subclass BabyBanana instead of the class Banana. This can be implemented for example using the factory method pattern [Gam+95] which describes how to create objects by leaving the choice which class to instantiate to the library implementor. The knowledge that apples are replaced by apples and bananas are substituted by either bananas or baby bananas can be specified in the coupling invariant, which is then checked to hold in all observable states:

```
invariant forall old Fruit o1, new Fruit o2 :: o1 ∼ o2 ⇒
  (o1 instanceof old Apple ⇔ o2 instanceof new Apple);
```

Depending on which implementation types are public or not, some combinations are acceptable whereas others are not. If Apple and Banana are public classes in the old library implementation, Banana can never be returned by the new library implementation instead of an Apple. Otherwise, a dynamic type check from a program context, for example using the **instanceof** operator, could distinguish them (see Section 6.4.2). To take these properties into account, a formal model of the type system and the properties of source compatibility is needed. The verifier, developed as part of this thesis, encodes many such properties. For example, calls of methods with same name but on objects of unrelated types (see the example in Section 7.2) should not be considered as valid simulation steps.

7.2 Information Hiding

Information hiding is an essential principle for modular development. The following two examples illustrate coupling invariants that specify properties about the internal representation of the library implementations.

Name generation example. We present in Figure 7.2 a Java adaptation of an example given by Ahmed, Dreyer, and Rossberg [ADR09]. Both library implementations (**old** and **new**) provide a Factory class that has two methods; a method fresh to generate fresh names, and a method check to test whether two names are equal. The old library implementation uses an integer field in the returned instance to represent the identity (where we ignore integer overflows) and the new library implementation uses directly the object identity. This relation between the two library implementations, which guarantees that both implementations of the check method return the same result, is captured by the coupling invariant in the following way:

```
invariant forall old C c1, old C c2, new C c3, new C c4 ::
  c1 ~ c3 ∧ c2 ~ c4 ⇒ (c1.z == c2.z ⇔ c3 == c4);
```

In order to establish this invariant as well for calls to the fresh method, an additional property of the old library implementation is needed, namely that the used integer value ++x is indeed fresh:

```
invariant forall old C c :: c.z <= old Factory.x;
```

Such a property about a single (library) implementation is usually called a *representation invariant* and can be modeled in ISL in the same way as coupling invariants. A more elaborate example follows.

OBool example. Libraries are usually composed of many classes and form their behavior with multiple cooperating objects. Consider the example given in Figure 7.3, adapted from Banerjee and Naumann [BN05a]. The example library consists of a Bool class that represents mutable boolean objects and a class OBool that realizes the same behavior by wrapping the previous class Bool.

```
package namegen; // old impl.
public class C {
  int z;
  C(int z) { this.z = z; }
}
```

```
package namegen; // old impl.
public class Factory {
  private static int x;
  public static C fresh() {
    return new C(++x);
  }
  public static boolean
    check(C o1, C o2) {
    if (o1 != null ∧ o2 != null)
      return o1.z == o2.z;
    return false;
  }
}
```

```
package namegen; // new impl.
public class C {}
```

```
package namegen; // new impl.
public class Factory {
  public static C fresh() {
    return new C();
  }
  public static boolean
    check(C o1, C o2) {
    if (o1 != null ∧ o2 != null)
      return o1 == o2;
    return false;
  }
}
```

Figure 7.2: Name generation example

```
public class Bool { // old and new impl.
  private boolean f;
  public void set(boolean b) { f = b; }
  public boolean get() { return f; }
}
```

```
public class OBool { // old impl.
  private Bool g = new Bool();
  public OBool(){ g.set(false); }
  public void set(boolean b){
    g.set(b);
  }
  public boolean get(){
    return g.get();
  }
}
```

```
public class OBool { // new impl.
  private Bool g = new Bool();
  public OBool(){ g.set(true); }
  public void set(boolean b){
    g.set(!b);
  }
  public boolean get(){
    return !g.get();
  }
}
```

Figure 7.3: OBool example

The class Bool is identical in both library implementations. The class OBool is implemented in the new implementation by storing the complement in the wrapped Bool instance. The example illustrates that objects of the same class Bool can appear in different roles, either as exposed or internal objects.

The coupling invariant then specifies that for corresponding Bool objects (which implies that they are exposed), the boolean value that is stored in field f is the same. For corresponding OBool objects, the boolean values that are stored in their referenced (non-exposed) Bool instances are complements. In the ISL specification language, this looks as follows:

```
invariant forall old Bool o1, new Bool o2 :: o1 ~ o2 ⇒ o1.f == o2.f;
invariant forall old OBool o1, new OBool o2 :: o1 ~ o2 ⇒ o1.g.f == !o2.g.f;
```

The coupling invariant consists of the logical conjunction of all specified invariants. A requirement we have is that the specifications must be well-formed. In particular, the verifier does not accept specifications where **null** could potentially be dereferenced, e.g., o1.g.f. As we know that for exposed OBool objects the g field always refers to Bool objects, we put this knowledge into the following invariant

```
invariant forall old OBool o :: o.g != null; // same for new OBool
```

and write the same for **new** OBool objects. This condition has to be put before the previous invariants as preceding invariants are used to prove well-formedness of succeeding invariants. A more detailed overview of well-formedness is given in Section 8.1.

Even though the specification is now well-formed, the given coupling invariant is not strong enough to prove backward compatibility. Two more issues need to be solved. First, a verifier cannot know whether the method calls g.set(...) and g.get() in lines 3, 5 and 8 may lead to execution of code in the program context as g might point to an object of a subclass of Bool that is defined in the program context with overriding methods. The reason for this is that, when considering the execution paths through the methods, the verifier is unaware of the statement g = **new** Bool() in line 2. The knowledge, that no method overridden by the context is called, has to be put as part of a representation invariant.[1] A simple way to assert that the methods are not overwritten by the program context is to specify that the object referred to by g has been created by the library. As library implementations are supposed to be definition-complete (i.e., contain all dependencies and can be typechecked/compiled in isolation)

[1]An example where calling methods will lead to execution of code in the program context is given in the next subsection.

Bool is the only possible implementation type. The ISL language provides a number of built-in predicates that reify the reasoning concepts of Section 6.4.1. The predicate *createdByLibrary* determines whether an object has been created by code of the library or by code of the program context and returns **true** in the first case, and **false** in the second one:

```
invariant forall old OBool o :: createdByLibrary(o.g); // same for new OBool
```

The second reason why the coupling invariant is not strong enough is directly related to a classical problem of object-orientation, namely aliasing. We show how to address it for the given example but believe this problem to be largely orthogonal to the issues addressed in this thesis. In particular, we do not prescribe any specific aliasing discipline. We first state using the built-in ISL predicate *exposed* that the objects referenced by the g field are not exposed. This ensures that the first invariant which was presented at the beginning of this subsection does not apply to these objects.

```
invariant forall old OBool o :: !exposed(o.g); // same for new OBool
```

Next, we describe the exact shape of the compound OBool object structures. A simple way to describe the structure is to assert that there is no aliasing between different g fields. We specify the invariant

```
invariant forall old OBool o1, old OBool o2 :: o1 != o2 ⇒ o1.g != o2.g; //Alt. 1
```

and write the same for **new** OBool objects. Note that a typical ownership discipline [CPN98] could be used to achieve the same effect.

Interestingly, a weaker form of aliasing is sufficient to prove equivalence of the previous implementations, namely that the Bool objects referenced by the g field are consistently aliased, meaning that the Bool objects referenced by g coincide for two arbitrary pairs of corresponding OBool objects:

```
invariant forall old OBool o1, old OBool o2, new OBool o3, new OBool o4 ::
  o1 ∼ o3 ∧ o2 ∼ o4 ⇒ (o1.g == o2.g ⇔ o3.g == o4.g); //Alt. 2
```

To illustrate the differences between the previous two variants of specifications, let us consider a variant of the example where both implementations of OBool have an additional method which returns a shallow clone of the current OBool object by sharing the same inner Bool instance:

```
public OBool clone() { OBool cl = new OBool(); cl.g = g; return cl; }
```

This is an example where we have multiple exposed objects sharing a common representation, a difficult scenario for ownership disciplines. The first invariant (Alt. 1, No aliasing) is violated by this method, but equivalence can still be established using the second invariant (Alt. 2). The key observation we made from examples such as the previous one is that stating the exact shape of the object structures is not always needed to prove backward compatibility. Often it is sufficient to find some kind of isomorphism between the object structures. For example, the previous aliasing property can be reformulated as a graph homomorphism of the field relation g from the graph of the bijection ∼ to the graph of some bijection bij:

```
invariant exists binrelation bij :: bijective(bij) ∧
  forall old OBool o1, new OBool o2 :: o1 ∼ o2 ⇒ related(bij, o1.g, o2.g);
```

ISL supports custom binary relations (beside the special built-in bijection ∼) as we found them to be particularly useful for the setting of equivalence checking. The ISL specification language provides the built-in type **binrelation** to denote binary relations on reference values and the built-in predicates *related* and *bijective* to check whether two reference values are in a relation and to check whether a relation is bijective. Unfortunately, the automatic verifier BCVerifier fails to verify the OBool constructors using the given invariant. The reason for this is that the underlying SMT solver is not smart enough to find an instantiation for bij that satisfies the given conditions. Ideally, the bijection bij in the poststates should be similar as for the one in the prestates where the newly created Bool objects are added. To assist the prover, the library developer needs to describe how the bijection changes over time. Similar as in specification languages for single programs [Cha+05], we introduce ghost variables in ISL to enable the definition and manipulation of auxiliary state.

Extension of Program State. A *ghost variable* is an updatable variable that does not appear in the program. ISL provides facilities to declare and assign values to ghost variables. The variables can then be referred to in the coupling invariant. To preserve the behavior of the library, ghost variables can only be assigned values from the program code, but not influence variables of the implementation. To verify the previous example, we declare three ghost variables bij, x1 and x2 in a global scope. The variable bij represents the previously discussed bijection between internal Bool objects. The variables x1 and x2 are used to refer to the Bool objects that are added to the relation bij. Initial values for the ghost variables need to be specified, as the coupling invariant is checked for the initial states of the programs. In observable states, the relation bij must be bijective and the ghost variables x1 and x2 contain the **null** value:

```
var binrelation bij = empty();
var old Bool x1 = null; var new Bool x2 = null;
invariant bijective(bij) ∧ x1 == null ∧ x2 == null;
invariant forall old OBool o1, new OBool o2 ::
  o1 ∼ o2 ⇒ related(bij, o1.g, o2.g);
```

The ISL function *empty* yields the empty relation. To update the relation bij, the following steps are taken. The newly created Bool objects are assigned to the ghost variables x1 and x2. This is done at the beginning of the respective OBool constructor. When control flow exits the library, i.e., right before the next observable states, the relation bij is updated with the values of x1 and x2 if both values are non-null, which means that the constructor has been executed. Finally null is assigned to both x1 and x2 to preserve the invariant. In ISL, the given steps can be specified as follows:

```
local place p1 = line 3 of old OBool assign x1 = this.g nosync;
local place p2 = line 3 of new OBool assign x2 = this.g nosync;
assign bij = if x1 != null ∧ x2 != null then add(bij, x1, x2) else bij;
assign x1 = null;
assign x2 = null;
```

Local places and **nosync** are explained in more detail in Sections 7.4 and 7.5. In this case, they are solely used to assign a value to a ghost variable at a specific program point.[2] All expressions in ISL are pure, meaning that their evaluation is free of side effects. Following this principle, the ISL function *add* yields the relation where the given values are added.

7.3 Callbacks

Callbacks are ubiquitous in object-oriented programs which makes reasoning very hard. The following adapted example [MS88; BN05a], given in Figure 7.4, presents an interface C with a method run that can be implemented by clients of the library. The class A has a method exec that invokes the run method on the passed parameter and returns a boolean that denotes whether the value stored in the g field is even. The class also has a method inc which increments the field g by two.

In a new implementation of A, the library developer now optimizes the body of the exec method and replaces `return (g + i) % 2 == 0;` by `return true;`. As there is no implementation of the C interface in the library implementations,

[2]In JML [Bur+03], the assignments to ghost variables are stated as comments in the code. As we wanted to leave the implementations untouched and for tooling reasons, we opt to state the assignments as part of the specification.

```
public interface C { // old and new impl.
  public void run();
}
```

```
public class A { // old impl.
  private int g = 0;
  public boolean exec(C c) {
    int i = 4;
    if (c != null) c.run();
    return (g + i) % 2 == 0;
  }
  public void inc() { g = g + 2; }
}
```

```
public class A { // new impl.
  private int g = 0;
  public boolean exec(C c) {
    int i = 4;
    if (c != null) c.run();
    return true;
  }
  public void inc() { g = g + 2; }
}
```

Figure 7.4: Callback example

the verifier can prove for both library implementations that the call c.run() will lead to execution of code in the program context. The crucial part is that a program context can now call back into the library, for example the method inc on the same object. This means that the coupling invariant needs to be established before the call of run and can be assumed to hold after the call. The specification necessary to verify the given example states that the value stored in the field g is even in observable states:

```
invariant forall old A a :: a.g % 2 == 0;
```

The verifier inlines external calls by default into the verification condition. This means that the verifier checks that there are corresponding external calls in both implementations (same method name and signature, corresponding receiver and parameters) and that the coupling invariant holds. The verifier then drops all knowledge about both heaps, assumes that the invariant holds, and continues the verification process at the point where the external calls happened by assuming that the returned values are corresponding. Instead of inlining external calls, the verifier can also be configured to prove the given example in two separate steps (up to that point, and then from that point on). The drawback about splitting the verification condition is that information about the stack then needs to be encoded in the invariant. The splitting program point is defined using a **place** definition. The splitting behavior can be deactivated for places with the **nosplit** option. The invariant then needs to be strengthened by stating that for all library stack slices, if the topmost stack frame in the stack slice is at that particular call of the method run, the value stored in the local variable i is 4.[3]

```
place p1 = call run in line 5 of old A;
place p2 = call run in line 5 of new A;
invariant forall int s :: librarySlice(old, s) ∧ at(p1, s) ⇒ eval(p1, s, i) == 4;
```

The ISL specification relies on the built-in function *librarySlice* to characterize whether the integer s is a valid index to a library stack slice of the stack of the old library implementation. The function *at* denotes that the execution is at a certain program point in a specific stack slice. The function *eval* allows to access the value of local variables at a certain program point. For example, the application *eval*(p1, s, i) yields the value of the local variable i at the program point denoted by p1 in the stack slice indexed by s. Place definitions enable ISL specifications to be more strongly typed. For *eval* expressions to be well-formed in ISL, it must be ensured that the execution is at the specified place. This is done by guarding the *eval* expressions with corresponding *at* expressions.

Callback with lock example. We present in Figure 7.5 a Java adaptation of a tricky example given by Ahmed, Dreyer, and Rossberg [ADR09]. Let us first ignore the lines of code that make use of the lock field. In this variation of the callback example, the run method is called in line 8 of the old implementation before incrementing the value x. In the new implementation, the value of x is first stored in a variable n. After the callback in line 9, the value stored in n is used to increment x. In the setting of callbacks which might call the inc method again and again, the value stored in n might be outdated upon return of the run method. To safeguard against callbacks, the implementations use a lock variable that disables the functionality of the inc method for reentrant calls. Under this locking behavior, both implementations can be proven equivalent with the following coupling invariant:

```
invariant forall old A o1, new A o2 ::
  o1 ∼ o2 ⇒ o1.x == o2.x ∧ o1.lock == o2.lock;
place p = call run in line 9 of new A nosplit;
invariant forall int s :: librarySlice(new, s) ∧ at(p, s) ⇒
  eval(p, s, this.lock ∧ n == this.x);
```

To ensure that the value of n is not changed when the method run is running, we assert that in all stack slices at the place p2 the value of x is equal to n.

[3]A weaker invariant, stating that the value is even, is also sufficient.

```
public interface C { // old and new impl.
  public void run();
}
```

```
public class A { // old impl.
  private int x;
  private boolean lock;
  public void inc(C c) {
    if (c == null) return;
    if (!lock) {
      lock = true;
      c.run();
      x = x + 1;
      lock = false;
    }
  }
  public int get() {
    return x;
  }
}
```

```
public class A { // new impl.
  private int x;
  private boolean lock;
  public void inc(C c) {
    if (c == null) return;
    if (!lock) {
      lock = true;
      int n = x;
      c.run();
      x = n + 1;
      lock = false;
    }
  }
  public int get() {
    return x;
  }
}
```

Figure 7.5: Callback with lock example

Awkward example. We present in Figure 7.6 another Java adaptation of an example given by Ahmed, Dreyer, and Rossberg [ADR09] (which they could not prove using their reasoning method), a variation of the "awkward" example of Pitts and Stark [PS98]. In the setting of callbacks, the value of the field x in the old implementation changes back and forth between 0 and 1. Due to the shape of the call stack, however, the value that is returned is always 1, which is given as a proof obligation by the new implementation that always returns 1. For simplicity, let us first assume that the field x and the methods are static. The example then is specified using the following invariant:

```
place p = call run in line 7 of old Awk nosplit;
invariant forall int s :: librarySlice(old, s) ∧ s == topSlice(old) − 1 ∧ at(p, s)
  ⇒ old Awk.x == 1;
```

The invariant states that if the top-most stack slice that belongs to the library is at the second invocation of the run method, then the value of x is 1.

```
public interface C { // old and new impl.
  public void run();
}
```

```
public class Awk { // old impl.      1
  private int x;                     2
  public int exec(C c) {             3
    x = 0;                           4
    if (c != null) c.run();          5
    x = 1;                           6
    if (c != null) c.run();          7
    return x;                        8
  }                                  9
}                                   10
```

```
public class Awk { // new impl.      1
  public int exec(C c) {             2
    if (c != null) c.run();          3
    if (c != null) c.run();          4
    return 1;                        5
  }                                  6
}                                    7
```

Figure 7.6: Awkward example

If the field x is non-static (as in the code listing), the invariant needs to account for multiple objects. We can then not talk about the top-most library stack slice, but the top-most stack slice that is associated with an object. We say that the value of x is 1 for a certain object if the highest stack slice for this object is at p2, i.e., there is no higher stack slice for that object that is at p1. Written formally, the invariant thus becomes:

```
place p1 = call run in line 5 of old Awk nosplit;
place p2 = call run in line 7 of old Awk nosplit;
invariant forall int s :: librarySlice(old, s) ⇒ at(p1, s) ∨ at(p2, s);
invariant forall int s :: librarySlice(old, s) ∧ at(p2, s)
  ∧ !(exists int s2 :: s2 > s ∧ librarySlice(old, s2) ∧ at(p1, s2)
          ∧ eval(p2, s, this) == eval(p1, s2, this))
  ⇒ eval(p2, s, this.x == 1);
```

The examples shown in this section demonstrate that our approach can handle complex callback scenarios.

7.4 Control Flow Relations

In a setting with no callbacks, library implementations can also hide complex control flow. Reasoning about program behavior in the setting of loops and recursion usually requires some kind of induction principle. To prove steps that encompass methods with complex loops or recursion, we allow the programmer to specify that the coupling relation can relate custom user-definable non-observable states. The library developer specifies these program states for the old and new library implementation. These states are defined similarly to *conditional breakpoints* (debugging terminology) and are called **local places**. Using the local places, the library developer can establish custom relations using a coupling invariant. To facilitate the specification of such properties that hold only at local places, we introduce **local invariants**. Local invariants are defined in a similar way as coupling invariants but only have to hold in local places, i.e., synchronization points where the library is in control of execution.

7.4.1 Synchronous Execution

As an example for local places and local invariants, consider the example in Figure 7.7, adapted from Barthe, Crespo, and Kunz [BCK11]. The old implementation uses a **for** loop and the new implementation uses a **while** loop. The body of the **for** loop is executed once more than the body of the **while** loop, namely for the value `i == 0`. To prove that both methods yield the same results, we establish a local simulation that relates the second iteration of the loop in the old implementation and the first iteration of the loop in the new implementation and then consecutive loop iterations. We first characterize the states in both library implementations that we want to relate. The first local place definition `p1` denotes all program states where the statement to be executed is the first one in line 5 of the old implementation and where the value of the local variable `i` is positive. Similarly, the local place definition `p2` denotes the states where the execution is at the beginning of line 6 of the new implementation.

The next step is to define the relation that ties the states of the two implementations together. As we want to reason locally about the execution here, we are only interested in the top-most stack slice. The overloaded function *at* yields the top-most stack splice if no additional parameter is provided. As only a single local place is defined for each implementation, the implicit invariant

```
public class C { // old impl.              1
  public int m(int n){                     2
    int x = 0;                             3
    for(int i=0; i<n; i++){                4
      x += i;                              5
    }                                      6
    return x;                              7
  }                                        8
}                                          9
```

```
public class C { // new impl.              1
  public int m(int n){                     2
    int x = 0;                             3
    int i = 1;                             4
    while(i<n){                            5
      x += i;                              6
      i++;                                 7
    }                                      8
    return x;                              9
  }                                       10
}                                         11
```

```
local place p1 = line 5 of old C when i > 0;
local place p2 = line 6 of new C;
local invariant at(p1) ∧ at(p2) ⇒
     eval(p1, n) == eval(p2, n)
  ∧ eval(p1, x) == eval(p2, x)
  ∧ eval(p1, i) == eval(p2, i);
```

Figure 7.7: One-off loop example

at(p1) ⇔ *at*(p2) must hold. The application *eval*(p1, n) yields the value of the local variable n at the program point denoted by p1 (in the top-most stack slice). The local invariant states that in coupled non-observable states, the values of the local variables n, x, and i of both implementations coincide. Due to the introduction of local places, four program paths need to be considered to reason about the *new* implementation, depicted in the following control flow diagram on the right: (1) From an observable state, depicted in plain white, calling the method m with an input $n \leq 1$ that leads to returning from the method and resulting again in an observable state (i.e., the self-loop on the white state), (2) from an observable state calling the method m with $n > 1$ to the program place p2, (3) from the program place p2 going into the next loop iteration to the program place p2 if $i{-}1 < n$ in the pre-state, and (4) by returning from p2 to an observable state if $i{-}1 \geq n$.

$$\begin{tikzcd} \boxed{} \arrow[loop above, "\text{call/return m}"] \arrow[r, bend left, "\text{call m}"] & \boxed{p1} \arrow[loop above] \arrow[l, bend left, "\text{return m}"] \end{tikzcd} \qquad \begin{tikzcd} \boxed{} \arrow[loop above, "\text{call/return m}"] \arrow[r, bend left, "\text{call m}"] & \boxed{p2} \arrow[loop above] \arrow[l, bend left, "\text{return m}"] \end{tikzcd}$$

For the old implementation, an infinite number of paths must be considered: Executing the body of the for loop zero times, once, twice, thrice, etc. Luckily, we can prove that a certain depth is never reached.[4] This means that either a local place or an observable state (via external method call with split option) is encountered before or we have an infeasible path. In the example, two "executions" of the loop body are sufficient because during the second iteration the place p1 with $i > 0$ is always reached. Effectively, this reduces the feasible paths to the ones depicted in the control flow diagram on the left. Comparing this diagram to the one on the right illustrates the synchronization of the executions of both implementations.

The proof obligations are as follows. We have to check that a local place is reached in the old implementation if and only if a local place is reached in the new implementation. If such local places are reached, the invariants (and in particular local invariants) are checked. We also have to assume two arbitrary local places such that the local invariants hold and start execution from these places and check whether the continuations behave similarly. For the example, the local invariant is proved to hold initially, for each iteration and finally the last iteration guarantees that the values returned by both methods are the same.

Cubes example. We give another example for synchronous execution in Figure 7.8, which is adapted from Leino and Yessenov [LY12]. Both methods take as parameter an integer value n and compute the sum of cubes from 1 to n. The first method implementation computes the sum $\sum_{i=0}^{n} i^3$, whereas the second method implementation computes the same result with a different formula $(\sum_{i=0}^{n} i)^2$ that uses a single multiplication. We define two local places to synchronize the loop iterations of both method bodies. The coupling invariant, adapted directly from Leino and Yessenov, then connects the values as follows:

```
local place inLoop1 = line 6 of old C when i < n;
local place inLoop2 = line 6 of new C when i < n ∧ 2 ∗ t == i ∗ (i + 1);
local invariant at(inLoop1) ∧ at(inLoop2) ⇒
    eval(inLoop1, n) == eval(inLoop2, n)
  ∧ eval(inLoop1, i) == eval(inLoop2, i)
  ∧ eval(inLoop1, s) == eval(inLoop2, t ∗ t);
```

[4]The BCVERIFIER unrolls loops and recursion automatically unrolled up to a user-definable depth d and tries to prove that depth $d+1$ is never reached.

```
public class C { // old impl.
  public int cubes(int n){
    int i = 0;
    int s = 0;
    while(i < n){
      i++;
      s += i * i * i;
    }
    return s;
  }
}
```

```
public class C { // new impl.
  public int cubes(int n){
    int i = 0;
    int t = 0;
    while(i < n){
      i++;
      t += i;
    }
    return t * t;
  }
}
```

Figure 7.8: Cubes example

7.4.2 Asynchronous Execution

More complex relations can be specified that allow to relate one state in one implementation to many states in the other implementation. We say that one of the implementations is **stalled** while the other executes. Only one implementation can be stalled at a time. If the old implementation is stalled, then we additionally need to prove that the new implementation is not diverging. The proof obligation is stated in terms of a **termination measure**, an integer expression which must be positive and strictly decreasing between consecutive local places. The application of stalling places and termination measure can be seen in the example of Figure 7.9. In this example, we want to prove that the new implementation of the method m terminates. As a reference implementation for the method m, we use an implementation that obviously terminates in all cases.

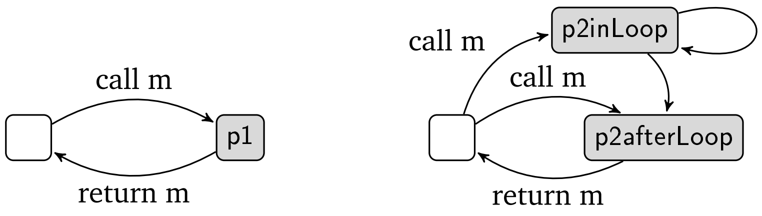
We define a local place p1 for the body of the old method and two local places p2inLoop and p2afterLoop in the new method, one in the loop and one after the loop. We then need to show that the new implementation always reaches the p2afterLoop place. Graphically depicted, we have the following situation:

```
public class C { // old impl.
  public void m(int n) {
    return;
  }
}
```

```
public class C { // new impl.
  public void m(int n) {
    int x = 0;
    for (int i = 0; i < n; i++) {
      x = x + i;
    }
    return;
  }
}
```

```
local place p2inLoop = line 5 of new C;
local place p2afterLoop = line 7 of new C;
local place p1 = line 3 of old C
  stall when at(p2inLoop) with measure
    if at(p2inLoop) then eval(p2inLoop, n − i) else 0;
local invariant at(p2inLoop) ⇒ eval(p2inLoop, i < n);
```

Figure 7.9: Termination example



The idea is now to stall the computation of the old implementation until the new implementation reaches p2afterLoop. This is specified by the stall condition *at*(p2inLoop) which states that the old implementation is stalled at the place p1 if the computation in the new implementation starts in the place p2inLoop. The termination measure is specified as a positive integer expression. To ensure that the expression is positive and the measure decreases from p2inLoop to p2afterLoop, the local invariant $i < n$ is added. Writing *eval*(p2inLoop, $n - i$) is short for *eval*(p2inLoop, n) − *eval*(p2inLoop, i). The condition $i < n$ is needed[5] as the measure can otherwise not be proved to be positive.

The previous for vs. while loop example (synchronous execution) could also be proved by defining the local place p1 without the condition $i > 0$ but stalling

[5]Note that the condition *eval*(p2inLoop, $i < n$) could also be put as a **when** clause for the local place p2inLoop.

the place p2 until the place p1 is reached the second time. For this, the place p2 is defined as follows:

```
stall when at(p1) ∧ eval(p1, i) == 0;
```

The last part of the local invariant (*eval*(p1, i) == *eval*(p2, i)) then needs to be replaced by

```
eval(p1, i) >= 0 ∧ eval(p1, n) > 1 ∧
eval(p2, i) == (if eval(p1, i) == 0 then 1 else eval(p1, i))
```

where the first line fixes the value ranges of i and n for the verifier to only consider reasonable paths. The second line establishes the connection between the value of the i variables in the first iteration (**then** clause), and then successive iterations (**else** clause).

Local places are a powerful concept and have many other uses: (1) They can be used to handle diverging computations, for example, to show that two methods terminate or diverge for the same inputs. (2) Synchronous execution in combination with asynchronous execution can be used to verify that a recursive method and a method with a loop behave equivalently. If the recursive method does the computation before the recursive call, each recursive call can be associated with one iteration of the loop. Asynchronous computations can then be used to stall the loop-based implementation when popping the stack frames of the recursive function.

7.5 Larger Case Study: ObservableList

In this section, we show that the techniques presented so far can directly be applied to a complex example studied by Banerjee and Naumann [BN05a]. We also illustrate the facilities of ISL to describe invariants that allow reasoning about the shape of complex call stacks, needed in the setting of recursive method calls. Figures 7.10 and 7.11 present two library implementations of the observer pattern [Gam+95]. Each implementation consists of a public Observer interface, to be implemented by clients of the library, a public Observable class to register and notify observers and a non-public Node class that is used by the Observable class to manage the observers in a singly linked list. In addition, the Observable class provides a method get(**int** i) to retrieve the *i*-th observer that was registered using the method add and a method iterator to iterate over the observers using the interfaces and classes presented in Figure 7.12 (going beyond the example of Banerjee and Naumann [BN05a]). Formal reasoning about (and verification of) such implementations is highly non-trivial as the sizes of the data structures, as well as the computations (e.g., number of loop iterations in the old notifyAllObs() method implementation), are unbounded.

The data and control flow representations of the implementations differ in a number of ways. The old implementation of the Observable class stores the first observer directly in the first Node object whereas the new implementation uses a sentinel node. The Node class of the old implementation provides no methods, whereas the new implementation provides a proper constructor and getter and setter methods. The method add illustrates the manipulation of the internal representation, and the method get shows how internal control flow can depend on input values. The methods of the new implementation are written in a clearer and more concise way than their counterpart in the old implementation. The method notifyAllObs, illustrating the possibility of callbacks, loops over all nodes in the old implementation and notifies the observers, whereas the new implementation relies on the recursive method notifyRec defined in the new Node class.

We have shown backward compatibility of the implementations using the BCVerifier tool. In the following, we give the most relevant parts of the ISL specification that was used. The first step is to establish a relation between the heap state of the old and the new implementation. Similarly to the OBool example, we use a bijection between the internal Node objects. The bijection is constructed such that the pair (**null**, **null**) is part of the relation. For two corresponding Observable objects, the first node of the old implementation and the first real node, skipping the sentinel node, are in the bijection. The sentinel node is not in the relation. If two nodes are in the relation, the nodes referred to by the next fields are also in the relation, and the observer objects which are stored in the ob fields are corresponding:

```
var binrelation bij = add(empty(), null, null);
invariant bijective(bij) ∧ related(bij, null, null);
invariant forall old Observable l1, new Observable l2 ::
  l1 ∼ l2 ⇒ related(bij, l1.fst, l2.snt.next);
invariant forall old Node n1, new Observable l2 :: !related(bij, n1, l2.snt);
invariant forall old Node n1, new Node n2 ::
  related(bij, n1, n2) ⇒ related(bij, n1.next, n2.next) ∧ n1.ob ∼ n2.ob;
```

Similarly to the OBool example, to verify the add methods, the bijection needs to be updated with the right values:

```
var old Node x1 = null;
var new Node x2 = null;
```

```
package util; // old and new impl.
public interface Observer {
  public void notifyObs();
}
```

```
package util; // old impl.
class Node {
  Observer ob;
  Node next;
}
```

```
package util; // new impl.
class Node {
  private Observer ob;
  private Node next;

  Node(Observer ob, Node next) {
    this.ob = ob;
    this.next = next;
  }

  Node getNext() {
    return next;
  }

  void setNext(Node next) {
    this.next = next;
  }

  Observer getObs() {
    return ob;
  }

  void notifyRec() {
    ob.notifyObs();
    if (next != null) {
      next.notifyRec();
    }
    return; // dummy statement
  }
}
```

Figure 7.10: ObservableList example (Observer and Node)

```java
package util; // old impl.
public class Observable {
  Node fst;
  int modCount = 0;

  public void add(Observer ob) {
    if (ob == null) return;
    Node newNode = new Node();
    newNode.ob = ob;
    newNode.next = fst;
    fst = newNode;
    modCount++;
  }

  public Observer get(int i) {
    int c = 0;
    Node n = fst;
    while(c < i) {
      if (n != null) {
        n = n.next;
        c++;
      } else { break; }
    }
    if (n != null) { return n.ob; }
    else { return null; }
  }

  public void notifyAllObs() {
    Node n = fst;
    while (n != null) {
      n.ob.notifyObs();
      n = n.next;
    }
    return; // dummy statement
  }

  public Iterator iterator() {
    return new MyIter(this);
  }
}
```

```java
package util; // new impl.
public class Observable {
  Node snt = new Node(null, null);
  int modCount = 0;

  public void add(Observer ob) {
    if (ob == null) return;
    snt.setNext(new Node(ob,
                snt.getNext()));
    modCount++;
  }

  public Observer get(int i) {
    Node n = snt.getNext();
    for (int c = 0; c < i; c++) {
      if (n == null) return null;
      n = n.getNext();
    }
    if (n == null) return null;
    return n.getObs();
  }

  public void notifyAllObs() {
    Node n = snt.getNext();
    if (n != null) n.notifyRec();
  }

  public Iterator iterator() {
    return new ObsIter(this);
  }
}
```

Figure 7.11: ObservableList example (Observable)

```
package util; // old and new impl.
public interface Iterator {
  boolean hasNext();
  Observer next();
}
```

```
package util; // old impl.
class MyIter implements Iterator {
  private Node n;
  private Observable o;
  private int cnt;

  MyIter(Observable o) {
    this.o = o;
    this.n = o.fst;
    this.cnt = o.modCount;
  }

  public boolean hasNext() {
    if (o.modCount != cnt)
      return false;
    return n != null;
  }

  public Observer next() {
    if (o.modCount != cnt)
      return null;
    if (n == null)
      return null;
    Observer temp = n.ob;
    n = n.next;
    return temp;
  }
}
```

```
package util; // new impl.
class ObsIter implements Iterator {
  private Node n;
  private Observable o;
  private int cnt;

  ObsIter(Observable o) {
    this.o = o;
    this.n = o.snt.getNext();
    this.cnt = o.modCount;
  }

  public boolean hasNext() {
    if (o.modCount != cnt)
      return false;
    return n != null;
  }

  public Observer next() {
    if (o.modCount != cnt)
      return null;
    if (n == null)
      return null;
    Observer temp = n.getObs();
    n = n.getNext();
    return temp;
  }
}
```

Figure 7.12: ObservableList example (Iterator, MyIter and ObsIter)

```
invariant x1 == null ∧ x2 == null;
local place p1 = line 11 of old Observable assign x1 = newNode nosync;
local place p2 = line 7 of new Node assign x2 = this nosync;
assign bij = if x1 != null ∧ x2 != null then add(bij, x1, x2) else bij;
assign x1 = null;
assign x2 = null;
```

The local places have the option **nosync** to configure that these places are not synchronized to places in the old implementation, i.e., they only serve to insert the assignment expressions in the code.

To verify the get methods, we define local places that synchronize the executions of the **while** and the **for** loop:

```
local place q1 = line 20 of old Observable when c < i ∧ n != null;
local place q2 = line 16 of new Observable when c < i ∧ n != null;
local invariant at(q1) ⇔ at(q2);
local invariant at(q1) ∧ at(q2) ⇒ related(bij, eval(q1, n), eval(q2, n))
  ∧ eval(q1, c) == eval(q2, c) ∧ eval(q1, i) == eval(q2, i);
```

Finally, to verify the notifyAllObs methods, we define local places that synchronize each loop iteration to a recursive call. Here it is important to encode the shape of the call stack for the new implementation, i.e., that the method notifyRec was originally called from the method notifyAllObs:

```
local place pcall = call notifyRec in line 25 of new Observable nosync;
local place pn1 = line 31 of old Observable when n != null;
local place pn2 = line 24 of new Node when topFrame(new) > 0 ∧ at(pcall, 0);
local invariant at(pn1) ⇔ at(pn2);
local invariant at(pn1) ∧ at(pn2) ⇒ related(bij, eval(pn1, n), eval(pn2, this));
```

The function *topFrame*(**new**) yields the offset of the current (top) stack frame in the current library stack slice of the new implementation. The function *at*(pcall, 0) determines whether the stack frame at offset 0 (bottom of the current stack slice) is currently at the place pcall. Similar as for the get methods, we state that the node referred to by n in the loop and the node referred to by **this** in the recursive method are in the bijection bij. We need to take special care that this property is preserved as well after the notification of an observer with the notifyObs method. Notifying an observer can lead to reentrant calls. A program context can for example call the method *add* during the notification. Similar as for the heaps (see Section 7.3), all knowledge about ghost variables, that is not stated in the coupling invariant, is dropped when the call is inlined into the verification condition. This means that after the calls we have lost the information that the variables n and **this** are in the bijection. This information needs to be added as an invariant:

```
place pc1 = call notifyObs in line 31 of old Observable nosplit;
place pc2 = call notifyObs in line 24 of new Node nosplit;
invariant forall int s :: librarySlice(old, s) ∧ at(pc1, s) ∧ at(pc2, s) ⇒
        related(bij, eval(pc1, s, n), eval(pc2, s, this));
```

The **nosplit** option denotes that the call is inlined into the verification condition. The invariant quantifies (implicitly) over all library stack slices, which means that the property must not only hold for the topmost stack slice, but all stack slices. This allows the verifier to check that the property is not destroyed by other interactions (e.g., by calling the method *add* during the notification). As nodes are only added to the bijection and never removed, the property is trivially preserved.[6]

Each loop iteration is connected to a call of the recursive method. Finally, we have the situation where the loop condition and the condition for the recursive call do not hold anymore. In this case, the execution in the old implementation leaves the loop. In the new implementation, the execution is left with a stack slice which has a size that represents the number of Node objects visited. As this size is not statically fixed, the path that ends all the notifyRec method invocations is not bounded in size and the verifier needs assistance to prove termination. We introduce two local places, one after the loop and another at the end of the notifyRec method. As our tool chain currently only allows the definition of local places before an existing statement, we introduce two dummy return statements in the library implementations. Finally, we use asynchronous execution by stalling the old implementation at the old place up to the point where the stack slice of the new implementation has size 1. The size of the stack slice of the new library implementation serves as the termination measure:

```
local place qn1 = line 34 of old Observable
  stall when topFrame(new) > 1 with measure topFrame(new);
local place qn2 = line 28 of new Node when topFrame(new) > 0 ∧ at(pcall, 0);
local invariant at(qn1) ⇔ at(qn2);
```

The last step is to prove is that the executions starting from qn1 and qn2 are properly simulated. This follows directly from the shape of the stack specified for qn2 and the negated stalling condition for qn1.

For the iterators, a few additional invariants are necessary. First, the referenced lists must be non-null and cnt must not exceed modCount. Then, for corresponding iterators, the cnt value must be equal and the referenced lists corresponding. If the iterators are not invalidated (cnt == o.modCount), the referenced nodes must be in the internal bijection bij.

[6]Note that this still allows nodes to be removed from the list.

```
invariant forall old MyIter i :: i.o != null; // same for new ObsIter
invariant forall old MyIter i :: i.cnt <= i.o.modCount; // same for new ObsIter
invariant forall old MyIter i1, new ObsIter i2 ::
  i1 ∼ i2 ⇒ i1.cnt == i2.cnt ∧ i1.o ∼ i2.o;
invariant forall old MyIter i1, new ObsIter i2 ::
  i1 ∼ i2  ∧ i1.cnt == i1.o.modCount ⇒ related(bij, i1.n, i2.n);
```

This concludes the example. In this chapter, we have seen that ISL, using a manageable set of specification concepts, can be readily used to specify coupling relations between complex changes in library implementations. In the following chapter, a more structured overview of ISL is given, together with a tool that automatically verifies all of the presented examples.

8 Specification and Tool Support

Computers do not solve problems
– computers carry out solutions, specified by people, to problems.

— D. D. Spencer

This chapter introduces the implementation of the reasoning approach, detailed in the previous chapters. The BCVERIFIER tool takes two library implementations and an ISL specification as input, and checks backward compatibility. It fully verifies all the examples in this thesis. Section 8.1 presents an overview of the ISL specification language, Section 8.2 the BCVERIFIER tool, and Section 8.3 related work on specification and verification approaches.

8.1 Invariant Specification Language

ISL is a first-order logic-like specification language that provides facilities to express complex data and control flow relations between two library implementations. In the following, we give a short overview of the language.

8.1.1 Syntax

The syntax of ISL is presented in Figure 8.1. Non-terminals are represented in ALL CAPS. We use the meta-symbol | to denote alternatives and the brackets $[\ldots]$ to group elements. The meta-symbol$[\mathsf{X}]^?$ is used to denote an optional item X and $[\mathsf{X}]^*$ to denote an arbitrary sequence of elements X. The non-terminal ID represents identifiers and INT integer constants. In contrast to the previous chapters where specifications were written using logical connectives, we use the ASCII representation of ISL in this chapter. The logical connectives have the following ASCII representation in ISL: 1) $\Rightarrow$ becomes ==> 2) $\Leftrightarrow$ becomes <==> 3) $\wedge$ becomes && and 4) $\vee$ becomes ||.

8.1.2 Types and Semantics

In the following, we present the types occurring in ISL specifications and an informal semantics. A full formalization is out of scope for this thesis.

ISL supports the following types:

- Java primitive types **boolean** and **int**.
- Java class and interface types parameterized with library version. The library version is either **old** or **new**. The Java type can be referenced by the fully qualified name or just by the name of the class if it is unambiguous.
- The special **place** type that is used to type places.
- The special built-in **binrelation** type defining binary relations on reference values (i.e., object identifiers or **null**).

Built-in functions. ISL only supports built-in functions, presented in the following.

- **boolean** *exposed*(Object o) checks if the object o is exposed.
- **boolean** *createdByLibrary*(Object o) checks if the object o is created by the library or the program context.
- **int** *topSlice*(VERSION v) returns the index of the top-most stack slice in the stack of the old or the new library implementation.
- **int** *topFrame*(VERSION v, **int** slice) returns the index for the top-most stack frame in the stack slice indexed by slice for the old or the new library implementation.
- **int** *topFrame*(VERSION v) is a shorthand for *topFrame*(v, *topSlice*(v)).
- **boolean** *librarySlice*(VERSION v, **int** slice) checks if the stack slice indexed by slice in the stack of the old or new library implementation belongs to the library.

```
SPECIFICATION ::= [DECLARATION]*

DECLARATION ::= [local]? invariant EXPRESSION ;
    | [local]? place ID = PROGPOS [when EXPRESSION]?
        [STALLCONDITION]? [ASSIGN]* [nosplit]? [nosync]?;
    | var VARDEF = EXPRESSION ;
    | ASSIGN ;

PROGPOS ::= line INT of TYPE | call ID in line INT of TYPE

ASSIGN ::= assign ID = EXPRESSION

STALLCONDITION ::= stall when EXPRESSION
        [with measure EXPRESSION]?

EXPRESSION ::= ID
    | true | false | null | INT // literals
    | EXPRESSION . ID // field access
    | TYPE . ID // static field access
    | UNARYOPERATOR EXPRESSION
    | EXPRESSION BINARYOPERATOR EXPRESSION
    | EXPRESSION instanceof TYPE // Java instanceof operator
    | if EXPRESSION then EXPRESSION else EXPRESSION
    | ( EXPRESSION ) // parenthesized expression
    | ID ( [EXPRESSION [, EXPRESSION]*]? ) // function call
    | [forall | exists] VARDEF [, VARDEF]* :: EXPRESSION
    | VERSION

VARDEF ::= TYPE ID

TYPE ::= int | boolean | binrelation | VERSION ID[.ID]*

VERSION ::= old | new

BINARYOPERATOR ::= ~ // correspondence relation
    | + | − | * | / | % | == // Java operators
    | != | < | <= | > | >= | && | || // Java operators
    | ==> | <==> // other logical operators

UNARYOPERATOR ::= ! | − // Java operators
```

Figure 8.1: Syntax of ISL

- **boolean** *librarySlice*(VERSION v) is a shorthand for *librarySlice*(v, *topSlice*(v)).
- **boolean** *at*(**place** p, **int** slice, **int** frame) checks if the stack frame indexed by frame in the stack slice indexed by slice is currently at place p.
- **boolean** *at*(**place** p, **int** slice) is a shorthand for *at*(p, slice, *topFrame*(v, slice)) where v is the version of the place p.
- **boolean** *at*(**place** p) is a shorthand for *at*(p, *topSlice*(v)) where v is the version of the place p.
- T *eval*(**place** p, **int** slice, **int** frame, EXPRESSION<T> e) evaluates the expression e in the context of the place p. This means that local Java variables, that are defined at the given place, can be used. The values of the variables will be taken from the stack frame indexed by frame of the stack slice indexed by slice. The expression e is typed using the type of the local Java variables at place p.
- T *eval*(**place** p, **int** slice, EXPRESSION<T> e) is a shorthand for *eval*(p, slice, *topFrame*(v, slice), e) where v is the version of the place p.
- T *eval*(**place** p, EXPRESSION<T> e) is a shorthand for *eval*(p, *topSlice*(v), e) where v is the version of the place p.
- **boolean** *related*(**binrelation** b, **old** Object o1, **new** Object o2) checks if the pair (o1, o2) is in the relation b.
- **binrelation** *empty*() returns the empty binary relation on reference values.
- **binrelation** *add*(**binrelation** b, **old** Object o1, **new** Object o2) returns the relation which is the same as b except where the pair (o1, o2) is added.
- **binrelation** *remove*(**binrelation** b, **old** Object o1, **new** Object o2) returns the same relation as b except where the pair (o1, o2) is removed.

Operators. All boolean operators are short-circuit operators and evaluated from left to right. The right expression is only evaluated if the value of the left expression does not already fix the value of the overall expression. The correspondence operator $\sim$ expects two reference types: one from the old library implementation on the left hand side and one from the new library implementation on the right hand side. An expression o1 $\sim$ o2 evaluates to **true** if and only if o1 and o2 are two objects in correspondence or o1 and o2 are both **null**.

Top-level constructs. Top-level expressions appearing in local place definitions are implicitly wrapped with *eval* for that place. This means that local Java variables can directly be used in the definition without using *eval* expressions (see, e.g., the definition of p1 in Figure 7.7). Termination measures are specified by integer expressions which must be positive. This becomes, together with the decreasing property, part of the proof obligations.

Global invariants (without the keyword **local**) have to hold in observable states and at local places. Local invariants must hold at local places. Local invariants can be considered as syntactic sugar: **local invariant** e; is equivalent to **invariant** *librarySlice*(*topSlice*(**old**)) ⇒ e;.

Local places with a stalling condition that are defined for the old implementation must have a termination measure, as we must show that there is only a finite number of steps in the new implementation whenever execution is stalled in the old implementation. Conversely, stalling local places that are defined for the new implementation must not have a termination measure.

8.1.3 Well-formedness

To check whether a place or an invariant is well-formed, a separate proof obligation is generated. Typical well-formedness conditions for expressions are the following:

- There must not be any division by zero.
- **null** must not be dereferenced.

As all boolean operators are short-circuit operators the first expression in the following example is well-formed and the second expression is not well-formed.

```
x > 0 && y/x == 2
y/x == 2 && x > 0
```

The order in which invariants are defined is important. Preceding invariants can be used to show that following invariants are well-formed. In the following example the first invariant states that c.x is never zero and thus the second invariant is well-formed. If the invariants were defined in reverse order, the well-formedness of the division could not be shown.

```
invariant forall old C c :: c.x != 0
invariant forall old C c :: 10 / c.x > 3
```

In the following, we give a definition of well-formedness for expressions as a function WD[...]. We use the meta brackets [] to distinguish them from standard brackets in the ISL language. TR[...] denotes the translation function, which is not detailed in the thesis. We present the well-formedness checks for the most important expressions.

```
WD[if e then e1 else e2] := WD[e] ∧
  (TR[e] ⇒ WD[e1]) ∧ (¬TR[e] ⇒ WD[e2])
WD[e1 && e2] := WD[e1] ∧ (TR[e1] ⇒ WD[e2])
WD[e1 ==> e2] := WD[e1 && e2]
WD[e1 || e2] := WD[e1] ∧ (¬TR[e1] ⇒ WD[e2])
WD[e1 / e2] := WD[e1] ∧ WD[e2] ∧ TR[e2 != 0]
WD[e1 % e2] := WD[e1 / e2]
WD[e.f] := WD[e] ∧ TR[e != null]
WD[exposed(e)] := WD[e] ∧ TR[e != null]
WD[createdByLibrary(e)] := WD[e] ∧ TR[e != null]
WD[at(p, es, ef)] := WD[es] ∧ WD[ef] ∧
  TR[0 <= es] ∧ TR[es <= topSlice(v)] ∧
  TR[0 <= ef] ∧ TR[ef <= topFrame(v,es)]
  // where v is the version of the place p
WD[eval(p, es, ef, e)] := WD[at(p,es,ef)] ∧ TR[at(p,es,ef)] ∧ WD[e]
```

8.1.4 Discussion

The inclusion of some ISL features, such as binary relations and ghost variables, was example-driven. The core features are, however, directly motivated by the need to access the enhanced configurations (heap and stack) as well as using the correspondence relation to relate reference values of both configurations. Here, the stack allows direct indexing (with positive integer expressions), whereas such an access to the heap is not useful due to the non-deterministic nature of the object allocator. Heap access is realized by using local variables on the stack or by quantifying over all objects of a certain type. The syntax of ISL is kept very simple. For example, stack access is realized with a few pre-defined functions. A more powerful syntax would certainly make specifications feel more natural. One important and conflicting goal of the current language draft is however to facilitate the rapid prototyping of new concepts.

Further more powerful features might be desirable for other examples. Migrating more features from specification-based techniques in the single-program world would probably be extremely useful, e.g., pre-post specifications of methods, further specification-only types such as sets, lists and maps, and higher-level specification constructs for typically occurring relations. Other features become superfluous in the two-program world or can already be nicely simulated with the given approach. A good example for this are ghost variables on the instance level. Currently ISL only supports ghost variables that are defined in a global scope. Ghost fields or local variables cannot directly be specified but can easily be simulated by standard fields and local variables. A library developer can introduce these additional variables (and corresponding assignments) directly in the implementations. An additional proof obligation is then to show that the same library implementation without ghost variables is equivalent to the one using ghost variables, i.e., that the ghost variables have no effect on the observable behavior.

8.2 The BCVerifier Tool

The main task of the BCVerifier implementation is to generate a representation of the libraries under investigation, the specification of the coupling invariant and the proof obligations for the intermediate verification language Boogie [Lei08]. Boogie is usually used as an intermediate language to study correctness of single programs with respect to specifications, for example Spec# [BLS05], Dafny [Lei10], or Chalice [LMS09]. The purpose of Boogie is to facilitate the generation of verification conditions for these complex programming languages. Such generation is split into two parts; first, the program and proof obligations are transformed into a corresponding Boogie representation, from which the Boogie tool [Bar+05] can then generate logical formulas which can be fed to theorem provers. For our studies, the SMT solver Z3 [MB08] was used. The Boogie encoding is explained in the Master thesis of Weber [Web12]. A tricky part is to encode the interleaving of the executions of both library implementations properly. In contrast to existing encodings of the aforementioned languages, our encoding does not only reify a single stack frame but the complete stack in the Boogie program. To encode the complex control flow from and to the two library implementations (e.g., local places), both library implementations are generated into a single Boogie procedure. The control flow is then encoded using unstructured control statements (`goto`).

The BCVerifier accepts specifications using one of two possible syntaxes: (1) ISL (see Section 8.1) is a high level specification language to specify the coupling invariant. Specifications in this language are validated against the Java library code and transformed into a consistent Boogie specification. (2) The low-level syntax uses pure BOOGIE expressions, which are directly inserted as is into the generated BOOGIE representation.

The BCVERIFIER also offers a number of configuration options. The recursion and loop unroll cap needs to be fixed to a reasonable value by the programmer. This triggers how often the BOOGIE program is (soundly) unrolled. For example, the OBool example needs a higher unroll count as the Cell example because of control flow paths that involve internal method and constructor calls. Unfortunately, unrolling is done globally at the level of the generated interleaved BOOGIE procedure. Even though most paths through the unrolled procedure are infeasible, the unrolled procedure is fed in its whole to the SMT solver. Here, a two-pass approach would be better, incrementally checking reachability of certain paths and then selective unrolling [LQL12].

The performance of BCVERIFIER depends on the unroll count, and how much aliasing is involved in the example. Examples without complex aliasing are usually verified within a few seconds, whereas more elaborate examples can take up to a minute. We believe, however, that encoding aliasing properties in a smarter way (e.g., using ownership techniques) and using static analyses that over-approximate reachability of program paths can vastly improve the performance. Changes to BOOGIE itself over the course of the development of BCVERIFIER also improved the performance dramatically. For example, BOOGIE now interprets integer division and modulo, on which our BOOGIE model of stack slices relies.

The BCVERIFIER tool currently supports a limited subset of Java. In particular, arrays, floats, doubles, static fields and exceptions are not yet supported. Furthermore, the tool is unaware of the standard JDK library except that there is a class java.lang.Object which is at the root of the type hierarchy. The user feedback component of BCVERIFIER works as follows: when the verification process fails, then the verifier returns a path through the library implementations and the proof obligation that failed. To improve the quality of BCVERIFIER and gain confidence in its results, we use an automated suite of both positive and negative tests. We also use smoke tests, where the BOOGIE verifier searches for infeasible paths through the generated BOOGIE program, to detect an inconsistent axiomatization of our formal model.

A running instance of the web frontend of BCVERIFIER is available here [BCVb] and the code of BCVERIFIER here [BCVa].

8.3 Related Work

Of the works mentioned in Sections 5.2 and 6.3, only Leino and Yessenov present an embedding of their reasoning approach into a mechanized verification framework. While tackling the more general problem of stepwise refinement of class specifications into implementations, they impose a set of restrictions: (1) The refinement relation is established between single classes. (2) As refinements are described by sets of changes, the classes must share structural similarities in their method bodies. (3) Callbacks and more complex control flow relations are not considered.

In the following, we present work on program comparison techniques that were not specifically developed for the object-oriented setting. Relational Hoare Logic [Ben04] and Relational Separation Logic [Yan07] provide custom logics to reason about program equivalence. Currently, only simple imperative languages are considered, structurally similar programs assumed, and (automated) verifiers based on these logics do not exist. In the area of compiler optimizations, a lot of work has been done on proving intra-procedural transformations. Kundu, Tatlock, and Lerner [KTL09] employ Parameterized Equivalence Checking (PEC) to fully automatically verify equivalence of low-level program code using bisimulation relations. The PEC approach automatically infers a correlation relation (which, in our setting, would amount of automatically determining local places and invariants). Asynchronous steps (i.e., stalling places), are not covered by the approach.

Self-composition [BDR11] allows to describe information flow policies in terms of a safety property; the idea is to sequentially compose a program with a slightly modified version of itself and employ traditional verification to check whether equal inputs lead to equal outputs. For Java programs, Darvas, Hähnle, and Sands [DHS05] use a dynamic logic and the KeY tool, whereas Naumann [Nau06] employs the ESC/Java and Spec# tools to verify information flow properties. Leino and Müller [LM08] use self-composition to verify with Boogie that two executions of the same method yield equivalent results.

Barthe, Crespo, and Kunz [BCK11] provide more sophisticated verification conditions for equivalences in imperative programs using the notion of product programs that supports a direct reduction of relational verification to standard verification. The basic idea is to construct step-by-step a single program out of two programs. We believe this to be difficult in the object-oriented setting. First, the product program has to be represented in the language. As not every internally called method instance in one program needs to be represented by a single method instance in another program, stacks of different size would need to be merged. Another issue is dynamic dispatch, which would lead to a quadratic blow up as all possible combinations of invocations would need to be represented in the product program. Inspired by capabilities of their model, we represented their rule for asynchronous steps by the concept of stalling places.

Godlin and Strichman [GS12] prove equivalence of closely related C programs, which they call regression verification. They operate under a fixed notion of equivalence (functions with same input should emit same outputs) and use uninterpreted functions for recursive calls. More complex value relations than equivalence between internal function calls cannot be specified. Hawblitzel et al. [Haw+13] describe a contract mechanism called Mutual Summaries to modularly compare two procedural programs. The idea is to generalize single program contracts that describe the effect of a procedure to two programs by describing the relative effect of two procedures from different programs. To fit the concept of specifications that describe relative procedural effects, loops first need to be translated into tail-recursive procedures. Object-oriented features are not considered and the notion of equivalence needs to be defined by the programmer. By relating computational effects of internal parts of libraries, the Mutual Summaries approach can handle aspects that are not possible to describe using local invariants, like reordering of internal method calls. The concepts are currently being realized in the tool SymDiff [Lah+12], which also uses the automatic program verifier BOOGIE. At the moment, SymDiff only supports simple syntactic mappings to match procedures, globals, and constants of two programs. In closely related work [Lah+13], Lahiri et al. give a different approach (named differential assertion checking) to write relative specifications. This approach is amenable to automatic inference of relative specifications for pairs of procedures.

9 Conclusion

Life can only be understood backwards; but it must be lived forwards.

— Søren Kierkegaard

Backward compatibility for class libraries is an important problem to study. This thesis provides a formal model of class libraries and backward compatibility and integrates it in a mechanized approach. The last chapter of this thesis gives, in light of the presented material, a recapitulation of the contributions made (Section 9.1). Finally, it provides directions for future work and gives a general outlook (Section 9.2).

9.1 Contributions

The contributions made by this thesis are manyfold, the most prominent ones of which are:

- A formal treatment of source compatibility and its implications on language and program design.
- The development of a fully abstract trace-based semantics that captures the most essential properties of object-oriented class libraries.
- The development of a sound and complete reasoning approach for backward compatibility and the development of a specification language to support said reasoning approach.
- The implementation of the reasoning approach in an automatic verifier.

Although each contribution can be judged on its own merits individually, the combination of these puzzle pieces yields a powerful approach to tackle backward compatibility for class libraries.

Historical Background. Arnd Poetzsch-Heffter, Mathias Weber and Peter Zeller contributed directly to the goal of this thesis. To better detail the contributions made by them, I give an overview of how this thesis came about (to the best of my knowledge). My thesis advisor Arnd Poetzsch-Heffter provided the initial idea to develop a fully abstract semantics for packages. This was sparked by earlier work of Poetzsch-Heffter, Gaillourdet, and Schäfer to develop a fully abstract semantics for a new (dynamic) component model [PGS08].[1] The non-standard nature of the component model led to a complicated semantic model, which hindered practical uses. The goal of the subsequent work carried out by Poetzsch-Heffter and myself was to derive a useful practical model for a well-known notion of component. Hence, Java packages were chosen. Many versions of the semantics for various subsets of Java were developed over the years 2009–2011 until a practically relevant subset was identified [WP11]. Various forms of the semantics and notions of observation were chosen for the given language subset. In the end, a small subset was identified that contained many relevant aspects of Java and led itself to a full formalization. Later, I adapted these definitions to a setting with a differently formalized notion of observation (reachability of a certain state instead of termination) and used specialized simulations to prove the connection to the operational semantics. Meanwhile (2008–2010), I studied source compatibility aspects on a larger Java subset in more detail [WP10] and developed an implementation of a source compatibility checker for a substantial Java subset [SCWeb]. The Encapsulation Analyzer [SCWeb], used in Section 2.4.1, was developed as part of a Bachelor thesis by Mathias Weber that I supervised. The BCVerifier (2012–2013) was a joint implementation effort with Weber and Zeller which I coordinated. Based on my earlier results to manually encode the verification conditions in Boogie [WP12], Weber implemented most of the library implementation to Boogie translation as part of his Master thesis [Web12]. This sparked the initial ideas for a high-level Java-like specification language to describe coupling invariants, which was implemented by Zeller as a student assistant. Finally, to add a good user experience to the tool, I developed a web frontend for BCVerifier.

[1] My master's thesis was based on this component model [Wel08a; Wel08b]

9.2 Discussion and Outlook

The story does not end here, of course. There are many scenarios for backward compatibility of class libraries for which the presented approach does not provide satisfactory answers. In the following, I point out the limitations of the model and discuss various directions for future work.

One of the strong points of the presented theory of backward compatibility is that it allows the representation of the library implementations to change in complex ways. Small changes to library implementations (e.g., renaming a field) are currently, however, not well supported. Such changes at present require the library developer to state as part of the coupling invariant that most parts of the state remain indeed the same (in the form of an identity relation on the states modulo the correspondence relation). To overcome this limitation, it is necessary to either have better support at the theory or at the tool level (e.g., automatic inference of properties). To further enhance automation, a vast improvement would be to include in BCVERIFIER an automatic inference of single-program properties such as nullness or exposedness of fields. Ultimately, it would be useful to infer parts of the coupling invariants and establish local invariants. Here, the additional difficulty of local invariants over loop invariants is to find the program points that should be matched. A first way to automatically establish such specifications might be to derive them from recorded refactoring operations.

Although the presented reasoning method is, in theory, complete with respect to the formalized notion of backward compatibility, from a practical point of view, some coupling relations are very difficult (if not impossible) to express in a concise way. The presented specification language also has its limitations. Some specifications are not easily expressible in ISL, either because they become very large or ISL not powerful enough. ISL is solely based on invariants. This means that the properties can only be expressed in terms of invariants. This makes it very hard to reason about relative effects, for example the change of order of two commuting operations. If the control flow and data manipulation of related data of the library implementations do not correspond in a rather linear way, invariant-based specifications deter to specifications that capture the full functional behavior. For example, if one loop with n iterations doing two things becomes replaced by two consecutive loops with n iterations, coupling invariants are probably not the right tool (from a reasoning point of view). Coupling invariants correspond to class invariants in the single-program setting, lifted from the level of single classes to libraries, whereas local invariants can be seen as a generalization of loop invariants to two programs. Migrating more features from specification-based techniques in the single-program world to the world of two programs could also be extremely useful. Such features include pre-post specifications of methods, further specification-only types such as sets, lists and maps, and higher-level specification constructs for typically occurring relations. For example, this could contribute to an easy description of the relation between a loop and its corresponding tail-recursive method. On the downside, reasoning about effects of computation allows us to handle more scenarios but introduces a whole new level of complexity (specification of state transformers instead of states).

The conciseness of ISL specifications could also be improved. Based on my experience with the case studies, the specification code has about the size of the code that has been changed in the library. It can be further decreased by improving the specification language ISL. Currently, the syntax of ISL is quite verbose. Places are textually defined and invariants textually attached to certain places. With better tool support, places and invariants could be directly inserted into the code or attached like break-points for debugging (e.g., in a verification IDE). I expect that automation techniques can be developed to find coupling invariants for the code parts with no changes or the parts with simple syntactic changes (e.g., by tracking refactoring steps). Having a formal model for (a subset of) ISL would also provide a strong case for examining further higher-level specification constructs.

Another aspect not covered in this thesis are more complex information hiding policies. This brings in specifications (that need to specify such policies) or good defaults for such policies (often found in various ownership disciplines). Sometimes we are not interested in the equivalence of the full behavior of two library implementations but only under a restricted set of contexts or a subset of the API. For example, we might be interested in checking that the new version of a library has the same behavior as the old one with respect to a subset of its interface methods. Here BCVerifier could be enhanced to consider only a marked subset of methods as entry points (e.g., as in the case of the Eclipse PDE API Tools [EclPDE]). Currently this can be regulated by using appropriate access modifiers. It remains for future work to investigate other more complex restrictions on contexts and to extend the specification and verification technique to handle these restrictions. Other weaker notions of compatibility might be useful, e.g., those that do not check that returned values are related/equal, but only check that the new implementation does not crash when the old implementation shows behavior. This opens up the possibility for the user to specify a broader relation between values that appear as input/output, e.g., a relation on references that is not necessarily a bijection.

To make the approach usable in practice, additional language features must be considered. Concurrency and exceptions are having especially big impacts on the existing model. Another important aspect is to introduce modularity in the reasoning approach to ensure scalability of the approach. Possibilities to achieve this are structural disciplines (e.g., [BN05b]), special logics (e.g., separation logic [ORY01] or region logic [BNR08]), or other more ad-hoc approaches (e.g., [Coh+10]).

Bringing formal verification tools to the mainstream is a crucial challenge for the twenty-first century. I hope that this thesis contributes to the challenge and serves as incentive to other researchers to leverage the advancements in automatic program verification to construct similar (and more powerful) tools in the setting of backward compatibility verification.

Bibliography

[Ábr+04] Erika Ábrahám, Marcello M. Bonsangue, Frank S. de Boer, and Martin Steffen. “Object Connectivity and Full Abstraction for a Concurrent Calculus of Classes”. In: *ICTAC*. Ed. by Zhiming Liu and Keijiro Araki. Vol. 3407. Lecture Notes in Computer Science. Springer, 2004, pp. 37–51 (cit. on pp. 10, 64, 67).

[ADR09] Amal Ahmed, Derek Dreyer, and Andreas Rossberg. “State-dependent representation independence”. In: *POPL*. Ed. by Zhong Shao and Benjamin C. Pierce. ACM, 2009, pp. 340–353 (cit. on pp. 101, 108, 109).

[Anc+05] Davide Ancona, Ferruccio Damiani, Sophia Drossopoulou, and Elena Zucca. “Polymorphic bytecode: compositional compilation for Java-like languages”. In: *POPL*. Ed. by Jens Palsberg and Martín Abadi. ACM, 2005, pp. 26–37 (cit. on p. 37).

[AZ01] Davide Ancona and Elena Zucca. “True Modules for Java-like Languages”. In: *ECOOP*. Ed. by Jørgen Lindskov Knudsen. Vol. 2072. Lecture Notes in Computer Science. Springer, 2001, pp. 354–380 (cit. on p. 37).

[AZ04] Davide Ancona and Elena Zucca. “Principal typings for Java-like languages”. In: *POPL*. Ed. by Neil D. Jones and Xavier Leroy. ACM, 2004, pp. 306–317 (cit. on p. 37).

[Bar+05] Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. “Boogie: A Modular Reusable Verifier for Object-Oriented Programs”. In: *FMCO*. Ed. by Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever. Vol. 4111. Lecture Notes in Computer Science. Springer, 2005, pp. 364–387 (cit. on p. 131).

[Bar+06] Gilles Barthe, Lilian Burdy, Julien Charles, Benjamin Grégoire, Marieke Huisman, Jean-Louis Lanet, Mariela Pavlova, and Antoine Requet. “JACK - A Tool for Validation of Security and Behaviour of Java Applications”. In: *FMCO*. Ed. by Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever. Vol. 4709. Lecture Notes in Computer Science. Springer, 2006, pp. 152–174 (cit. on p. 2).

[BAW98] Ralph-Johan J. Back, Abo Akademi, and J. Von Wright. *Refinement Calculus: A Systematic Introduction*. Springer, Heidelberg, 1998 (cit. on p. 85).

[BCK11] Gilles Barthe, Juan Manuel Crespo, and César Kunz. “Relational Verification Using Product Programs”. In: *FM*. Ed. by Michael Butler and Wolfram Schulte. Vol. 6664. Lecture Notes in Computer Science. Springer, 2011, pp. 200–214 (cit. on pp. 111, 133).

[BCVa] BCVerifier Code Repository. `https://softech.cs.uni-kl.de/hg/public/bcverifier` (cit. on pp. 10, 132).

[BCVb] BCVerifier Web Frontend. `https://softech.cs.uni-kl.de/bcverifier` (cit. on pp. 10, 132).

[BDR11] Gilles Barthe, Pedro R. D’Argenio, and Tamara Rezk. “Secure information flow by self-composition”. In: 21.6 (2011), pp. 1207–1252 (cit. on p. 133).

[Ben04] Nick Benton. “Simple relational correctness proofs for static analyses and program transformations”. In: *POPL*. Ed. by Neil D. Jones and Xavier Leroy. ACM, 2004, pp. 14–25 (cit. on p. 133).

[BG05] Joshua Bloch and Neal Gafter. *Java Puzzlers: Traps, Pitfalls, and Corner Cases*. Addison-Wesley Professional, 2005 (cit. on p. 32).

[BGP01] Marina Biberstein, Joseph Gil, and Sara Porat. “Sealing, Encapsulation, and Mutability”. In: *ECOOP*. Ed. by Jørgen Lindskov Knudsen. Vol. 2072. Lecture Notes in Computer Science. Springer, 2001, pp. 28–52 (cit. on p. 33).

[BHS07] Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt, eds. *Verification of Object-Oriented Software: The KeY Approach*. LNCS 4334. Springer-Verlag, 2007 (cit. on p. 2).

[BJ01] Joachim van den Berg and Bart Jacobs. “The LOOP Compiler for Java and JML”. In: *TACAS*. Ed. by Tiziana Margaria and Wang Yi. Vol. 2031. Lecture Notes in Computer Science. Springer, 2001, pp. 299–312 (cit. on p. 2).

[BLS05] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. “The Spec# Programming System: An Overview”. In: vol. 3362. Lecture Notes in Computer Science. Springer-Verlag, 2005, pp. 49–69 (cit. on pp. 2, 131).

[BMW00] Ralph-Johan Back, Anna Mikhajlova, and Joakim von Wright. “Class Refinement as Semantics of Correct Object Substitutability”. In: 12.1 (2000), pp. 18–40 (cit. on p. 91).

[BN05a] Anindya Banerjee and David A. Naumann. “Ownership confinement ensures representation independence for object-oriented programs”. In: 52.6 (2005), pp. 894–960 (cit. on pp. 8, 9, 90, 101, 106, 116, 117).

[BN05b] Anindya Banerjee and David A. Naumann. “State Based Ownership, Reentrance, and Encapsulation”. In: *ECOOP*. Ed. by Andrew P. Black. Vol. 3586. Lecture Notes in Computer Science. Springer, 2005, pp. 387–411 (cit. on pp. 91, 139).

[BNR08] Anindya Banerjee, David A. Naumann, and Stan Rosenberg. “Regional Logic for Local Reasoning about Global Invariants”. In: *ECOOP*. Ed. by Jan Vitek. Vol. 5142. Lecture Notes in Computer Science. Springer, 2008, pp. 387–411 (cit. on p. 139).

[BTF05] Ittai Balaban, Frank Tip, and Robert M. Fuhrer. “Refactoring support for class library migration”. In: *OOPSLA*. Ed. by Ralph E. Johnson and Richard P. Gabriel. ACM, 2005, pp. 265–279 (cit. on p. 35).

[Buc10] Alex Buckley. *JSR 294 and Module Systems*. `http://blogs.oracle.com/abuckley/entry/jsr_294_and_module_systems`. Jan. 2010 (cit. on p. 24).

[Bur+03] Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joseph Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. “An overview of JML tools and applications”. In: 80 (2003), pp. 75–91 (cit. on p. 106).

[Cha+05] Patrice Chalin, Joseph R. Kiniry, Gary T. Leavens, and Erik Poll. "Beyond Assertions: Advanced Specification and Verification with JML and ESC/Java2". In: *FMCO*. Ed. by Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever. Vol. 4111. Lecture Notes in Computer Science. Springer, 2005, pp. 342–363 (cit. on p. 105).

[CN96] Kingsum Chow and David Notkin. "Semi-automatic update of applications in response to library changes". In: *ICSM*. IEEE Computer Society, 1996, pp. 359– (cit. on p. 35).

[Coh+10] Ernie Cohen, Michal Moskal, Wolfram Schulte, and Stephan Tobies. "Local Verification of Global Invariants in Concurrent Programs". In: *CAV*. Ed. by Tayssir Touili, Byron Cook, and Paul Jackson. Vol. 6174. Lecture Notes in Computer Science. Springer, 2010, pp. 480–494 (cit. on p. 139).

[Coo89] William R. Cook. *A Denotational Semantics of Inheritance*. 1989 (cit. on p. 90).

[Cor+03] John Corwin, David F. Bacon, David Grove, and Chet Murthy. "MJ: a rational module system for Java and its applications". In: *OOPSLA*. Ed. by Ron Crocker and Guy L. Steele Jr. ACM, 2003, pp. 241–254 (cit. on p. 37).

[CPN98] David G. Clarke, John Potter, and James Noble. "Ownership Types for Flexible Alias Protection". In: *OOPSLA*. Ed. by Bjørn N. Freeman-Benson and Craig Chambers. ACM, 1998, pp. 48–64 (cit. on pp. 91, 104).

[Dar08] Joseph D. Darcy. *Kinds of Compatibility*. `http://blogs.oracle.com/darcy/entry/kinds_of_compatibility`. Aug. 2008 (cit. on pp. 15, 32).

[DHS05] Ádám Darvas, Reiner Hähnle, and David Sands. "A Theorem Proving Approach to Analysis of Secure Information Flow". In: *SPC*. Ed. by Dieter Hutter and Markus Ullmann. Vol. 3450. Lecture Notes in Computer Science. Springer, 2005, pp. 193–209 (cit. on p. 133).

[Dig+08] Danny Dig, Stas Negara, Vibhu Mohindra, and Ralph E. Johnson. "*ReBA*: *re*factoring-aware *b*inary *a*daptation of evolving libraries". In: *ICSE*. Ed. by Wilhelm Schäfer, Matthew B. Dwyer, and Volker Gruhn. ACM, 2008, pp. 441–450 (cit. on p. 35).

[DJ06] Danny Dig and Ralph E. Johnson. "How do APIs evolve? A story of refactoring". In: 18.2 (2006), pp. 83–107 (cit. on pp. 1, 33).

[Dmi02] Mikhail Dmitriev. "Language-specific make technology for the Java programming language". In: *OOPSLA*. Ed. by Mamdouh Ibrahim and Satoshi Matsuoka. ACM, 2002, pp. 373–385 (cit. on p. 36).

[DPW12] Ferruccio Damiani, Arnd Poetzsch-Heffter, and Yannick Welsch. "A type system for checking specialization of packages in object-oriented programming". In: *SAC*. Ed. by Sascha Ossowski and Paola Lecca. ACM, 2012, pp. 1737–1742 (cit. on pp. 2, 25, 35, 162).

[Dro+08] Sophia Drossopoulou, Adrian Francalanza, Peter Müller, and Alexander J. Summers. "A Unified Framework for Verification Techniques for Object Invariants". In: *ECOOP*. Ed. by Jan Vitek. Vol. 5142. Lecture Notes in Computer Science. Springer, 2008, pp. 412–437 (cit. on p. 5).

[DWE98] Sophia Drossopoulou, David Wragg, and Susan Eisenbach. "What is Java Binary Compatibility?" In: *OOPSLA*. Ed. by Bjørn N. Freeman-Benson and Craig Chambers. ACM, 1998, pp. 341–361 (cit. on p. 37).

[Ecl12] Eclipse Naming Conventions. `http://wiki.eclipse.org/Naming_Conventions`. Apr. 2012 (cit. on pp. 32, 33).

[EclPDE] Eclipse PDE API Tools. `http://www.eclipse.org/pde/pde-api-tools/` (cit. on pp. 1, 14, 34, 138).

[ECM06] ECMA. *C# Language Specification (Standard ECMA-334, 4th edition)*. `http://www.ecma-international.org/publications/standards/Ecma-334.htm`. June 2006 (cit. on pp. 22, 30).

[Fil+10a] Ivana Filipovic, Peter W. O'Hearn, Noam Rinetzky, and Hongseok Yang. "Abstraction for concurrent objects". In: 411.51-52 (2010), pp. 4379–4398 (cit. on p. 67).

[Fil+10b] Ivana Filipovic, Peter W. O'Hearn, Noah Torp-Smith, and Hongseok Yang. "Blaming the client: on data refinement in the presence of pointers". In: 22.5 (2010), pp. 547–583 (cit. on pp. 91, 92).

[FKF99] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. "A Programmer's Reduction Semantics for Classes and Mixins". In: *Formal Syntax and Semantics of Java*. Ed. by Jim Alves-Foss. Vol. 1523. Lecture Notes in Computer Science. Springer, 1999, pp. 241–269 (cit. on pp. 7, 41).

[Fla+02] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. "Extended Static Checking for Java". In: *PLDI*. Ed. by Jens Knoop and Laurie J. Hendren. ACM, 2002, pp. 234–245 (cit. on p. 2).

[For+95] Ira R. Forman, Michael H. Conner, Scott Danforth, and Larry K. Raper. "Release-to-Release Binary Compatibility in SOM". In: *OOPSLA*. Ed. by Rebecca Wirfs-Brock. ACM, 1995, pp. 426–438 (cit. on p. 37).

[Fre06] Tammo Freese. "Refactoring-aware version control". In: *ICSE*. Ed. by Leon J. Osterweil, H. Dieter Rombach, and Mary Lou Soffa. ACM, 2006, pp. 953–956 (cit. on p. 35).

[Gam+95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995 (cit. on pp. 100, 116).

[Goe11] Brian Goetz. *Interface evolution via virtual extension methods*. `http://cr.openjdk.java.net/~briangoetz/lambda/Defender Methods v4.pdf`. June 2011 (cit. on p. 33).

[Gos+05] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Third Edition*. The Java Series. Addison-Wesley, 2005 (cit. on pp. 15–17, 24, 32–34, 36, 37).

[GP11] Kathrin Geilmann and Arnd Poetzsch-Heffter. "Modular Checking of Confinement for Object-Oriented Components using Abstract Interpretation". In: *International Workshop on Aliasing, Confinement and Ownership*. 2011 (cit. on p. 90).

[GPV01] Christian Grothoff, Jens Palsberg, and Jan Vitek. "Encapsulating Objects with Confined Types". In: *OOPSLA*. Ed. by Linda M. Northrop and John M. Vlissides. ACM, 2001, pp. 241–253 (cit. on pp. 7, 13, 17).

[GS12] Benny Godlin and Ofer Strichman. "Regression Verification: Proving the Equivalence of Similar Programs". In: (2012) (cit. on p. 134).

[GY11] Alexey Gotsman and Hongseok Yang. “Liveness-Preserving Atomicity Abstraction”. In: *ICALP (2)*. Ed. by Luca Aceto, Monika Henzinger, and Jiri Sgall. Vol. 6756. Lecture Notes in Computer Science. Springer, 2011, pp. 453–465 (cit. on p. 67).

[Haw+13] Chris Hawblitzel, Ming Kawaguchi, Shuvendu K. Lahiri, and Henrique Rebêlo. “Towards Modularly Comparing Programs Using Automated Theorem Provers”. In: *CADE*. Ed. by Maria Paola Bonacina. Vol. 7898. Lecture Notes in Computer Science. Springer, 2013, pp. 282–299 (cit. on p. 134).

[HD05] Johannes Henkel and Amer Diwan. “CatchUp!: capturing and replaying refactorings to support API evolution”. In: *ICSE*. Ed. by Gruia-Catalin Roman, William G. Griswold, and Bashar Nuseibeh. ACM, 2005, pp. 274–283 (cit. on p. 35).

[Hen88] Matthew Hennessy. *Algebraic theory of processes*. MIT Press series in the foundations of computing. MIT Press, 1988, pp. I–VI, 1–270 (cit. on pp. 51, 56).

[Her+07] Sebastian Herold et al. “CoCoME - The Common Component Modeling Example”. In: *CoCoME*. Ed. by Andreas Rausch, Ralf Reussner, Raffaela Mirandola, and Frantisek Plasil. Vol. 5153. Lecture Notes in Computer Science. Springer, 2007, pp. 16–53 (cit. on p. 163).

[HM80] Matthew Hennessy and Robin Milner. “On Observing Nondeterminism and Concurrency”. In: *ICALP*. Ed. by J. W. de Bakker and Jan van Leeuwen. Vol. 85. Lecture Notes in Computer Science. Springer, 1980, pp. 299–309 (cit. on p. 92).

[Hoa72] C. A. R. Hoare. “Proof of Correctness of Data Representations”. In: 1 (1972), pp. 271–281 (cit. on p. 85).

[IPW01] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. “Featherweight Java: a minimal core calculus for Java and GJ”. In: 23.3 (2001), pp. 396–450 (cit. on p. 41).

[Jac+11] Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. “VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java”. In: *NASA Formal Methods*. Ed. by Mihaela Gheorghiu Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi. Vol. 6617. Lecture Notes in Computer Science. Springer, 2011, pp. 41–55 (cit. on p. 2).

[Jigsaw] Project Jigsaw. http://openjdk.java.net/projects/jigsaw/ (cit. on p. 38).

[JR05a] Alan Jeffrey and Julian Rathke. "A fully abstract may testing semantics for concurrent objects". In: 338.1-3 (2005), pp. 17–63 (cit. on p. 51).

[JR05b] Alan Jeffrey and Julian Rathke. "Java Jr: Fully Abstract Trace Semantics for a Core Java Language". In: *ESOP*. Ed. by Shmuel Sagiv. Vol. 3444. Lecture Notes in Computer Science. Springer, 2005, pp. 423–438 (cit. on pp. 7, 10, 36, 64, 66, 67).

[JSR277] JSR 277: Java Module System. http://jcp.org/en/jsr/detail?id=277. Nov. 2006 (cit. on p. 37).

[JSR294] JSR 294: Improved Modularity Support in the Java Programming Language. http://jcp.org/en/jsr/detail?id=294 (cit. on p. 38).

[KPW10] Ilham W. Kurnia, Arnd Poetzsch-Heffter, and Yannick Welsch. "State-based Object Models Are More Abstract Than Trace-based Models: Towards a Unified Specification Framework". In: *Technical Report No. 2010-13*. 2010-13. Karlsruhe, June 2010, pp. 268–282 (cit. on p. 163).

[KTL09] Sudipta Kundu, Zachary Tatlock, and Sorin Lerner. "Proving optimizations correct using parameterized program equivalence". In: *PLDI*. Ed. by Michael Hind and Amer Diwan. ACM, 2009, pp. 327–337 (cit. on p. 133).

[KW06] Vasileios Koutavas and Mitchell Wand. "Bisimulations for Untyped Imperative Objects". In: *ESOP*. Ed. by Peter Sestoft. Vol. 3924. Lecture Notes in Computer Science. Springer, 2006, pp. 146–161 (cit. on p. 92).

[KW07] Vasileios Koutavas and Mitchell Wand. "Reasoning About Class Behavior". In: *Informal Workshop Record of FOOL*. 2007 (cit. on pp. 8, 36, 92).

[Lag04] Giovanni Lagorio. "Capturing ghost dependencies in Java sources". In: 3.11 (2004), pp. 77–96 (cit. on p. 37).

[Lah+12] Shuvendu K. Lahiri, Chris Hawblitzel, Ming Kawaguchi, and Henrique Rebêlo. “SYMDIFF: A Language-Agnostic Semantic Diff Tool for Imperative Programs”. In: *CAV*. Ed. by P. Madhusudan and Sanjit A. Seshia. Vol. 7358. Lecture Notes in Computer Science. Springer, 2012, pp. 712–717 (cit. on p. 134).

[Lah+13] Shuvendu K. Lahiri, Kenneth L. McMillan, Rahul Sharma, and Chris Hawblitzel. “Differential assertion checking”. In: *ESEC/SIGSOFT FSE*. Ed. by Bertrand Meyer, Luciano Baresi, and Mira Mezini. ACM, 2013, pp. 345–355 (cit. on p. 134).

[Lam12] Leslie Lamport. “How to write a 21st century proof”. In: 11 (1 2012), pp. 43–63 (cit. on p. 16).

[Lei08] K. Rustan M. Leino. *This is Boogie 2. Manuscript KRML 178*. Available at `http://research.microsoft.com/~leino/papers.html`. 2008 (cit. on p. 131).

[Lei10] K. Rustan M. Leino. “Dafny: An Automatic Program Verifier for Functional Correctness”. In: *LPAR (Dakar)*. Ed. by Edmund M. Clarke and Andrei Voronkov. Vol. 6355. Lecture Notes in Computer Science. Springer, 2010, pp. 348–370 (cit. on p. 131).

[Lie+10] Michael Lienhardt, Ivan Lanese, Mario Bravetti, Davide Sangiorgi, Gianluigi Zavattaro, Yannick Welsch, Jan Schäfer, and Arnd Poetzsch-Heffter. “A Component Model for the ABS Language”. In: *FMCO*. Ed. by Bernhard K. Aichernig, Frank S. de Boer, and Marcello M. Bonsangue. Vol. 6957. Lecture Notes in Computer Science. Springer, 2010, pp. 165–183 (cit. on p. 163).

[LM04] K. Rustan M. Leino and Peter Müller. “Object Invariants in Dynamic Contexts”. In: *ECOOP*. Ed. by Martin Odersky. Vol. 3086. Lecture Notes in Computer Science. Springer, 2004, pp. 491–516 (cit. on p. 91).

[LM08] K. Rustan M. Leino and Peter Müller. “Verification of Equivalent-Results Methods”. In: *ESOP*. Ed. by Sophia Drossopoulou. Vol. 4960. Lecture Notes in Computer Science. Springer, 2008, pp. 307–321 (cit. on p. 133).

[LMS09] K. Rustan M. Leino, Peter Müller, and Jan Smans. “Verification of Concurrent Programs with Chalice”. In: *FOSAD*. Ed. by Alessandro Aldini, Gilles Barthe, and Roberto Gorrieri. Vol. 5705. Lecture Notes in Computer Science. Springer, 2009, pp. 195–222 (cit. on p. 131).

[LQL12] Akash Lal, Shaz Qadeer, and Shuvendu K. Lahiri. “A Solver for Reachability Modulo Theories”. In: *CAV*. Ed. by P. Madhusudan and Sanjit A. Seshia. Vol. 7358. Lecture Notes in Computer Science. Springer, 2012, pp. 427–443 (cit. on p. 132).

[LY12] K. Rustan M. Leino and Kuat Yessenov. “Stepwise refinement of heap-manipulating code in Chalice”. In: 24.4-6 (2012), pp. 519–535 (cit. on pp. 92, 113, 133).

[MB08] Leonardo Mendonça de Moura and Nikolaj Bjørner. “Z3: An Efficient SMT Solver”. In: *TACAS*. Ed. by C. R. Ramakrishnan and Jakob Rehof. Vol. 4963. Lecture Notes in Computer Science. Springer, 2008, pp. 337–340 (cit. on p. 131).

[Mey97] Bertrand Meyer. *Object-Oriented Software Construction*. Second edition. Prentice-Hall, 1997 (cit. on p. 5).

[MFH01] Sean McDirmid, Matthew Flatt, and Wilson C. Hsieh. “Jiazzi: New-Age Components for Old-Fashioned Java”. In: *OOPSLA*. Ed. by Linda M. Northrop and John M. Vlissides. ACM, 2001, pp. 211–222 (cit. on p. 37).

[Mic99] Sun Microsystems. *Code Conventions for the Java Programming Language*. http://www.oracle.com/technetwork/java/codeconv-138413.html. Apr. 1999 (cit. on p. 33).

[Mil77] Robin Milner. “Fully Abstract Models of Typed *lambda*-Calculi”. In: 4.1 (1977), pp. 1–22 (cit. on pp. 1, 5).

[Mor68] James H. Morris. *Lambda-Calculus Models of Programming Languages*. 1968 (cit. on p. 51).

[Mor94] Carroll C. Morgan. *Programming from specifications, 2nd Edition*. Prentice Hall International series in computer science. Prentice Hall, 1994, pp. I–XV, 1–332 (cit. on pp. 85, 91).

[MP00] Jörg Meyer and Arnd Poetzsch-Heffter. “An Architecture for Interactive Program Provers”. In: *TACAS*. Ed. by Susanne Graf and Michael I. Schwartzbach. Vol. 1785. Lecture Notes in Computer Science. Springer, 2000, pp. 63–77 (cit. on p. 2).

[MP98] Peter Müller and Arnd Poetzsch-Heffter. “Kapselung und Methodenbindung: Javas Designprobleme und ihre Korrektur”. In: *Java-Informations-Tage*. 1998, pp. 1–10 (cit. on p. 38).

[MPL06] Peter Müller, Arnd Poetzsch-Heffter, and Gary T. Leavens. “Modular invariants for layered object structures”. In: 62.3 (2006), pp. 253–286 (cit. on p. 5).

[MPU04] Claude Marché, Christine Paulin-Mohring, and Xavier Urbain. “The KRAKATOA tool for certificationof JAVA/JAVACARD programs annotated in JML”. In: 58.1-2 (2004), pp. 89–106 (cit. on p. 2).

[MS88] Albert R. Meyer and Kurt Sieber. “Towards Fully Abstract Semantics for Local Variables”. In: *POPL*. Ed. by Jeanne Ferrante and P. Mager. ACM Press, 1988, pp. 191–203 (cit. on p. 106).

[MS97] Anna Mikhajlova and Emil Sekerinski. “Class Refinement and Interface Refinement in Object-Oriented Programs”. In: *FME*. Ed. by John S. Fitzgerald, Cliff B. Jones, and Peter Lucas. Vol. 1313. Lecture Notes in Computer Science. Springer, 1997, pp. 82–101 (cit. on p. 91).

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *Definition of standard ML*. MIT Press, 1990, pp. I–XI, 1–101 (cit. on p. 35).

[Nau06] David A. Naumann. “From Coupling Relations to Mated Invariants for Checking Information Flow”. In: *ESORICS*. Ed. by Dieter Gollmann, Jan Meier, and Andrei Sabelfeld. Vol. 4189. Lecture Notes in Computer Science. Springer, 2006, pp. 279–296 (cit. on p. 133).

[Net05] NetBeans Code Conventions. `http://netbeans.org/community/guidelines/code-conventions.html`. July 2005 (cit. on p. 33).

[NSS12] David A. Naumann, Augusto Sampaio, and Leila Silva. “Refactoring and representation independence for class hierarchies”. In: 433 (2012), pp. 60–97 (cit. on p. 90).

[ORY01] Peter W. O’Hearn, John C. Reynolds, and Hongseok Yang. “Local Reasoning about Programs that Alter Data Structures”. In: *CSL*. Ed. by Laurent Fribourg. Vol. 2142. Lecture Notes in Computer Science. Springer, 2001, pp. 1–19 (cit. on p. 139).

[OSGI] OSGi Service Platform. `http://www.osgi.org/` (cit. on pp. 24, 37, 38).

[PGS08] Arnd Poetzsch-Heffter, Jean-Marie Gaillourdet, and Jan Schäfer. “Towards a Fully Abstract Semantics for Object-Oriented Program Components”. July 2008 (cit. on p. 136).

[Plo77] Gordon D. Plotkin. “LCF Considered as a Programming Language”. In: 5.3 (1977), pp. 223–255 (cit. on pp. 1, 5).

[Poe+12] Arnd Poetzsch-Heffter, Christoph Feller, Ilham W. Kurnia, and Yannick Welsch. “Model-Based Compatibility Checking of System Modifications”. In: *ISoLA (1)*. Ed. by Tiziana Margaria and Bernhard Steffen. Vol. 7609. Lecture Notes in Computer Science. Springer, 2012, pp. 97–111 (cit. on pp. 2, 162).

[PS98] Andrew Pitts and Ian Stark. “Operational Reasoning for Functions with Local State”. In: *Higher Order Operational Techniques in Semantics*. Ed. by Andrew Gordon and Andrew Pitts. Publications of the Newton Institute, Cambridge University Press, 1998, pp. 227–273 (cit. on p. 109).

[Riv07] Jim des Rivières. *Evolving Java-based APIs*. `http://wiki.eclipse.org/Evolving_Java-based_APIs`. Oct. 2007 (cit. on pp. 1, 14, 34).

[Sch04] Norbert Schirmer. “Analysing the Java package/access concepts in Isabelle/HOL”. In: 16.7 (2004), pp. 689–706 (cit. on p. 38).

[SCWeb] Source Compatibility for Java Packages Website. `http://softech.cs.uni-kl.de/~scomp` (cit. on pp. 32, 136).

[SEM08] Max Schäfer, Torbjörn Ekman, and Oege de Moor. “Sound and extensible renaming for java”. In: *OOPSLA*. Ed. by Gail E. Harris. ACM, 2008, pp. 277–294 (cit. on p. 35).

[SKS07] Davide Sangiorgi, Naoki Kobayashi, and Eijiro Sumii. “Environmental Bisimulations for Higher-Order Languages”. In: *LICS*. IEEE Computer Society, 2007, pp. 293–302 (cit. on p. 92).

[SP07a] Eijiro Sumii and Benjamin C. Pierce. “A bisimulation for dynamic sealing”. In: 375.1-3 (2007), pp. 169–192 (cit. on p. 92).

[SP07b] Eijiro Sumii and Benjamin C. Pierce. “A bisimulation for type abstraction and recursion”. In: 54.5 (2007) (cit. on p. 92).

[SSP07] Rok Strnisa, Peter Sewell, and Matthew J. Parkinson. “The java module system: core design and semantic definition”. In: *OOPSLA*. Ed. by Richard P. Gabriel, David F. Bacon, Cristina Videira Lopes, and Guy L. Steele Jr. ACM, 2007, pp. 499–514 (cit. on p. 37).

[Ste06] Martin Steffen. “Object-Connectivity and Observability for Class-Based, Object-Oriented Languages”. Habilitation thesis. July 2006 (cit. on pp. 51, 67).

[WD96] Jim Woodcock and Jim Davies. *Using Z - Specification, Refinement, and Proof*. Prentice Hall, 1996 (cit. on pp. 66, 91).

[Web12] Mathias Weber. *Generating Boogie Verification Conditions for Backward Compatibility of Class Libraries*. Available at `http://softech.cs.uni-kl.de/pub?id=191`. Oct. 2012 (cit. on pp. 131, 136).

[Wel+13] Yannick Welsch, Mathias Weber, Peter Zeller, and Arnd Poetzsch-Heffter. "A Backward Compatibility Verifier for Java Libraries". Internal report, available at `https://softech.informatik.uni-kl.de/twiki/pub/Homepage/YannickWelsch/bcverifier.pdf`. 2013 (cit. on pp. 2, 163).

[Wel08a] Yannick Welsch. *Grey-Box Specification and Runtime Testing of Object-Oriented Program Components*. May 2008 (cit. on p. 136).

[Wel08b] Yannick Welsch. "Grey-box specifications for object-oriented program components". In: *OOPSLA Companion*. Ed. by Gail E. Harris. ACM, 2008, pp. 913–914 (cit. on pp. 136, 163).

[WF94] Andrew K. Wright and Matthias Felleisen. "A Syntactic Approach to Type Soundness". In: 115.1 (1994), pp. 38–94 (cit. on pp. 42, 50).

[WP10] Yannick Welsch and Arnd Poetzsch-Heffter. "Source Compatibility for Java Packages". Internal report, available at `https://softech.informatik.uni-kl.de/twiki/pub/Homepage/Publikationen/WelschPoetzschHeffter10Comp.pdf`. 2010 (cit. on pp. 8, 14, 15, 25, 136, 163).

[WP11] Yannick Welsch and Arnd Poetzsch-Heffter. "Full Abstraction at Package Boundaries of Object-Oriented Languages". In: *SBMF*. Ed. by Adenilso da Silva Simão and Carroll Morgan. Vol. 7021. Lecture Notes in Computer Science. Springer, 2011, pp. 28–43 (cit. on pp. 2, 56, 61, 136, 162).

[WP12] Yannick Welsch and Arnd Poetzsch-Heffter. "Verifying backwards compatibility of object-oriented libraries using Boogie". In: *Proceedings of the 14th Workshop on Formal Techniques for Java-like Programs*. FTfJP '12. ACM, 2012, pp. 35–41 (cit. on pp. 2, 136, 162).

[WP13] Yannick Welsch and Arnd Poetzsch-Heffter. “A Fully Abstract Trace-based Semantics for Reasoning About Backward Compatibility of Class Libraries”. In: Science of Computer Programming. To appear. Elsevier, 2013 (cit. on pp. 2, 162).

[WS11] Yannick Welsch and Jan Schäfer. “Location Types for Safe Distributed Object-Oriented Programming”. In: *TOOLS (49)*. Ed. by Judith Bishop and Antonio Vallecillo. Vol. 6705. Lecture Notes in Computer Science. Springer, 2011, pp. 194–210 (cit. on p. 162).

[WSP13] Yannick Welsch, Jan Schäfer, and Arnd Poetzsch-Heffter. “Location Types for Safe Programming with Near and Far References”. In: *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*. Ed. by Dave Clarke, James Noble, and Tobias Wrigstad. Vol. 7850. Lecture Notes in Computer Science. Springer, 2013, pp. 471–500 (cit. on p. 162).

[Yan07] Hongseok Yang. “Relational separation logic”. In: 375.1-3 (2007), pp. 308–334 (cit. on p. 133).

[Zen05] Matthias Zenger. “KERIS: evolving software with extensible modules”. In: 17.5 (2005), pp. 333–362 (cit. on p. 37).

List of Symbols

$\bullet$	Empty sequence or set	16
$\varnothing$	Empty map	16
$\_$	Place-holder for a free logical variable that occurs only once in the formula	16
$\bot$	Null type, to type the **null** expression	19
$<^{\mathbf{d}}_{X}$	Direct subtype relation on X	19
$<_X$	Transitive subtype relation (not reflexive)	19
$\leq_X$	Reflexive and transitive subtype relation	19
$\in_X$	Field or method membership relation	19
$\equiv^{\rho}$	Equivalence on terms modulo a renaming ρ	47
$\preccurlyeq^{\rho}_{L}$	Relation between program configurations if L parts are equivalent under ρ	68
$\overset{\gamma}{\rightsquigarrow}$	Small-step reduction relation	42
$\rightsquigarrow^{L}_{KX}$	Helper relation for τ steps local to L	43
$\xrightarrow{\overline{\gamma}}$	Large-step reduction relation	47
$\boxed{\downarrow}$	Input label	42
$\boxed{\uparrow}$	Output label	42
✓	Successful program run (makes observation)	52
✗	Unsuccessful program run (does not make observation)	52
$\lll^{\rho}$	Relation between deterministic program context and MGC	74
$\ggg^{\rho}$	Relation between MGC and constructed deterministic program context	79
c	Class name	17

$\mathcal{C}_X$	Set of (qualified) class names for which there is a declaration in X	19
D	Type declaration	16
E	Expression	16, 41, 51, 60
$\mathcal{E}$	Evaluation context	42
F	Field declaration	16
f	Field name	17
$\mathcal{F}$	Stack slice	41
$\mathcal{G}$	Field mapping	41
o	Object identifier	41
γ	Label	45, 51
Γ	Local variable typing	20
i	Interface name	17
$\mathcal{I}_X$	Set of (qualified) interface names for which there is a declaration in X	19
K	Codebase, usually denoting a program context	16, 18
L	Origin location	41
m	Method name	17
μ	Message	45
$M^{\mathbf{a}}$	Abstract method declaration	16
$M^{\mathbf{c}}$	Method declaration with body	16
$\mathcal{O}$	Heap (Object store)	41
o^{α}	Abstracted object	45
P	Package declaration, containing a list of type declarations	16
p	Package name	17
$\mathcal{P}_X$	Set of package names for which there is a package declaration in X	19

ρ	Renaming (bijective relation on object identifiers)	47
$\mathcal{S}$	Runtime configuration (of the form $KX, \mathcal{O}, \overline{\mathcal{F}}$)	41
$\mathcal{S}_{KX}^{\textbf{init}}$	Initial (startup) configuration	48
T	Shorthand to denote all kind of type names ($p.c \mid p.i \mid \bot$)	19
t	Class or interface name	16
T^{α}	Abstracted type	45
$\mathcal{T}_X^{\alpha}$	Set of abstracted types that can occur in traces with X	46
$\mathcal{T}_X$	Set of all types defined by X ($\mathcal{C}_X \cup \mathcal{I}_X \cup \{\mathsf{lang.Object}\} \cup \{\bot\}$)	19
τ	Silent transition label	43
V	Exposure flag	41
v	Reference value	41
v^{α}	Abstracted value	45
X	Codebase, usually denoting a library implementation	16, 18
x	Local variable or method parameter name	17
Y	Codebase, usually denoting a library implementation	16, 18

Index

About the Author

NAME:	Yannick Welsch
E-MAIL:	yannick@welsch.lu
WEB SITE:	http://www.welsch.lu

Education

- 2008-2013: Research Assistant at the Software Technology Group, University of Kaiserslautern, Germany.
- 2006-2008: Master of Science in Computer Science, University of Kaiserslautern, Germany, Thesis title: Grey-Box Specification and Runtime Testing of Object-Oriented Program Components.
- 2003-2006: Bachelor of Science in Computer Science, University of Kaiserslautern, Germany, Thesis title: Using Boxes in Java Programming: An evaluation.
- 2003: Diplôme de fin d'études secondaires, Lycée classique d'Echternach, Luxembourg, Section Latin et Langues Vivantes - Mathématiques, option Sciences Physiques.

Publications

A list of my (peer-reviewed) publications in reverse chronological order:

- Yannick Welsch and Arnd Poetzsch-Heffter. “A Fully Abstract Trace-based Semantics for Reasoning About Backward Compatibility of Class Libraries”. In: Science of Computer Programming. To appear. Elsevier, 2013
- Yannick Welsch, Jan Schäfer, and Arnd Poetzsch-Heffter. “Location Types for Safe Programming with Near and Far References”. In: *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*. Ed. by Dave Clarke, James Noble, and Tobias Wrigstad. Vol. 7850. Lecture Notes in Computer Science. Springer, 2013, pp. 471–500

  Invited contribution to LNCS State-of-the-Art Surveys volume.
- Yannick Welsch and Arnd Poetzsch-Heffter. “Verifying backwards compatibility of object-oriented libraries using Boogie”. In: *Proceedings of the 14th Workshop on Formal Techniques for Java-like Programs*. FTfJP ’12. ACM, 2012, pp. 35–41
- Arnd Poetzsch-Heffter, Christoph Feller, Ilham W. Kurnia, and Yannick Welsch. “Model-Based Compatibility Checking of System Modifications”. In: *ISoLA (1)*. Ed. by Tiziana Margaria and Bernhard Steffen. Vol. 7609. Lecture Notes in Computer Science. Springer, 2012, pp. 97–111
- Ferruccio Damiani, Arnd Poetzsch-Heffter, and Yannick Welsch. “A type system for checking specialization of packages in object-oriented programming”. In: *SAC*. ed. by Sascha Ossowski and Paola Lecca. ACM, 2012, pp. 1737–1742
- Yannick Welsch and Arnd Poetzsch-Heffter. “Full Abstraction at Package Boundaries of Object-Oriented Languages”. In: *SBMF*. ed. by Adenilso da Silva Simão and Carroll Morgan. Vol. 7021. Lecture Notes in Computer Science. Springer, 2011, pp. 28–43

  Received the best paper award at the SBMF conference.
- Yannick Welsch and Jan Schäfer. “Location Types for Safe Distributed Object-Oriented Programming”. In: *TOOLS (49)*. Ed. by Judith Bishop and Antonio Vallecillo. Vol. 6705. Lecture Notes in Computer Science. Springer, 2011, pp. 194–210

- Michael Lienhardt, Ivan Lanese, Mario Bravetti, Davide Sangiorgi, Gianluigi Zavattaro, Yannick Welsch, Jan Schäfer, and Arnd Poetzsch-Heffter. “A Component Model for the ABS Language”. In: *FMCO*. ed. by Bernhard K. Aichernig, Frank S. de Boer, and Marcello M. Bonsangue. Vol. 6957. Lecture Notes in Computer Science. Springer, 2010, pp. 165–183
- Yannick Welsch. “Grey-box specifications for object-oriented program components”. In: *OOPSLA Companion*. Ed. by Gail E. Harris. ACM, 2008, pp. 913–914
- Sebastian Herold, Holger Klus, Yannick Welsch, Constanze Deiters, Andreas Rausch, Ralf Reussner, Klaus Krogmann, Heiko Koziolek, Raffaela Mirandola, Benjamin Hummel, Michael Meisinger, and Christian Pfaller. “CoCoME - The Common Component Modeling Example”. In: *CoCoME*. ed. by Andreas Rausch, Ralf Reussner, Raffaela Mirandola, and Frantisek Plasil. Vol. 5153. Lecture Notes in Computer Science. Springer, 2007, pp. 16–53

Other relevant drafts (not subsumed by previously mentioned publications):

- Yannick Welsch, Mathias Weber, Peter Zeller, and Arnd Poetzsch-Heffter. “A Backward Compatibility Verifier for Java Libraries”. Internal report, available at `https://softech.informatik.uni-kl.de/twiki/pub/Homepage/YannickWelsch/bcverifier.pdf`. 2013
- Yannick Welsch and Arnd Poetzsch-Heffter. “Source Compatibility for Java Packages”. Internal report, available at `https://softech.informatik.uni-kl.de/twiki/pub/Homepage/Publikationen/WelschPoetzschHeffter10Comp.pdf`. 2010
- Ilham W. Kurnia, Arnd Poetzsch-Heffter, and Yannick Welsch. “State-based Object Models Are More Abstract Than Trace-based Models: Towards a Unified Specification Framework”. In: *Technical Report No. 2010-13*. 2010-13. Karlsruhe, June 2010, pp. 268–282