A Backward Compatibility Verifier for Java Libraries*

Yannick Welsch, Mathias Weber, Peter Zeller, and Arnd Poetzsch-Heffter

University of Kaiserslautern, Germany
`{welsch,m_weber,p_zeller,poetzsch}@cs.uni-kl.de`

Abstract. Proving that a library implementation is backward compatible with an older version can be challenging, as the internal representation of the library might completely change and the clients of the library are usually unknown. This is particularly difficult in the setting of object-oriented programs with complex heaps and callbacks. Mechanical verification is a key success factor to make such proofs feasible.
In this paper, we present a technique to verify backward compatibility or equivalence of class libraries. The technique works for complex implementations, with recursion and loops, in the setting of unknown program contexts. Reasoning about backward compatibility is done in terms of a special simulation relation between programs that use the old library implementation and programs that use the new library implementation. The verification process relies on a *coupling invariant* that describes this relation. We introduce a novel specification language to formulate coupling invariants and present the BCVERIFIER tool, first of its kind, that implements the verification approach using the automatic verifier BOOGIE. Our approach is validated by a number of classic examples.

1 Introduction

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 [14], 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 large libraries with many users, for example the standard class libraries that are part of the Java platform. The state of the practice in backward compatibility checking is that a small set of library developers use informal guidelines (e.g., [37]) and tools employing simple syntactic rules (e.g., [16]) to check aspects of backward compatibility. Although progress has been made in program comparison techniques in the recent years, practical tools that allow reasoning about backward compatibility in terms of full functional behavior of the libraries and that cover main-stream languages are not available yet.

Reasoning about the behavior of class library implementations is challenging as the number of possible contexts is infinite and contexts are complex. Furthermore, intricate object-oriented encapsulation mechanisms enable the program configurations between

* This research is partly funded by the EU project FP7-231620 HATS: Highly Adaptable and Trustworthy Software using Formal Models.

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

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

Fig. 1. Cell example

different library implementations to be significantly different. The verification approach in this paper builds on a sound and complete method for reasoning about backward compatibility of class libraries that we developed in recent work [41, 42]. Directly comparing library implementations is not only useful when no specifications are available; we conjecture, similar to Godlin and Strichman [19], that it may often be simpler to verify behavioral equivalence of two similar library implementations than to build a specification of the full behavior of the library and verify conformance to the specification. We further conjecture that, in the setting of class libraries, such verification tasks should be feasible by employing similar automated tools as in the case of specification conformance checking [5, 8, 17, 24]. To validate this claim, we developed a backward compatibility verifier, called BCVERIFIER, that targets the automatic program verifier BOOGIE [4].

To illustrate our goals, let us consider a very simple[1] library at the left of Fig. 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 Fig. 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. This property is called *backward compatibility*.

Verifying backward compatibility consists of ensuring that there is a special simulation relation between programs that use the old library implementation and programs that use 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. The verification process relies on a *coupling invariant* that describes this relation. The coupling invariant has to be provided by the user of the tool. The BCVERIFIER tool then checks that the relation induced by the coupling invariant is a proper simulation for the provided library implementations.

[1] A more elaborate example can be found in Fig. 2 at page 17.

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. We developed a specification language which allows to formally state this property as a coupling invariant:

```
invariant forall old Cell o1, new Cell o2 :: o1 ~ o2 ==> if o2.f then o1.c ~ o2.c1 else o1.c ~ o2.c2;
```

The invariant specifies that in properly simulated observable program states, the aforementioned property holds 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.

Contributions. In an earlier workshop paper [43], we studied the feasibility of using the automated verifier Boogie to encode our verification conditions. This was evaluated by translating the verification conditions of a few examples by hand. We now present the first tool-supported formal verification approach for backward compatibility or equivalence checking of object-oriented libraries. The paper provides the following technical contributions:

– The simulation-based reasoning method needed to prove two class libraries equivalent. In contrast to our previous work [41, 42], we extend the reasoning approach to the setting of recursion and loops.
– The *Invariant Specification Language* (ISL) to formally specify coupling invariants. ISL is a first-order logic-like specification language that provides facilities to express complex data and control flow relations between two library implementations.
– The publicly available[2] BCVerifier tool that implements the verification approach using the automatic verifier Boogie.
– A variety of classic examples that illustrate and validate the approach.

Outline. In the remainder of the paper, we introduce the necessary concepts and proof obligations along with a series of examples demonstrating the typical challenges that arise when proving backward compatibility of object-oriented libraries, such as subtyping, dynamic dispatch, callbacks, recursion and loops. The specification language ISL is introduced in lockstep with the examples. A structured overview of ISL can be found in Appendix B. Section 2 introduces the simulation-based reasoning approach. Section 3, the main section of this paper, presents the above-mentioned challenges, illustrates them using examples and gives solutions for dealing with the different forms of data and control flow abstraction. Section 4 applies the techniques to a more elaborate example and illustrates the various features of ISL in combination. Section 5 discusses the implementation of the BCVerifier tool. Section 6 presents the related work, and finally Section 7 concludes and discusses future work.

[2] For more information on how to access the tool, we refer to Appendix A.

2 Reasoning about Backward Compatibility

In the following, we introduce the core reasoning approach, its underlying formal model, and the terminology that is used throughout the rest of the paper. Due to space restrictions, we can only give a semi-formal account of the reasoning approach. A full formalization of the concepts with soundness and completeness proofs can be found in our earlier work [41, 42].

2.1 Interface Compatibility

A prerequisite for backward compatibility is that the new library implementation provides at least the interface[3] of the old implementation. In modern object-oriented languages, interfaces of libraries are complex due to the interplay of inheritance, subtyping, name-space mechanisms and accessibility modifiers. We assume that libraries consist of a collection of sealed packages [21], 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. 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. For example, narrowing the type of a method parameter breaks compatibility for callers whereas widening a parameter type breaks compatibility for the implementors [37]. 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 to specify that a class can only be subclassed in the library but not by clients. As a solution to this, Eclipse developers use additional annotations [16] to refine the description of library interfaces.

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 [20]. Source and binary compatibility are incomparable; neither one implies the other [12]. However, a set of checkable rules

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

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, large parts of the theory are directly transferable to a setting where binary compatibility is desired instead.

2.2 Preliminaries

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 BCVERIFIER tool 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 3.3), the observable states[4] are not statically bound to program points such as start and end of library methods. For simplicity, we only consider a single-threaded setting. A generalization of the theory to a setting with concurrency is considered as future work.

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, 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, differing 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 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.

2.3 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

[4] We use the term *observable states* in allusion to the *visible states* based techniques [15, 35].

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.

2.4 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.

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()). 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.

2.5 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 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 same name and similar signature was called and the receiver/parameters are indistinguishable, or for a method return whether the return values are indistinguishable.

For the Cell library in Fig. 1, program contexts can either call the 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). We illustrate the proof obligations for the method get using a Hoare triple [23]:

```
                           res1 := x1.get(); expose res1;
{ WfSim && Inv && x1 ~ x2 }  res2 := x2.get(); expose res2;  { WfSim && Inv }
                           add (res1,res2) to relation ~
```

We assume to be in indistinguishable states (WfSim) and assume that the user-supplied coupling invariant Inv holds. We also assume that the methods get are called on corresponding Cell objects (x1 ~ x2) where x1 is of the old and x2 of the new Cell type. We then consider the execution of both method bodies and expose the resulting values which are returned. Note that more care with possibly non-terminating methods and the special **null** value needs to be taken, which we omit in this example for simplicity. Finally the correspondence relation is updated with the returned values. We then need to check that we arrive in indistinguishable states again (e.g., that the updated correspondence relation is still bijective) and that the coupling invariant is preserved.

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. 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. In the following, we show the applicability of our theory using a number of examples.

3 Specification and Verification Technique

Automatic verification in the setting of object-orientation, callbacks, general recursion and loops is difficult. Each subsection describes typical challenges that arise in the setting of proving backward compatibility of object-oriented libraries. We then provide various means for dealing with the different forms of data and control flow abstraction that appear in libraries. Section 3.1 discusses type abstraction, Section 3.2 state abstraction, Section 3.3 callbacks and Section 3.4 more elaborate forms of control flow.

3.1 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.

As an example, consider a library implementation that has a public interface Fruit with implementing classes Apple and Banana. In the new implementation, the library developer decides to sometimes deliver objects of a new subclass ChiquitaBanana instead of the class Banana. This can be implemented for example using the factory method pattern [18] 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 Chiquita 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 2.4). To take these properties into account, the verifier needs a formal model of the type system and the properties of source compatibility. The BCVERIFIER encodes many such properties. For example, calls of methods with same name but on objects of unrelated types (see the example in Section 3.2) should not be considered as valid simulation steps.

3.2 State Abstraction

Libraries are usually composed of many classes and form their behavior with multiple cooperating objects. Consider the following example from Banerjee and Naumann [2]. 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. 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.

```
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(); }
}
```

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 Appendix B.

Even though the specification is now well-formed, the given coupling invariant does not provide enough static knowledge such that our program verifier can prove backward compatibility. Two more issues need to be solved. First, the verifier does not know whether the method calls g.set(...) and g.get() in lines 3, 5 and 7 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 overwriting methods. 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) Bool is the only possible implementation type. The ISL language provides a number of built-in predicates that reify the reasoning concepts of Section 2.3. The predicate *createdByLibrary* determines whether an object has been created by code of the library or program context:

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

An example where calling methods will lead to execution of code in the program context is given in the next subsection. The coupling invariant, as given, is still not strong enough to prove backward compatibility. A classical problem of object-orientation remains, 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 paper. 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;
```

and write the same for **new** OBool objects. Note that a typical ownership discipline [11] 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);
```

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 (No aliasing) is violated by this method, but equivalence can still be established using the second invariant. The key observation we made from examples similar 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 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 [10], 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 Section 3.4. In this case, they are solely used to assign a value to a ghost variable at a specific program point.[5] 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.

3.3 Callbacks

Callbacks are ubiquitous in object-oriented programs which makes reasoning very hard. The following adapted example [34, 2] 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.

```
public interface C {
  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; }
}
```

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, 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)

[5] In JML, the assignments to ghost variables are stated as comments in the code. As we would like to leave the implementations untouched and for tooling reasons, we opt to state the assignments as part of the specification.

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[6]. 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.[7] The quantification over all library stack slices in invariants that mention a stack frame is implicit in ISL:

```
place p1 = call run in line 8 of old A;
place p2 = call run in line 8 of new A;
invariant at(p1) ==> eval(p1, i) == 4;
```

The ISL specification relies on the built-in function *at* to denote that the execution is at a certain program point. The function *eval* allows to access the value of local variables at a certain program point. For example, the application *eval*(p1, i) yields the value of the local variable i at the program point denoted by p1. Place definitions enable the BCVERIFIER to typecheck *eval* expressions. 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.

3.4 Control Flow Relations

The splitting option is useful to break up the verification task of recursive computations which encompass external method calls. In a setting with no callbacks, library implementations can also hide complex control flow. Our solution is to provide various means to split the verification task into simpler subtasks. To prove steps that encompass methods with complex loops or recursion, we use **local simulation relations**. These auxiliary relations are used to prove properties that hold between custom user-definable non-observable states. The library developer specifies the program states for the old and new library implementation where the verification task is split at. These states are defined similarly to *conditional breakpoints* (debugging terminology) and are called **local places**. Using the local places, the library developer can establish local simulation relations that are specified using a **local invariant**. Local invariants are defined in a similar way as coupling invariants but talk only about the current (top-most, library) stack slice.

Synchronous Execution. As an example for local places and local invariants, consider the example below, adapted from Barthe, Crespo, and Kunz [6]. 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

[6] The splitting behavior can be deactivated for places with the **nosplit** option.
[7] A weaker invariant, stating that the value is even, is also sufficient.

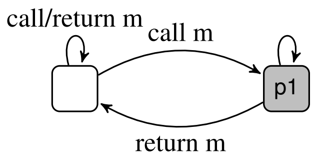
second iteration of the loop in the old implementation and the first iteration of the loop in the new implementation and then consecutive loop iterations. We first characterize the states in both library implementations that we want to relate. The first local place definition p1 denotes all program states 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 5 of the new implementation.

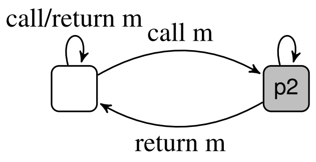
```
public class C { // old impl.          1
  public int m(int n){                 2
    int x = 0;                         3
    for(int i=0; i<n; i++){            4
      x += i;                          5
    }                                  6
    return x;                          7
} }      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

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

```
local place p1 = line 5 of old C when i > 0;
local place p2 = line 5 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);
```

The next step is to define the relation that ties the states of the two implementations together. As only a single local place is defined for each implementation, the implicit invariant *at*(p1)<==> *at*(p2) is added. Recall that the application *eval*(p1, n) yields the value of the local variable n at the program point denoted by p1. The local invariant states that in coupled non-observable states, the values of the local variables n, x, and i of both implementations coincide. The verification system considers for the *new* implementation four program paths, 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 $i-1 \geq n$.





For the old implementation, the verifier would have to consider an infinite number of paths: Executing the body of the for loop zero times, once, twice, thrice, etc. As a solution, loops and recursion are automatically unrolled up to a user-definable depth d. The verifier then proves that depth $d + 1$ is never reached. 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 copies of the loop body are sufficient because during the second iteration the place p1 with $i > 0$ is always reached. Effectively, the unrolling 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 verification system checks 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 local invariants are checked. The verification system also assumes two arbitrary local places such that the local invariants hold and starts execution from these places and checks whether the continuations behave similarly. For the example, the local invariant is proven to hold initially, for each iteration and finally the last iteration guarantees that the values returned by both methods are the same.

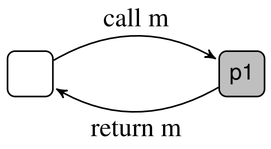
Asynchronous Execution. More complex relations can be specified that allow to relate one state in one implementation to many states in the other implementation. We say that one of the implementations is **stalled** while the other executes. Only one implementation can be stalled at a time. If the old 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 following example. 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.

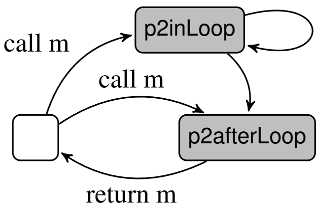
```
public class C { // old impl.                1
  public void m(int n) {                     2
    return;                                  3
} }                                          4
```

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

```
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);
```

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:





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[8] as the measure can otherwise not proved to be positive.

The previous for vs. while loop example (synchronous execution) could also be proven by defining the local place p1 without the condition i > 0 but stalling 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, and the asynchronous computation can be used to stall when for one implementation the loop has terminated and for the other implementation the call stack needs to be resolved.

4 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 [2]. 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. Figure 2 presents two library implementations of the observer pattern [18]. 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. Verification tasks for such implementations are 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.

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

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

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

```java
package util; // old impl.
public class Observable {
  private Node fst;

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

  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
  }
}
```

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

```java
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
  }
}
```

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

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

  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();
  }
}
```

Fig. 2. ObservableList example

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. Similar as for 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;
```

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

```
local place p1 = line 10 of old Observable assign x1 = newNode nosync;
local place p2 = line 7 of new Node assign x2 = this nosync;
```

The local places have the option **nosync** to configure that these places are not synchronized to places in the old implementation, i.e., no checks occur at this place.

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

```
local place q1 = line 18 of old Observable when c < i && n != null;
local place q2 = line 15 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 24 of new Observable nosync;
local place pn1 = line 34 of old Observable when n != null;
local place pn2 = line 19 of new Node when stackIndex(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 *stackIndex*(**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 3.3), 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 34 of old Observable nosplit;
place pc2 = call notifyObs in line 19 of new Node nosplit;
invariant at(pc1) && at(pc2) ==> related(bij, eval(pc1, n), eval(pc2, 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.[9]

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 37 of old Observable
  stall when stackIndex(new) > 1 with measure stackIndex(new);
local place qn2 = line 23 of new Node when stackIndex(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.

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

5 The BCVerifier Tool

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 paper. 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 [31]. Boogie is usually used as an intermediate language to study correctness of single programs with respect to specifications, for example Spec# [5], Dafny [30], or Chalice [33]. 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 [4] can then generate logical formulas which can be fed to theorem provers. For our studies, the SMT solver Z3 was used. Our Boogie encoding is explained in one of the authors Master thesis [40]. 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 Appendix B) 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 web frontend of the BCVerifier tool only supports the ISL syntax, whereas the command line version supports both (see Appendix A).

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, first incrementally checking reachability of certain paths and then selective unrolling.

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 is still very simple: when the verification process fails, then the verifier simply returns a path through the generated BOOGIE representation of the libraries, that lead to the issue. An ongoing effort is to map errors back into the high-level language. 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.

6 Related Work

In the related work, we focus on three aspects: work on reasoning about the behavioral equivalence of object-oriented program parts, other program comparison techniques and tool implementations of the approaches.

In the setting of object-oriented programs, the two most popular ways to reason about the behavior of class implementations is to use denotational methods and/or bisimulations. Banerjee and Naumann [2] 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. In subsequent work [3], they use a discipline using assertions and ghost fields to specify invariants and heap encapsulation (by ownership techniques) and to deal with reentrant callbacks.

Jeffrey and Rathke [25] give a fully abstract trace semantics for a Java subset with a package-like construct. Inheritance, down-casting and cross-border instantiation are not considered. Using similar techniques, Steffen [38] and Ábrahám et al. [1] give a fully abstract semantics for a concurrent class-based language (without inheritance and subtyping). It remains unclear in both cases how the trace semantics can be applied for verification purposes.

Koutavas and Wand, building on their earlier work [26] and the work of Sumii and Pierce [39], use bisimulations on a standard operational semantics to reason about the equivalence of *single* classes [27] in different Java subsets. The subset they consider includes inheritance and down-casting but neither interfaces nor accessibility of types.

In previous work [42], we have given a fully abstract trace-based semantics for class libraries of a sequential object-oriented programming language with interfaces, classes, inheritance and subtyping. In this model, backward compatibility corresponds to trace inclusion. To model encapsulation aspects, we considered a package system which allows package-local types. Geared towards practical application, the language is a more faithful subset of Java than in other formalizations [25]. The main advantage of our fully abstract trace-based semantics to the previously discussed ones is that it provides a strong link to a standard operational model. In subsequent work [41], we have shown that the trace inclusion can be proven by a particular form of simulations that are sound and complete with respect to backward compatibility. The notion of confinement (internal or exposed objects), in contrast to the work of Banerjee and Naumann, results from the encapsulation offered by the module system, in our case Java packages, and is fixed by the language semantics. An explicit representation of when object invariants are known to hold is not needed. In our approach, the language and module system dictate what the observable states are in which invariants must hold. This makes our reasoning method complete with respect to contextual equivalence. For example, we allow sharing the representation between different boundary objects (e.g., the extension of the OBool example in Section 3.2) which is not possible in other works [3]. A more detailed description of our reasoning approach and comparison to other works can be found in our theory paper [41]. Note that none of the previously mentioned works present an embedding of their reasoning approach into a mechanized verification framework.

Another body of work focuses on program comparison techniques not specifically developed for the object-oriented setting. Relational Hoare Logics [9] and Relational Separation Logic [44] provide a custom logic 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 [28] 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 [7] 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 [13] use a dynamic logic and the KeY tool, whereas Naumann [36] employs the ESC/Java and Spec# tools to verify information flow properties. Leino and Müller [32] use self-composition to verify with BOOGIE that two executions of the same method yield equivalent results.

Barthe, Crespo, and Kunz [6] 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 (see Section 3.1) 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 [19] 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. [22] 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 [29], 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.

7 Conclusion and Future Work

In this paper, we have presented BCVERIFIER, the first formal verifier to check equivalence of class library implementations. The verifier is complemented with a specification language that allows to formulate coupling invariants that describe the connection between the library implementations. We have employed the technique to verify a number of classic examples in the literature. As far as we are aware, this is the first time mechanical verification has been used to study the equivalence between two different class library implementations. Although there remain open questions, we have shown that mechanical (semi-automatic) verification of equivalence for structurally similar object-oriented libraries is feasible. We invite the reader to play with and check further examples on the web site (see Appendix A).

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. In the future, we would like to migrate more features from specification-based techniques in the single-program world to the world of two programs, e.g., pre-post specifications of methods. Another important aspect is to introduce modularity in the reasoning approach, as seen in the works of Banerjee and Naumann [3], to ensure scalability of the approach.

To enhance automation, a vast improvement would be to include in BCVERIFIER an automatic inference of single-program properties such as nullness, exposedness of fields etc. 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. We would also like to add further specification-only types such as sets, lists and maps to ISL, and establish higher level of specification constructs for typically occurring relations, which would allow for example for an easy description of the relation between a loop and its corresponding tail-recursive method.

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 [16]). Currently this can be regulated by using appropriate access modifiers. It remains future work to investigate other more complex restrictions on contexts and to extend the specification and verification technique to handle these restrictions.

References

[1] Ábrahám, E., Bonsangue, M. M., Boer, F. S. de, Steffen, M. "Object Connectivity and Full Abstraction for a Concurrent Calculus of Classes". In: *ICTAC*. Ed. by Liu, Z., Araki, K. Vol. 3407. LNCS. Springer, 2004, pp. 37–51.

[2] Banerjee, A., Naumann, D. A. "Ownership confinement ensures representation independence for object-oriented programs". In: *J. ACM* 52.6 (2005), pp. 894–960.

[3] Banerjee, A., Naumann, D. A. "State Based Ownership, Reentrance, and Encapsulation". In: *ECOOP*. Ed. by Black, A. P. Vol. 3586. LNCS. Springer, 2005, pp. 387–411.

[4] Barnett, M., Chang, B.-Y. E., DeLine, R., Jacobs, B., Leino, K. R. M. "Boogie: A Modular Reusable Verifier for Object-Oriented Programs". In: *FMCO*. Ed. by Boer, F. S. de, Bonsangue, M. M., Graf, S., Roever, W. P. de. Vol. 4111. LNCS. Springer, 2005, pp. 364–387.

[5] Barnett, M., Leino, K. R. M., Schulte, W. "The Spec# Programming System: An Overview". In: vol. 3362. LNCS. Springer-Verlag, 2005, pp. 49–69.

[6] Barthe, G., Crespo, J. M., Kunz, C. "Relational Verification Using Product Programs". In: *FM*. Ed. by Butler, M., Schulte, W. Vol. 6664. LNCS. Springer, 2011, pp. 200–214.

[7] Barthe, G., D'Argenio, P. R., Rezk, T. "Secure information flow by self-composition". In: *Mathematical Structures in Computer Science* 21.6 (2011), pp. 1207–1252.

[8] Beckert, B., Hähnle, R., Schmitt, P. H. *Verification of Object-Oriented Software: The KeY Approach*. Vol. 4334. LNCS. Berlin: Springer-Verlag, 2007.

[9] Benton, N. "Simple relational correctness proofs for static analyses and program transformations". In: *POPL*. Ed. by Jones, N. D., Leroy, X. ACM, 2004, pp. 14–25.

[10] Chalin, P., Kiniry, J. R., Leavens, G. T., Poll, E. "Beyond Assertions: Advanced Specification and Verification with JML and ESC/Java2". In: *FMCO*. Ed. by Boer, F. S. de, Bonsangue, M. M., Graf, S., Roever, W. P. de. Vol. 4111. LNCS. Springer, 2005, pp. 342–363.

[11] Clarke, D. G., Potter, J., Noble, J. "Ownership Types for Flexible Alias Protection". In: *OOPSLA*. Ed. by Freeman-Benson, B. N., Chambers, C. ACM, 1998, pp. 48–64.

[12] Darcy, J. D. *Kinds of Compatibility: Source, Binary, and Behavioral*. `http://blogs.oracle.com/darcy/entry/kinds_of_compatibility`.

[13] Darvas, Á., Hähnle, R., Sands, D. "A Theorem Proving Approach to Analysis of Secure Information Flow". In: *SPC*. Ed. by Hutter, D., Ullmann, M. Vol. 3450. LNCS. Springer, 2005, pp. 193–209.

[14] Dig, D., Johnson, R. E. "How do APIs evolve? A story of refactoring". In: *Journal of Software Maintenance* 18.2 (2006), pp. 83–107.

[15] Drossopoulou, S., Francalanza, A., Müller, P., Summers, A. J. "A Unified Framework for Verification Techniques for Object Invariants". In: *ECOOP*. Ed. by Vitek, J. Vol. 5142. LNCS. Springer, 2008, pp. 412–437.

[16] Eclipse PDE API Tools. `http://www.eclipse.org/pde/pde-api-tools/`.

[17] Filliâtre, J.-C., Marché, C. "The Why/Krakatoa/Caduceus Platform for Deductive Program Verification". In: *CAV*. Ed. by Damm, W., Hermanns, H. Vol. 4590. LNCS. Springer, 2007, pp. 173–177.

[18] Gamma, E., Helm, R., Johnson, R., Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[19] Godlin, B., Strichman, O. "Regression Verification: Proving the Equivalence of Similar Programs". In: *Software Testing, Verification and Reliability* (2012).

[20] Gosling, J., Joy, B., Steele, G., Bracha, G. *The Java Language Specification, Third Edition*. The Java Series. Boston, Mass.: Addison-Wesley, 2005.

[21] Grothoff, C., Palsberg, J., Vitek, J. "Encapsulating Objects with Confined Types". In: *OOPSLA*. Ed. by Northrop, L. M., Vlissides, J. M. ACM, 2001, pp. 241–253.

[22] Hawblitzel, C., Kawaguchi, M., Lahiri, S., Rebêlo, H. *Mutual Summaries and Relative Termination*. Tech. rep. MSR-TR-2011-112. Microsoft Research, 2011.

[23] Hoare, C. A. R. "An Axiomatic Basis for Computer Programming". In: *Commun. ACM* 12.10 (1969), pp. 576–580.

[24] Jacobs, B., Smans, J., Piessens, F. "A Quick Tour of the VeriFast Program Verifier". In: *APLAS*. Ed. by Ueda, K. Vol. 6461. LNCS. Springer, 2010, pp. 304–311.

[25] Jeffrey, A., Rathke, J. "Java Jr: Fully Abstract Trace Semantics for a Core Java Language". In: *ESOP*. Ed. by Sagiv, S. Vol. 3444. LNCS. Springer, 2005, pp. 423–438.

[26] Koutavas, V., Wand, M. "Bisimulations for Untyped Imperative Objects". In: *ESOP*. Ed. by Sestoft, P. Vol. 3924. LNCS. Springer, 2006, pp. 146–161.

[27] Koutavas, V., Wand, M. "Reasoning About Class Behavior". In: *Informal Workshop Record of FOOL*. 2007.

[28] Kundu, S., Tatlock, Z., Lerner, S. "Proving optimizations correct using parameterized program equivalence". In: *PLDI*. Ed. by Hind, M., Diwan, A. ACM, 2009, pp. 327–337.

[29] Lahiri, S. K., Hawblitzel, C., Kawaguchi, M., Rebêlo, H. "SYMDIFF: A Language-Agnostic Semantic Diff Tool for Imperative Programs". In: *CAV*. Ed. by Madhusudan, P., Seshia, S. A. Vol. 7358. LNCS. Springer, 2012, pp. 712–717.

[30] Leino, K. R. M. "Dafny: An Automatic Program Verifier for Functional Correctness". In: *LPAR (Dakar)*. Ed. by Clarke, E. M., Voronkov, A. Vol. 6355. LNCS. Springer, 2010, pp. 348–370.

[31] Leino, K. R. M. *This is Boogie 2. Manuscript KRML 178*. Available at `http://research.microsoft.com/~leino/papers.html`. 2008.

[32] Leino, K. R. M., Müller, P. "Verification of Equivalent-Results Methods". In: *ESOP*. Ed. by Drossopoulou, S. Vol. 4960. LNCS. Springer, 2008, pp. 307–321.

[33] Leino, K. R. M., Müller, P., Smans, J. "Verification of Concurrent Programs with Chalice". In: *FOSAD*. Ed. by Aldini, A., Barthe, G., Gorrieri, R. Vol. 5705. LNCS. Springer, 2009, pp. 195–222.

[34] Meyer, A. R., Sieber, K. "Towards Fully Abstract Semantics for Local Variables". In: *POPL*. Ed. by Ferrante, J., Mager, P. ACM Press, 1988, pp. 191–203.

[35] Meyer, B. *Object-Oriented Software Construction, 2nd Edition*. Prentice-Hall, 1997.

[36] Naumann, D. A. "From Coupling Relations to Mated Invariants for Checking Information Flow". In: *ESORICS*. Ed. by Gollmann, D., Meier, J., Sabelfeld, A. Vol. 4189. LNCS. Springer, 2006, pp. 279–296.

[37] Rivières, J. des. *Evolving Java-based APIs*. `http://wiki.eclipse.org/Evolving_Java-based_APIs`.

[38] Steffen, M. "Object-Connectivity and Observability for Class-Based, Object-Oriented Languages". Habilitation thesis. Technische Faktultät der Christian-Albrechts-Universität zu Kiel, July 2006.

[39] Sumii, E., Pierce, B. C. "A bisimulation for type abstraction and recursion". In: *J. ACM* 54.5 (2007).

[40] Weber, M. "Generating Boogie Verification Conditions for Backward Compatibility of Class Libraries". Available at `http://softech.cs.uni-kl.de/pub?id=191`. MA thesis. University of Kaiserslautern, Oct. 2012.

[41] Welsch, Y., Poetzsch-Heffter, A. "A Fully Abstract Trace-based Semantics for Reasoning About Backward Compatibility of Class Libraries". Submitted for journal publication, draft available at `http://softech.cs.uni-kl.de/~welsch`. Apr. 2012.

[42] Welsch, Y., Poetzsch-Heffter, A. "Full Abstraction at Package Boundaries of Object-Oriented Languages". In: *SBMF*. Ed. by Silva Simão, A. da, Morgan, C. Vol. 7021. LNCS. Springer, 2011, pp. 28–43.

[43] Welsch, Y., Poetzsch-Heffter, A. "Verifying Backwards Compatibility of Object-Oriented Libraries Using Boogie". In: FTfJP '12. New York, NY, USA: ACM, 2012, pp. 35–41.

[44] Yang, H. "Relational separation logic". In: *Theor. Comput. Sci.* 375.1-3 (2007), pp. 308–334.

A Appendix: BCVERIFIER Tool and Additional Examples

Additional examples are on the website running the BCVERIFIER web frontend:

`http://softech.cs.uni-kl.de/bcverifier`

To preserve the anonymity of reviewers, no statistics/tracking software is running on the website. Note that larger examples can take up to a few minutes to successfully verify. The source code of BCVERIFIER and instructions to build the command-line client and run the web frontend locally are available at the following public Mercurial repository:

`http://softech.cs.uni-kl.de/hg/public/bcverifier`

B Appendix: The Invariant Specification Language

B.1 Syntax

Non-terminals are represented in ALL CAPS. We use the meta-symbol | to denote alternatives and the brackets [...] to group elements. The meta-symbol $[X]^?$ is used to denote an optional item X and $[X]^*$ to denote an arbitrary sequence of elements X. The non-terminal ID represents identifiers and INT integer constants.

```
SPECIFICATION ::= [DECLARATION]*

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

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

ASSIGN ::= assign ID = EXPRESSION

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

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

VARDEF ::= TYPEDEF ID

TYPEDEF ::= int | boolean | binrelation | [[old | new] ID [.ID]*]

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

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

B.2 Types

Currently the following types are supported:

- Java primitive types **boolean** and **int**.
- Java class and interface types prefixed 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**).

The correspondence operator ~ expects two reference values: one from the old library on the left hand side and one from the new library on the right hand side. Termination measures are specified by (positive) integer expressions.

Local variables appearing in *eval* and *at* are checked whether they are defined at the specific program point and their type is used to type the contained expression. Top-level expressions appearing in local place definitions are implicitly wrapped with *eval* for that place.

B.3 Semantics

Global invariants have to hold at every observable point. Local invariants must hold at local places and are used to prove global invariants. Invariants that contain global places are implicitly all-quantified over all possible library stack slices. Local places only talk about the current top-most stack slice.

The expressions e that appear in the definitions of local places are per default wrapped in *eval*(p, e) where p denotes the place that is being defined.

Correspondence relation operator. An expression o1 ~ o2 yields true if and only if o1 and o2 are two objects in correspondence or o1 and o2 are both **null**.

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

- **boolean** *exposed*(Object o) returns whether the object o is exposed or not.
- **boolean** *createdByLibrary*(Object o) returns true when the object o is created by the library and false when it was created by the program context.
- **int** *stackIndex*(**old**) or **int** *stackIndex*(**new**) returns the number for the top-most stack frame in the considered stack slice for the old or the new library implementation.
- **boolean** *at*(**place** p, **int** i) returns whether the stack frame numbered i in the considered stack slice is currently at place p.
- **boolean** *at*(**place** p) is a shorthand for *at*(p, *stackIndex*(**old**)) or *at*(p, *stackIndex*(**new**)) depending on whether the place p was defined for the old or new implementation.
- T *eval*(**place** p, **int** i, EXPRESSION<T> e) evaluates the expression e in the context of the place p. This means that local Java variables, that are visible at the given place, can be used. The values of the variables will be taken from the stack frame in the relative position i of the considered stack slice (i must be in the range 0 to *stackIndex*(...) with bounds inclusive).

– T *eval*(**place** p, EXPRESSION<T> e) is a shorthand for *eval*(p, *stackIndex*(**old**), e) or for *eval*(p, *stackIndex*(**new**), e) depending on whether the place p was defined for the old or new implementation.
– **boolean** *related*(**binrelation** b, **old** Object o1, **new** Object o2) returns whether 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 o1 and o2 are added.
– **binrelation** *remove*(**binrelation** b, **old** Object o1, **new** Object o2) returns the same relation as b except where o1 and o2 are removed.

B.4 Well-formedness

To check whether a place or an invariant is well-formed, a separate proof obligation is generated.

Local places. Stalling local places 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 that correspond to the state in the old implementation. Conversely, stalling local places that are defined for the new implementation cannot have a termination measure (as we do not care about the behavior of the new library implementation if the old library implementation diverges). Local places should not appear in non-local invariants.

Expressions. Typical well-formedness conditions for expressions are the following.

– There must not be any division by zero
– **null** must not be dereferenced

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. Therefore 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. Invariants defined at the top 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 C c :: c.x != 0
invariant forall C c :: 10 / c.x > 3
```

In the following, we give a formal 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 illustrated here.

$$\begin{array}{l}
\mathsf{WD}\left[\textbf{if}\ \mathsf{e}\ \textbf{then}\ \mathsf{e1}\ \textbf{else}\ \mathsf{e2}\right] := \mathsf{WD}\left[\mathsf{e}\right] \wedge ((\mathsf{TR}\left[\mathsf{e}\right] \Rightarrow \mathsf{WD}\left[\mathsf{e1}\right]) \wedge (\mathsf{TR}\left[\mathsf{e}\right] \rightarrow \mathsf{WD}\left[\mathsf{e3}\right])) \\
\mathsf{WD}\left[\mathsf{e.f}\right] := \mathsf{WD}\left[\mathsf{e}\right] \wedge \mathsf{TR}\left[\mathsf{e}\ \mathsf{!=}\ \textbf{null}\right] \\
\mathsf{WD}\left[\mathsf{e1}\ \&\&\ \mathsf{e2}\right] := \mathsf{WD}\left[\mathsf{e1}\right] \wedge (\mathsf{TR}\left[\mathsf{e1}\right] \Rightarrow \mathsf{WD}\left[\mathsf{e2}\right]) \\
\mathsf{WD}\left[\mathsf{e1} \mathrel{\mathsf{==>}} \mathsf{e2}\right] := \mathsf{WD}\left[\mathsf{e1}\ \&\&\ \mathsf{e2}\right] \\
\mathsf{WD}\left[\mathsf{e1}\ \|\ \mathsf{e2}\right] := \mathsf{WD}\left[\mathsf{e1}\right] \wedge (\neg\mathsf{TR}\left[\mathsf{e1}\right] \Rightarrow \mathsf{WD}\left[\mathsf{e2}\right]) \\
\mathsf{WD}\left[\mathsf{e1}\ /\ \mathsf{e2}\right] := \mathsf{WD}\left[\mathsf{e1}\right] \wedge \mathsf{WD}\left[\mathsf{e2}\right] \wedge \mathsf{TR}\left[\mathsf{e2}\ \mathsf{!=}\ 0\right] \\
\mathsf{WD}\left[\mathsf{e1}\ \%\ \mathsf{e2}\right] := \mathsf{WD}\left[\mathsf{e1}\ /\ \mathsf{e2}\right] \\
\mathsf{WD}\left[\textbf{forall}\ \mathsf{x} :: \mathsf{e}\right] := \mathsf{TR}\left[\textbf{forall}\ \mathsf{x}\right] : \mathsf{WD}\left[\mathsf{e}\right] \\
\mathsf{WD}\left[\textbf{exists}\ \mathsf{x} :: \mathsf{e}\right] := \mathsf{TR}\left[\textbf{forall}\ \mathsf{x}\right] : \mathsf{WD}\left[\mathsf{e}\right] \\
\mathsf{WD}\left[\mathit{at}(\mathsf{p}, \mathsf{e})\right] := \mathsf{WD}\left[\mathsf{e}\right] \wedge \mathsf{TR}\left[0 \mathrel{\mathsf{<=}} \mathsf{e}\right] \wedge \mathsf{TR}\left[\mathsf{e} \mathrel{\mathsf{<=}} \mathit{stackIndex}(...)\right] \\
\mathsf{WD}\left[\mathit{eval}(\mathsf{p}, \mathsf{e1}, \mathsf{e2})\right] := \mathsf{WD}\left[\mathit{at}(\mathsf{p}, \mathsf{e1})\right] \wedge \mathsf{TR}\left[\mathit{at}(\mathsf{p}, \mathsf{e1})\right] \wedge \mathsf{WD}\left[\mathsf{e2}\right] \\
\mathsf{WD}\left[\mathit{exposed}(\mathsf{e})\right] := \mathsf{TR}\left[\mathsf{e}\ \mathsf{!=}\ \textbf{null}\right] \\
\mathsf{WD}\left[\mathit{createdByLibrary}(\mathsf{e})\right] := \mathsf{TR}\left[\mathsf{e}\ \mathsf{!=}\ \textbf{null}\right]
\end{array}$$

The check whether the termination measure is positive (not shown here) is done after local invariants are assumed to hold.