Verifying Backwards Compatibility of Object-Oriented Libraries Using Boogie*

Extended Abstract

Yannick Welsch
University of Kaiserslautern, Germany
welsch@cs.uni-kl.de

Arnd Poetzsch-Heffter
University of Kaiserslautern, Germany
poetzsch@cs.uni-kl.de

ABSTRACT

Proving that a library is backwards compatible to an older version can be challenging, as the internal representation of the libraries might completely differ and the clients of the library are usually unknown. This is especially difficult in the setting of object-oriented programs with complex heaps and callbacks. Mechanical verification is a key success factor to make such proofs practicable.

In this paper, we present a technique to verify the backwards compatibility or equivalence of class libraries in the setting of unknown program contexts. For a number of textbook examples we have formulated the verification conditions as input to the Boogie program verification system and validated the approach.

Categories and Subject Descriptors

D.3.1 [**Programming languages**]: Formal Definitions and Theory—*Semantics*; F.3.1 [**Logics and meanings of programs**]: Specifying and Verifying and Reasoning about Programs—*Mechanical verification*

General Terms

Languages, Theory, Verification

Keywords

Full abstraction, Class libraries, Trace semantics, Contextual equivalence, Backward compatibility

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 backwards compatibility with existing clients (called *breaking API changes* in [11]), but often libraries should be modified, extended, or refactored in such a way that client code is not affected. Software developers can use informal guidelines (e.g., [31]) and special tools (e.g., [13]) to check compatibility aspects. Whereas in previous work [10] we studied a type system to support the interface evolution of class libraries, we focus on behavioral properties in this paper. Reasoning about the behavioral equivalence of two library implementations is not only useful when no specifications are available; we conjecture (similar to Godlin and Strichman [17]) that it may often be simpler to verify the 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 (e.g., [5, 7, 15, 21]). Our ultimate goal is to leverage the power of existing verification tools for object-oriented programs to safely check that two different OO library implementations exhibit the same behavior in the setting of unknown program contexts.

Reasoning about the equivalence of class library implementations is challenging for the following reasons: 1) the number of possible contexts is infinite and contexts are complex and 2) the states and heaps can be significantly different between the different library implementations. In order to focus on verification aspects, we use the sound and complete method from [35] for reasoning about backwards compatibility of class libraries that we developed specifically for this task at hand. To simplify the presentation and the formal model in this paper, we focus on (contextual) equivalence, which is the symmetric version of backwards compatibility.

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

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

Figure 1: Cell example

Example. To illustrate our goals, let us consider a very simple library at the left of Fig. 1 which provides a Cell class to store and retrieve references to objects. 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

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

FTfJP'12, June 12, 2012, Beijing, China

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 was set. This second representation allows to add a method to retrieve the previous value, e.g. public Object previous(){ return f ? c_2: c_1; }.

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. 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 c_1 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 c_2 corresponds to the value that is stored in c. In the remainder of the paper, we give a formal underpinning to this intuition. We show how to capture the relation (called *coupling invariant*) between the two libraries formally and how to automatically verify the equivalence of such libraries using Boogie.

Related work. There is a large body of work on studying the equivalence of programs and program parts. In this presentation, we focus on such studies 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 or bisimulations.

Denotational methods have been successfully used to investigate properties of object-oriented programs [9]. The denotational semantics according to these methods provide representations of program parts (e.g., classes) as mathematical objects describing how program parts modify the stack and heap. However, these denotations are often not *abstract* enough, i.e., they differentiate between classes that have the same behavior. 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 [22] give a fully abstract trace semantics for a Java subset with a package-like construct. However, they do not consider inheritance, down-casting and cross-border instantiation. It remains unclear how their trace semantics can be applied for verification purposes. Using similar techniques, Steffen [32] and Ábrahám et al. [1] give a fully abstract semantics for a concurrent class-based language (without inheritance and subtyping). In the setting of concurrent data structures, Gotsman and Yang [18] use traces and a most general client to check whether a library linearizes another. Filipovic et al. [14] use a trace abstraction to study whether sequential consistency or linearizability implies observational refinement.

Bisimulations were first used by Hennessy and Milner [19] to reason about concurrent programs. Sumii and Pierce used bisimulations which are sound and complete with respect to contextual equivalence in a language with dynamic sealing [33] and with type abstraction and recursion [34]. Koutavas and Wand, building on their earlier work [24] and the work of Sumii and Pierce, used bisimulations to reason about the equivalence of *single* classes [25] in different Java subsets. The subset they considered includes inheritance and down-casting. Their language, however, neither considers interfaces nor accessibility of types.

Our approach. In previous work [35], we have given a fully abstract trace-based semantics for class libraries of a *sequential* object-oriented programming language with the typical OO features, namely interfaces, classes, inheritance and subtyping. To model encapsulation aspects, we considered a package system which allows package-local types. As our semantics is geared towards practical application, the language is a more faithful subset of Java than [22]. The main idea behind our fully abstract semantics was to combine characteristics from (1) denotational, trace-based semantics, i.e., the mental model of traces which characterize the behavior of a library in terms of its input and output, as well as (2) bisimulation approaches, i.e., by providing a strong link to a standard operational semantics which leads to a direct connection to Hoare-like program logics.

The idea behind our verification method is based on the states where control is outside of the library, which we call the *observable states*[1]. The (coupling) invariant that relates the configurations of both libraries must hold at corresponding observable states in the execution. As libraries are comprised of multiple classes, the observable states are, in contrast to [12, 29, 30], not statically bound to program points like start and end of methods.

Our approach is modular in the sense that we do not need knowledge about the contexts of the library. In this paper, we do not consider the other kind of modularity, namely that the task of verifying two libraries equivalent can be split into smaller tasks (i.e., relating smaller parts of the libraries first). In contrast to Banerjee and Naumann [3], we consider multiple classes and we do not use a special *inv/own* discipline as described in [4], i.e., an explicit representation of when object invariants are known to hold. In our case, it is clear when the invariant must hold, namely in observable states which result directly from the language and module system. On one hand, we have a less parametric framework than Banerjee and Naumann as the notion of confinement (that results from the encapsulation offered by the module system, in our case Java packages) is fixed by the language semantics. On the other hand, our framework is more powerful as it is not tied to an external confinement discipline. In particular, our reasoning method is complete with respect to contextual equivalence. We allow for example different boundary objects to share their representation (e.g. a list with iterators), which is not possible in [3].

All of the related work that we are aware of present no embedding of their reasoning framework into a mechanized verification framework. The closest we could find was the mechanized bisimulation for the nu-calculus by Benton and Koutavas [8] in Coq. We believe that due to the OO setting with nominal type system, object identity etc., we can achieve a high degree of automation. As a proof of concept, we use a fully automatic verifier to check interesting equivalences.

Contribution and outline. To the best of our knowledge, we present the first tool-supported formal verification approach for (modular) equivalence checking of OO libraries.

In Sect. 2 we give an introduction on how the behavior of a class library can be captured in terms of traces. Based on this characterization of the behavior we present in Sect. 3 how we can prove the equivalence between two implementations of a library. In Sect. 4 we outline for a simple example how to model the library implementations and the verification conditions in Boogie. Finally, we conclude in Sect. 5 and discuss future work.

[1] We use the term *observable states* in allusion to the *visible states* based techniques [12, 29].

2. LIBRARY EQUIVALENCE

In this section, we give an informal introduction to the fully abstract trace-based semantics that is formalized in [35]. In our approach, the denotation of a library, which are sets of packages that consist of classes and interfaces, is expressed by the interactions between code belonging to the library and code belonging to (program) contexts. It is defined in two steps starting from a standard operational semantics. In the first step, the operational semantics is augmented in a way that the interactions can be made explicit. This is similar as in formalizations geared towards type soundness proofs, where runtime configurations are usually augmented with additional type information. In the second step, traces of interaction labels are used to semantically characterize the package behavior. A non-trivial aspect is the treatment of inheritance, because with inheritance, some code parts of a class/object might belong to the context and other parts to the library under investigation.

Using traces allows abstracting from the state and heap representation in the two versions of the library. To obtain a finite representation of all contexts, we construct a nondeterministic *most general context* that exactly exhibits the possible behavior of contexts. Using an operational semantics as a starting point has the advantage that we can use simulation relations applied to standard configurations (i.e., heap and stack) for the full abstraction proof. Furthermore, it provides a direct formal relation to Hoare-like program logics.

To illustrate the trace semantics, we consider a simple utility library in Fig. 2 that provides classes to implement the Subject/Observer pattern. The observers, which are implementing the Observer interface, can be added to the Subject using the addObserver method and are stored in a linked list. The Subject also offers the possibility to get an iterator, basically a cursor, to navigate over the list of registered observers. Furthermore, the Subject provides the convenience method notify to update all the registered observers with the given argument. The example illustrates some of the difficulties when dealing with representation independence of OO libraries: (1) Multiple roles: The LinkedList class can be used both for boundary objects (as it is public) as well as an internal representation of the Subject class. Furthermore, there may be multiple boundary objects (e.g., iterators) accessing the same internal data (e.g. the LinkedList representation). (2) Type abstraction: The Iterator interface allows abstraction from concrete class implementations, i.e., clients of the library do not need to be aware of the concrete Iterator type. Vice-versa, the library does not need to know about possible classes in the program context implementing the Observer interface. (3) Callbacks: During a notification, i.e., call of the update method, an observer can make a callback to the Subject under consideration.

2.1 Traces

In order to abstract from the complex representation of the library (heap and stack configurations), we describe the library in terms of its *input/output* behavior. The main question to address is what we consider as the points where such observable behavior occurs and what the input/output information is. As we are in a sequential setting, control flow can at a fixed point in the execution either be in code of the library or in code of the program context. The points of observation thus become those where control flow changes from the library to the context and vice-versa. The behavior of a program context with a library is described by a trace, i.e., a sequence of labels that record the input/output between the context and the library. Due to the full

```
1  public interface Observer {
2    public void update(Object
         arg);
3  }
4  public class Subject {
5   private LinkedList obs;
6   public void
         addObserver(Observer
         o) {...}
7   public Iterator iterator() {
8    return new ObsIter();
9   }
10  public void notify(Object
         arg) {
11   while (...) {...
         o.update(arg); ...}
12   }
13   ...
14   private class ObsIter
         implements Iterator {
15    private int currIdx;
16    ObsIter() {...}
17    public boolean hasNext()
           {...}
18    public Object next() {...}
19   }
20  }
21  public interface Iterator {
22   public boolean hasNext();
23   public Object next();
24  }
25  public class LinkedList {...}
```

Figure 2: Observer example

```
1  public class IntObs
       implements Observer {
2   private int count = 0;
3   public void update(Object
         arg) {
4    count += (Integer)arg;
5   }}
6   // Body of main method
7   Subject sj = new Subject();
8   Observer ob = new IntObs();
9   sj.addObserver(ob);
10  sj.notify(new Integer(5));
```

Figure 3: Program context for Observer example

abstraction result in [35], we know the form of the input/output labels such that they capture all relevant information about the behavior of the library.

To illustrate how these traces look like, let us consider the program context in Fig. 3, which consists of an implementation of the Observer interface and a main method which uses the Subject. The traces which are generated by the program that consists of this program context and the utility library in Fig. 2 are of the form

$$\text{call } o_1.\text{addObserver}(o_2)\boxplus \cdot \text{rtrn } \_\hat{\boxplus} \cdot \text{call } o_1.\text{notify}(o_3)\boxplus \cdot \text{call } o_2.\text{update}(o_3)\hat{\boxplus} \cdot \text{rtrn } \_\boxplus \cdot \text{rtrn } \_\hat{\boxplus}$$

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

Program execution starts at line 6 (beginning of main method) of the program context in Fig. 3. Assuming default constructors, the first change in control happens at line 9, where the method addObserver is called. Execution jumps to the beginning of the body of this method, which is at line 6 in Fig. 2. Due to the change in control from the program context to the library, the input label call o_1.addObserver(o_2)$\boxplus$ is recorded. It contains the information that we have a method call of the method addObserver, that two distinct objects o_1 and o_2 are callee and parameter of the method call and that the direction of the change in control is from the program context to the library, which is denoted by $\boxplus$ for input. Within the body of the addObserver method (which is not shown), the observer is added to the list of observers and the method returns. When the method returns, we have again a change in control, but this time in the reverse direction. The output label rtrn $\_\hat{\boxplus}$ is thus recorded, which contains the information that the change in control is due to a method return, that no values are passed (void method) and $\hat{\boxplus}$ denotes that this is an output label. We are now back executing in the program context and execute the next statement in line 10 of Fig. 3 by calling the method notify. This leads to the label call o_1.notify(o_3)$\boxplus$, which contains the information that the method is called on the same object o_1 that we called addObserver before. In the body of the notify method

at line 11 in Fig. 2, the program iterates over the (singleton) list of registered observers and calls the update method. This leads again to a change in control as the update method for this particular observer has been defined in the program context. This is recorded by the label call o_2.update(o_3)↑ that shows exactly which objects are involved in this call. Finally the update method returns (rtrn _↓), then the notify method returns (rtrn _↑) and the main method terminates.

We have achieved to characterize the behavior of a library (with a program context) independently of the concrete representation of the library (i.e. heaps and stacks). However, we want to describe the behavior of a library not only in terms of a specific program context, but of all possible program contexts.

2.2 Most general context

We introduce a most general context (MGC) that enables all interactions that a concrete program context can engage in. Compared to a concrete program context, the most general context abstracts over types, objects and operational steps. In order to see what the MGC can do, let us reflect on what concrete contexts can do. In terms of program code, program contexts can define classes that rely on the library, either by calling code of the library or by extending classes and implementing interfaces of the library (e.g. class IntObs in Fig. 3). In terms of operational steps, program contexts can (1) create objects of classes that are defined in the program context (e.g. IntObs in line 8 of Fig. 3) or objects of classes that are defined in the library and accessible to the program context (e.g. Subject in line 7, but not ObsIter). (2) perform steps that do not lead to a change in control, e.g., access and write fields and method calls/returns that are dispatched to code of the program context (e.g. line 4 in Fig. 3). The MGC abstracts over these steps. (3) call methods that are defined in the library (e.g. line 9 in Fig. 3) or return to code that is defined in the library (e.g. line 5 in Fig. 3).

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

2.3 Formal model

In this subsection, we shortly introduce the formal model from [35]. We start with a standard small-step operational semantics (similar to FJ [20]) that reduces configurations ζ which consist of heap and stack. The reduction relation $\zeta \overset{\gamma}{\rightsquigarrow} \zeta'$ is a labelled relation where γ is an input or output label (see Sect. 2.1) for steps that lead to a change of control or the special hidden label τ otherwise. We say that X *controls execution* if code of X is executed. An interaction is a change of control. Labels record changes of control. An interaction trace is a finite sequence of labels. Interaction is considered from the viewpoint of the library. Input labels (marked by ↓) express a change of control from the context to the library; output labels (marked by ↑) express a change from the library to the context. There are input and output labels for method invocation and return, as well as labels for well-formed and abrupt program termination. The labels for method invocation and return include the parameter and result values together with their *abstracted types* (explained later).

To be able to generate the traces and to realize the MGC, we augment configurations in a way to make explicit which parts belong to the library and which parts belong to the program context. This means that (1) we tag stack frames whether the containing code originates from the program context or the library, (2) we tag objects in the heap whether they have been created by code of the program context or the library (origin flag), and (3) we tag objects whether they have been exposed or are internal. Objects are tagged as internal when they are created. If they are passed from the library to the program context or vice-versa (i.e., appear in the trace), however, they get retagged as exposed. At the end of a program run, the objects that are marked as exposed are exactly those that appeared in the trace.

We can then generate an (interaction) label if control flow passes from the program context to the library or vice-versa. The label records all relevant information, namely the direction, method call / return, name of method, objects exposed and an abstraction of types of exposed objects, which we have omitted in our example trace.

To compare traces of different library implementations in a way that is independent from concrete contexts, we abstract from types declared in the context (e.g., IntObs). Similarly, local types should not appear in the labels, because different libraries might use different local types, i.e, renaming the ObsIter class in Fig. 2 should not matter to program contexts.

Types in labels are *abstracted* to a representation that only preserves the information (1) which public supertypes of the type belong to the library and (2) which of their methods are not overridden by the context. The reason for (1) is that these are the types of the library that can be used in cast expressions in the context. Based on the label, it becomes clear which cast expressions will succeed and which will not. In our example, the type of the ObsIter objects is abstracted to the Iterator type. The reason for (2) is that, based on the label, we know the methods that, if invoked with the object as receiver, lead to changes of control. As a library defines a finite number of types, there are only a finite number of abstracted types that can occur in traces with the library. This set of abstracted types can directly be derived from the type declarations of the library.

We construct the most general context $mgc(X)$ based on the library implementation X that enables all possible interactions that X can engage in. The context $mgc(X)$ represents exactly all contexts that X can have. To represent the MGC, we have to extend the language with a non-deterministic choice operator. Whenever a program execution is in the MGC, it then executes (an arbitrary long) sequence of the following actions: creating new objects, performing cross-border method calls / returns using objects created in the context or exposed[2] objects, or program termination.

In order to have a finite representation of all possible classes in the context, we do a power set construction based on the classes of the library under consideration. For each abstracted type, a class is created such that whenever an object of this class appears in the trace, then it has the corresponding abstracted type. Full details for this construction can be found in [35]. As the MGC is a program context, it also has a main class with a main method.

Using the MGC, we can now compare two libraries. Two libraries have the same observational behavior if they exhibit the same set of traces when they are run with the MGC (full abstraction result from [35]). Note that the set of traces is prefix-closed. Furthermore, input labels generated by the MGC depend solely on the trace history (i.e., the labels that occurred so far). In the following, when we talk about the state of a library, we implicitly mean the state of the program that is composed of the library code and the MGC.

[2]Under the assumption that the most general context can realize arbitrary sharing of objects.

3. PROVING EQUIVALENCE

Proving equivalence between two libraries using the trace-based semantics is feasible, as we have a direct correspondence between the operational steps in the program code and the traces. In this section, we illustrate how this correspondence can be exploited for proving equivalences.

In the following, we often want to give a correspondence between different states of the first library and the second library. In order to deal with the non-deterministic choice of fresh object identifiers, we introduce (object) renamings. A renaming (denoted by ρ) is a bijective relation on object identifiers.

We describe the properties that hold between states (i.e. runtime configurations) of *similar* runs. Two runs are similar if they lead to similar traces and the MGC takes the same actions (i.e. makes the same non-deterministic choices) in both runs. Two traces are similar if they are equal modulo a renaming.

Observable states. In our methodology the observable states are those where the coupling invariant, which describes the relation between both libraries, must hold. These states are exactly those where the MGC controls execution.

Correspondence relation ρ. At observable states, the following properties hold between the states of the two libraries under consideration if they have a similar trace prefix. There is a renaming ρ from the exposed objects of the first configuration to the exposed objects of the second (i.e. the objects occurring in the trace). We call this renaming a *correspondence relation*. For objects related by ρ (i.e. corresponding objects), the heap entries match in the following way. Corresponding objects have the same origin (created by the context or by the library). The dynamic types of corresponding objects are equal if they are created by the MGC. Otherwise, they have the same abstracted types (see Sect. 2.1). Similarly, there is also a renaming between the (internal) objects that are created by the MGC. This property between the runtime configurations can be exploited to relate two implementations of a library, namely, we can talk about corresponding objects, which are those, that appear at the same positions in both traces.

We apply a fairly standard technique for proving equivalence to our trace-based setting. From the fully abstract semantics in [35] we get the property that there is a relation between corresponding observable states of two libraries whenever these libraries are behaviorally equivalent. This relation is called a *coupling* relation and has the form of an environmental bisimulation [23]. Furthermore, there is a correspondence relation ρ as previously described over these states.

Coupling invariant. The user has to formulate the coupling relation over both program states that holds in all observable states, which is also called a *coupling invariant*. The coupling relation between the states of both libraries can be described with the help of the correspondence relation ρ. We then need to prove for related inputs and coupled states that the next outputs are also related and the states coupled. We also need to make sure that the coupling holds at the initial state of the program (empty heap and stacks) and that steps in the MGC do not destroy the invariant, which is usually a very simple property to check (and can be deduced directly from a few syntactic restrictions on the specifications of the coupling). A superset of the shapes (i.e., types and method names) of all possible input labels can be derived from the code of the library, e.g., the public methods.

Example. In the following, we describe the coupling invariant for the Cell example in Fig. 1 using the correspondence relation ρ. For all corresponding objects $(o_1, o_2) \in \rho$ that have the dynamic type Cell or a subtype thereof and where the value of the field $o_2.f$ is true, the values that are stored in the fields $o_1.c$ and $o_2.c_1$ are either both null or corresponding objects, i.e. $(o_1.c, o_2.c_1) \in \rho$. Similarly, if the value of the field $o_2.f$ is false, then $(o_1.c, o_2.c_2) \in \rho$ or these fields are both null.

For this particular example, the input labels are of the form call o.get()中 and call o.set(v)中. The main proof obligation to check equivalence between the two Cell library implementations has the following form. We consider related inputs (e.g. call o_1.get()中 and call o_2.get()中 if $(o_1, o_2) \in \rho$) in states which are coupled (i.e., where the coupling invariant holds). We then have to prove that the states right after the next change in control are also coupled and the generated (output) labels are related.

From traces to code. An important part is to establish a relation between the input labels and the program code. If we have a call input label, we need to find out what the possible targets of such a call are, i.e. the method bodies in the library resulting from a method dispatch that leads to such a label. Similarly, if we have a return input label, we need to find the places in the library where we can return to. This can be approximated based on the types that appear in the input/output labels. As the labels are directly based on the runtime information, we can give an inverse of the abstraction function that computes the abstracted types of the labels. This inverse is a relation, i.e., it associates multiple places in the library with a certain shape of message. We illustrate the relation using an example. The formal definition can directly be derived from the abstraction function in [35].

Example. Consider the public class C and the local class D as part of a library where the method m in D overrides m in C.

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

Consider a program context from which the class D is not accessible (i.e. defined in another package). If now an input label of the form call $o.m(...)$中 occurs where the (observable) type of o is C, then the program could, depending on the actual runtime type, either lead to a dispatch to the method m defined in C or in D (i.e. BODY1 or BODY2). Important to note is that this scenario can only occur if a D object is exposed under the C type, e.g. if there is a method of the following form in the library:

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

A simple static analysis in the likes of [16] can be used to statically determine the shape of messages in most cases. Remaining (open) cases have to be formulated as part of the coupling invariant. For example, the invariant may specify the property that there are no exposed objects of dynamic type D. In that case, input labels could never contain D objects and thus never dispatch to a method in D.

In the following, we describe how to encode (parts of) the verification conditions in Boogie. For simplicity, we only consider terminating methods and no recursive method calls.

4. BOOGIE MODELING

Boogie is usually used as an intermediate language to study correctness of programs with respect to specifications. It is targeted by a variety of source languages (Spec# [5], Dafny [26],

```
type Ref; const null: Ref;
type Field _; type Heap = <α>[Ref, Field α] α;
type Var _; type StackPtr = int;
type StackFrame = <α>[Var α] α;
type Stack = [StackPtr] StackFrame;
type Bij = [Ref, Ref] bool;
var O1: Heap where ...; var O2: Heap where ...;
var S1: Stack where ...; var S2: Stack where ...;
var sp1: StackPtr; var sp2: StackPtr;
var ρ: Bij where ...;
const unique alloc, exposed, createdByCtxt: Field bool;
const unique this: Var Ref;
function RelNull(r1:Ref, r2:Ref, ρ:Bij) returns (bool) {
  (r1 == null ∧ r2== null) ∨ (r1 ≠ null ∧ r2≠ null ∧ ρ[r1,r2])
}
procedure {:inline 1} Update(r1:Ref, r2:Ref) modifies O1,O2,ρ; {
  assert RelNull(r1,r2,ρ) ∨ ( ¬(∃ r:Ref • ρ[r1,r]) ∧ ¬(∃ r:Ref • ρ
      [r,r2]) );
  if (r1 ≠ null ∧ r2≠ null) { O1[r1,exposed] := true; O2[r2,exposed]
      := true; ρ[r1,r2] := true; }
}
```

Figure 4: Boogie prelude (simplified)

Chalice [28]...). The purpose of Boogie is to facilitate the generation of verification conditions for today's 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 [6] can then generate logical formulas which are fed to theorem provers.

We model the libraries under investigation and the proof obligations in Boogie. We use the newer version of the Boogie language, namely Boogie version 2 [27], which has a richer type system than the previous version. We explain features of the Boogie language along with the modeling. We omit many details (e.g. typing aspects or exceptional control flow) in order to produce readable Boogie code. We start by modeling the prelude that encodes the basic (runtime) entities for the object-oriented languages we consider and then proceed with an encoding of our Cell example.

Prelude. A simplified version of our Boogie prelude is given in Fig. 4. We declare a general type Ref which represents values of a reference type, i.e., object identifiers and the null value. We then declare null as a symbolic constant of this type. Similar as for Spec#, heaps are modeled as a mapping from object identifiers and field names to values. As we want the possibility to store not only values of reference types but also the Boogie builtin types int and bool, we define heaps as polymorphic maps. The type of the field value then depends on the field name used. Stacks are modeled as a mapping from integers to stack frames and a stack pointer that is used to refer to the top of the stack. Stack frames are modeled as polymorphic mappings from (local) variable names to values. We also define renamings (i.e., bijective object relations), which are modeled as binary predicates.

We can then define the configurations for the libraries which we consider, namely the runtime configurations of the two libraries (with most general context) and the bijection ρ between objects of the two. Both heaps and stacks and the correspondence relation ρ are modeled as variables.

Beside the regular fields that have to be defined for specific programs, we have a number of ghost fields that say whether objects are allocated, what the class type of the object is, whether objects are exposed and whether they have been created by code of the library or the program context. The fields are annotated with the modifier unique to denote that these constants (of same type) are distinct.

We only consider well-formed heaps and stacks and a well-formed correspondence relation, which we state using helper predicates. The wellformedness conditions follow directly from [35], e.g. the wellformedness condition on the correspondence relation states (among others) that it exactly relates the exposed objects of both heaps. As throughout the examples and Java in general, we often have to deal with null, we define the predicate RelNull that yields whether two reference values are related or both null. In order to easily update the correspondence relation ρ, we add the procedure Update that ensures that the bijection property of ρ is preserved. This procedure serves purely as a macro as we instruct Boogie to inline the body of the procedure at call sites.

Example. In the following, we use the Cell example to illustrate a possible encoding of the library code and the proof obligations. We first encode the invariant that was informally described in Sect. 3. The coupling invariant is defined as a predicate over both program configurations and the correspondence relation.

```
const unique f: Field bool; const unique c,c1,c2: Field Ref;
function Inv(...) returns (bool) {
  ( ∀o1,o2:Ref • ρ[o1,o2] ∧ O2[o2,f] ⇒
                          RelNull(O1[o1,c], O2[o2,c1], ρ) ) ∧
  ( ∀o1,o2:Ref • ρ[o1,o2] ∧ ¬O2[o2,f] ⇒
                          RelNull(O1[o1,c], O2[o2,c2], ρ) )
}
```

The proof obligations are as follows. If two related input labels arrive and the invariant holds, then both libraries must reply with related output labels and preserve the invariant. For this example, we only have to consider two kinds of input labels, namely calls of the get and set methods. The output labels are always return labels. We start by considering related input labels of the form o_1.get() and o_2.get().

We assume that the invariant holds (line 2) and that the method call receivers are related, i.e., $\rho[\mathcal{S}_1[sp_1][this],\mathcal{S}_2[sp_2][this]]$ holds (Note that $\mathcal{S}_1[sp_1][this]$ denotes the object identifier that is stored on top of the first stack under the local variable name this). We then want to prove that, after the execution of both get method bodies, the result values can be safely added to the correspondence relation and the invariant still holds. As Boogie has no concept of "executing" code in parallel, we "run" one method body after the other. We encode the body of the first get method in line 3 and the body of the second get method in lines 4 to 8.

```
const unique res: Var Ref;                                          1
assume Inv(...) ∧ ρ[S1[sp1][this],S2[sp2][this]];                   2
S1[sp1][res] := O1[S1[sp1][this],c];                                3
if (O2[S2[sp2][this],f]) {                                          4
  S2[sp2][res] := O2[S2[sp2][this],c1];                             5
} else {                                                            6
  S2[sp2][res] := O2[S2[sp2][this],c2];                             7
}                                                                   8
call Update(S1[sp1][res],S2[sp2][res]);                             9
assert Inv(...);                                                    10
```

In order to complete the proof, we must also verify that the set methods and object creation (in the program context) do not destroy the invariant, which we omit here for lack of space.

5. CONCLUSION AND FUTURE WORK

In this short paper, we have presented how to use the fully abstract trace-based semantics defined in [35] to prove equivalence of class library implementations. We have employed the technique to verify a number of classical examples in the literature using Boogie. 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.

The Boogie intermediate verification language was very useful for rapid validation of the verification approach. In the future, we would like to encode more properties, namely the full typing information of the libraries. We are currently working on an tool to generate the Boogie model of the libraries automatically (for a Java subset).

We would also like to study how to connect the given approach with more traditional verification approaches based on specifications (which we believe to be complementary). Although there remain open questions, our initial experiments show that mechanical (semi-automatic) verification of equivalence for structurally similar OO libraries is feasible.

Acknowledgements. We were able to encode our concepts in a concise way and get first verification results very fast, which we attribute to the quality of the Boogie language and the Boogie verifier tool. We thank David Naumann for pointing out related work and Peter Müller for commenting on our Boogie encoding. Finally we thank the FTfJP 2012 reviewers for their comments.

6. 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 2004*. Ed. by Liu, Z., Araki, K. Vol. 3407. LNCS. Springer, Heidelberg, 2004, pp. 37–51.

[2] Banerjee, A., Naumann, D. A. "Ownership confinement ensures representation independence for object-oriented programs". In: *Journal of the 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., DeLine, R., Fähndrich, M., Leino, K. R. M., Schulte, W. "Verification of Object-Oriented Programs with Invariants". In: *Journal of Object Technology* 3.6 (2004), pp. 27–56.

[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] 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 2005*. Vol. 4111. LNCS. Springer-Verlag, 2006, pp. 364–387.

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

[8] Benton, N., Koutavas, V. *A Mechanized Bisimulation for the Nu-Calculus*. 2008.

[9] Cook, W. R. "A Denotational Semantics of Inheritance". PhD thesis. Brown University, 1989.

[10] Damiani, F., Poetzsch-Heffter, A., Welsch, Y. "A Type System for Checking Specialization of Packages in Object-Oriented Programming". In: SAC (OOPS). 2012.

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

[12] Drossopoulou, S., Francalanza, A., Müller, P., Summers, A. "A Unified Framework for Verification Techniques for Object Invariants". In: *ECOOP*. LNCS. 2008, pp. 412–437.

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

[14] Filipovic, I., O'Hearn, P. W., Rinetzky, N., Yang, H. "Abstraction for concurrent objects". In: *Theor. Comput. Sci* 411.51-52 (2010), pp. 4379–4398.

[15] 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.

[16] Geilmann, K., Poetzsch-Heffter, A. "Modular Checking of Confinement for Object-Oriented Components using Abstract Interpretation". In: *International Workshop on Aliasing, Confinement and Ownership*. 2011.

[17] Godlin, B., Strichman, O. "Regression verification". In: *DAC*. ACM, 2009, pp. 466–471.

[18] Gotsman, A., Yang, H. "Liveness-Preserving Atomicity Abstraction". In: *ICALP (2)*. Ed. by Aceto, L., Henzinger, M., Sgall, J. Vol. 6756. LNCS. Springer, 2011, pp. 453–465.

[19] Hennessy, M., Milner, R. "On Observing Nondeterminism and Concurrency." In: *ICALP*. 1980, pp. 299–309.

[20] Igarashi, A., Pierce, B. C., Wadler, P. "Featherweight Java: a minimal core calculus for Java and GJ". In: *ACM Transactions on Programming Languages and Systems* 23.3 (2001), pp. 396–450.

[21] 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.

[22] Jeffrey, A., Rathke, J. "Java Jr.: Fully Abstract Trace Semantics for a Core Java Language". In: *ESOP*. 2005, pp. 423–438.

[23] Koutavas, V., Levy, P. B., Sumii, E. "From Applicative to Environmental Bisimulation". In: *Electr. Notes Theor. Comput. Sci* 276 (2011).

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

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

[26] Leino, K. R. M. "Dafny: An Automatic Program Verifier for Functional Correctness". In: *LPAR-16*. LNCS. Springer-Verlag, 2010.

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

[28] 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.

[29] Meyer, B. *Object-Oriented Software Construction*. Second. Prentice-Hall, 1997.

[30] Müller, P., Poetzsch-Heffter, A., Leavens, G. T. "Modular invariants for layered object structures". In: *Sci. Comput. Program.* 62.3 (2006), pp. 253–286.

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

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

[33] Sumii, E., Pierce, B. C. "A Bisimulation for Dynamic Sealing". In: *Theoretical Computer Science* 375 (2007).

[34] Sumii, E., Pierce, B. C. "A Bisimulation for Type Abstraction and Recursion". In: *Journal of the ACM* 54 (2007).

[35] Welsch, Y., Poetzsch-Heffter, A. "Full Abstraction at Package Boundaries of Object-Oriented Languages". In: *SBMF 2011*. LNCS. Springer, 2011, pp. 28–43.