Verifying Eiffel Programs with Boogie

Julian Tschannen, Carlo A. Furia, Martin Nordio, and Bertrand Meyer

Chair of Software Engineering, ETH Zurich, Switzerland
{firstname.lastname}@inf.ethz.ch

Abstract. Static program verifiers such as Spec#, jStar, and VeriFast define the state of the art in automated functional verification techniques. The next open challenges are to make verification tools usable even by programmers not fluent in formal techniques. This paper presents AutoProof, a verification tool that translates Eiffel programs to Boogie and uses the Boogie verifier to prove them. In an effort to make AutoProof usable with real programs, the tool fully supports advanced object-oriented features including polymorphism and inheritance, and function objects. Support for more features such as exception handling is underway. It also adopts simple strategies to reduce the extra annotations needed to verify programs (e.g., frame conditions). The paper illustrates the main features of AutoProof's translation and demonstrates them with examples and a case study.

1 Usable Verification Tools

It is hard to overstate the importance of *tools* for software verification: tools have practically demonstrated the impact of general theoretical principles, and they have brought automation into significant parts of the verification process. Program provers, in particular, have matured to the point where they can handle complex properties of real programs. For example, provers based on Hoare semantics—e.g., Spec# [2] and ESC/Java [5]—support models of the heap to prove properties of object-oriented applications; other tools using separation logic—e.g., jStar [4] and VeriFast [7]—can reason about complex usages of pointers, such as in the visitor, observer, and factory design patterns. The experience gathered so far has also outlined some design principles, which buttress the development on new, improved verification tools; the success of the Spec# project, for example, has shown the value of using intermediate languages (Boogie [8], in the case of Spec#) to layer a complex verification process into simpler components, which can then be independently improved and reused across different projects.

The progress of verification tools is manifest, but it is still largely driven by challenge problems and examples. While case studies will remain important, verification tools must now also become more practical and *usable* by "lay" programmers. In terms of concrete goals, prover tools should support the complete semantics of their target programming language, and they should require a minimal annotational effort besides writing ordinary pre and postconditions of routines (methods).

The present paper describes AutoProof, a static verifier for Eiffel programs that makes some progress towards these goals of increased usability. AutoProof translates Eiffel programs annotated with contracts (pre and postconditions, class invariants, intermediate assertions) into Boogie programs. The translation currently handles sophisticated language features such as exception handling and function objects (called *agents* in Eiffel parlance, and *delegates* in C#). To reduce the need for additional annotations, AutoProof includes simple syntactic rules to generate standard frame conditions from postconditions, so that programmers have to write down explicit frame conditions only in the more complex cases.

This paper outlines the translation of Eiffel programs into Boogie, focusing on the most original features such as exception handling (which is peculiarly different in Eiffel, as opposed to other object-oriented languages such as Java and C#) and the generation of simple frame conditions. The translation of more standard constructs is described elsewhere [17]. At the time of writing, AutoProof still does not implement the support of exceptions, but the inclusion of the translation described in the paper is underway. The paper also reports a case study where we automatically verify several Eiffel programs, exercising different language features, with AutoProof. AutoProof is part of EVE (Eiffel Verification Environment), the research branch of the EiffelStudio integrated development environment, which integrates several verification techniques and tools. EVE is distributed as free software and freely available for download at: `http://se.inf.ethz.ch/research/eve/`

Outline. Section 2 presents the Boogie translation of Eiffel's exception handling primitives; Section 3 describes a translation of conforming inheritance that supports polymorphism; Section 4 sketches other features of the translation, such as the definition of "default" frame conditions; Section 5 illustrates the examples verified in the case study; Section 6 presents the essential related work, and Section 7 outlines future work.

2 Exceptions

Eiffel's exception handling mechanism is different than most object-oriented programming languages such as C# and Java. This section presents Eiffel's mechanism (Section 2.1), discusses how to annotate exceptions (Section 2.2), and describes the translation of Eiffel's exceptions to Boogie (Section 2.3) with the help of an example (Section 2.4).

2.1 How Eiffel Exceptions Work

Eiffel exception handlers are specific to each routine, where they occupy an optional **rescue** clause, which follows the routine body (**do**). A routine's **rescue** clause is ignored whenever the routine body executes normally. If, instead, executing the routine body triggers an exception, control is transferred to the **rescue** clause for exception handling. The exception handler should try to restore the state to conditions where the routine can execute normally. To this end, the body can run more than once, according to the value of an implicit variable **Retry**, local to each routine: when the execution of the handler terminates, if **Retry** has value **True** the routine body is run again, otherwise **Retry** is **False** and the pending exception propagates to the **rescue** clause of the caller routine.

Figure 1 illustrates the Eiffel exception mechanism with an example. The routine *attempt_transmission* tries to transmit a message by calling *unsafe_transmit*; if the latter routine terminates normally, *attempt_transmission* also terminates normally without executing the **rescue** clause. On the contrary, an exception triggered by *unsafe_transmit* transfers control to the **rescue** clause, which re-executes the body for *max_attempts* times; if all the attempts fail to execute successfully, the attribute (field) *failed* is set and the exception propagates.

```
attempt_transmission (m: STRING)
    local
        failures : INTEGER
    do
        failed := False
        unsafe_transmit (m)
    rescue
        failures := failures + 1
        if failures < max_attempts then
            Retry := True
        else
            failed := True
        end
    end
```

Fig. 1. An Eiffel routine with exception handler.

2.2 Specifying Exceptions

The postcondition of a routine with **rescue** clause specifies the program state both after normal termination and when an exception is triggered. The two post-states are in general different, hence we introduce a global Boolean variable *ExcV*, which is **True** iff the routine has triggered an exception. Using this auxiliary variable, specifying postconditions of routines with exception handlers is straightforward. For example, the postcondition of routine *attempt_transmission* in Figure 1 should say that *failed* is **False** iff the routine executes normally:

```
attempt_transmission (m: STRING)
    ensure
        ExcV implies failed
        not ExcV implies not failed
```

The example also shows that the execution of a **rescue** clause behaves as a loop: a routine r with exception handler r **do** s_1 **rescue** s_2 **end** behaves as the loop that first executes s_1 unconditionally, and then repeats $s_2 \,;\, s_1$ until s_1 triggers no exceptions or **Retry** is **False**. To reason about such implicit loops, we introduce a *rescue invariant* [14]; the rescue invariant holds after the first execution of s_1, and after each execution of $s_2 \,;\, s_1$. A reasonable rescue invariant of routine *attempt_transmission* is:

```
rescue invariant
    not ExcV implies not failed
    (failures < max_attempts) implies not failed
```

2.3 Eiffel Exceptions in Boogie

The auxiliary variable *ExcV* becomes a global variable in Boogie, so that every assertion can reference it. The translation also introduces an additional precondition $ExcV =$ **false** for every translated routine, because normal calls cannot occur when exceptions are pending. Then, a routine with body s_1 and rescue clause s_2 becomes in Boogie:

```
$\nabla(s_1, excLabel)$
excLabel: while (ExcV) {
              ExcV := false;
              Retry := false;
              $\nabla(s_2, endLabel)$
              if ($\neg$Retry) {ExcV := true; goto endLabel} ;
              $\nabla(s_1, excLabel)$
          }
endLabel:
```

where $\nabla(s, l)$ denotes the Boogie translation $\nabla(s)$ of the instruction s, followed by a jump to label l if s triggers an exception:

$$\nabla(s, l) \;=\; \begin{cases} \nabla(s', l)\,;\, \nabla(s'', l) & \text{if } s \text{ is the compound } s'\,;\, s'' \\ \nabla(s)\,;\; \textbf{if}\;\; (ExcV)\,\{\textbf{goto}\; l;\} & \text{otherwise} \end{cases}$$

Therefore, when the body s_1 triggers an exception, *ExcV* is set and the execution enters the rescue loop. On the other hand, an exception that occurs in the body of s_2 jumps out of the loop and to the end of the routine.

2.4 An Example of Exception Handling in Boogie

Figure 2 shows the translation of the example in Figure 1. To simplify the presentation, Figure 2 renders the attributes *max_attempts*, *failed*, and *transmitted* (set by *unsafe_transmit*) as variables rather than locations in a heap map. The loop in lines 22–36 maps the loop induced by the **rescue** clause, and its invariant (lines 23–24) is the rescue invariant.

```
var max_attempts: int; var failed : bool; var transmitted:bool;

procedure unsafe_transmit (m: ref);
      requires ExcV = false;
      modifies ExcV, transmitted;
      ensures ExcV ⟺ ¬ transmitted;

procedure attempt_transmission (m: ref);
      requires ExcV = false;
      modifies ExcV, transmitted, max_attempts, failed;
      ensures ExcV ⟺ failed ;

implementation attempt_transmission (m: ref)
{
  var failures : int;
  var Retry: bool;
  entry:
      failures := 0; Retry := false;
      failed := false;
      call unsafe_transmit (m); if (ExcV) { goto excL; }
      excL:
          while (ExcV)
              invariant ¬ExcV ⟹ ¬ failed ;
              invariant (failures < max_attempts) ⟹ ¬ failed ;
          {
              ExcV := false; Retry := false;
              failures := failures + 1;
              if ( failures < max_attempts) {
                  Retry := true;
              } else {
                  failed := true;
              }
              if (¬ Retry) {ExcV := true; goto endL;}
              failed := false
              call unsafe_transmit (m); if (ExcV) { goto excL; }
          }
      endL: return;
}
```

Fig. 2. Boogie translation of the Eiffel routine in Figure 1.

3 Inheritance and Polymorphism

The redefinition of a routine r in a descendant class can *strengthen* r's original postcondition by adding an **ensure then** clause, which conjoins the postcondition in the precursor. The example in Figure 3 illustrates a difficulty occurring when reasoning about postcondition strengthening in the presence of polymorphic types. The deferred (abstract) class *EXP* models nonnegative integer expressions and provides a routine *eval* to evaluate the value of an expression object; even if *eval* does not have an implementation in *EXP*, its postcondition specifies that the result is always nonnegative, and the value of attribute *last_value* is set as side effect. Classes *CONST* and *PLUS* respectively specialize *EXP* to represent integer (nonnegative) constants and addition. Class *ROOT* is a client of the other classes, and its *main* routine attaches an object of subclass *CONST* to a reference with static type *EXP*, thus exploiting polymorphism.

```
deferred class EXP
feature
    last_value : INTEGER
    eval
        deferred
        ensure
            last_value ≥ 0
        end
end
```

```
class PLUS inherit EXP feature
    left , right: EXP
    eval do
            left . eval; right . eval
            last_value := left . last_value +
                              right . last_value
        ensure then
            last_value = left . last_value +
                             right . last_value
        end
invariant
    no_aliasing: left ≠ right ≠ Current
end
```

```
class CONST inherit EXP
feature
    value: INTEGER
    eval
        do
            last_value := value
        ensure then
            last_value = value
        end
invariant
    positive_value : value ≥ 0
end
```

```
class ROOT
feature
    main
        local
            e: EXP
        do
            e := create {CONST}.make (5);
            e. eval
            check e. last_value = 5 end
        end
end
```

Fig. 3. Nonnegative integer expressions.

With the Boogie translation of polymorphic assignment implemented in Spec#, we can verify the assertion **check** *e.last_value* = 5 **end** in class *ROOT* only if *eval* is declared *pure*; *eval* is, however, not pure. The Spec# methodology selects the pre and postconditions according to static types for non-pure routines: the call *e.eval* only establishes *e.last_value* ≥ 0, not the stronger *e.last_value* = 5 that follows from *e*'s dynamic type *CONST*.

3.1 Polymorphism in Boogie

The Boogie translation implemented in AutoProof can handle polymorphism appropriately even for non-pure routines; it does so by extending the methodology for pure routines [3,11]. The rest of the section discusses how to translate postconditions of redefined routines in a way that accommodates polymorphism and while still supporting modular reasoning. Eiffel also supports *weakening of preconditions* in redefined routines; the translation to Boogie handles it similarly as for postconditions (we do not discuss it for brevity).

The translation of the postcondition of a routine r of class X with result type T (if any) relies on an auxiliary function *post.X.r*:

function *post.X.r*(*h1*, *h2*: *HeapType*; *c*: **ref**; *res*: *T*) **returns** (**bool**);

which predicates over two heaps (the pre and post-states in r's postcondition), a reference c to the current object, and the result *res*. r's postcondition in Boogie references the function *post.X.r*, and it includes the translation $\nabla_{post}(X.r)$ of r's postcondition clause syntactically declared in class X:

procedure *X.r* (*Current*: **ref**) **returns** (*Result*: *T*);
  free ensures *post.X.r*(*Heap*, **old**(*Heap*), *Current*, *Result*);
  ensures $\nabla_{post}(X.r)$

post.X.r is a **free ensures**, hence it is ignored when proving r's implementation and is only necessary to reason about usages of r.

The function *post.X.r* holds only for the type X; for each class Y which is a descendant of X (and for X itself), an axiom links r's postcondition in X to r's strengthened postcondition in Y:

axiom ($\forall$ *h1*, *h2*: *HeapType*; *c*: **ref**; *r*: *T* $\bullet$
    h1[*c*, $**type**] <: $Y \Longrightarrow$(*post.X.r*(*h1*, *h2*, *c*, *r*) $\Longrightarrow \nabla_{post}(Y.r)$));

The ghost field $**type** of the heap map stores dynamic types: *h1*[*c*,$**type**] is the type of reference c, hence the postcondition predicate *post.X.r* implies an actual postcondition $\nabla_{post}(Y.r)$ according to c's dynamic type.

3.2 An Example of Polymorphism with Postconditions

Figure 4 shows the essential parts of the Boogie translation of the example in Figure 3. The translation of routine *eval* in lines 3–6 references the function *post.EXP.eval*; the axioms in lines 8–13 link such function to r's postcondition in *EXP* (lines 8–10) and to the additional postcondition introduced in *CONST* for the same routine (lines 11–13). The rest of the figure shows the translation of the client class *ROOT*.

```
function post.EXP.eval(h1, h2: HeapType; c: ref) returns (bool);

procedure EXP.eval(current: ref);
  free ensures post.EXP.eval(Heap, old(Heap), current, result);
  ensures Heap[current, last_value] ≥ 0;
  // precondition and frame condition omitted

axiom (∀ h1, h2: HeapType; o: ref •
  h1[o, $type] <: EXP ⟹
    (post.EXP.eval(h1, h2, o) ⟹ (h1[o, last_value] ≥ 0)) );
axiom (∀ h1, h2: HeapType; o: ref •
  h1[o, $type] <: CONST ⟹
    (post.EXP.eval(h1, h2, o) ⟹ h1[o, last_value] = h1[o, value]) );

implementation ROOT.main (Current: ref) {
    var e: ref;
  entry:
      // translation of create {CONST} e.make (5)
    havoc e;
    assume Heap[e, $allocated] = false;
    Heap[e, $ allocated] := true;
    Heap[e, $type] := CONST;
    call CONST.make(e, 5);
      // translation of e.eval
    call EXP.eval(e);
      // translation of check e.last_value = 5 end
    assert Heap[e, last_value] = 5;
    return;
}
```

Fig. 4. Boogie translation of the Eiffel classes in Figure 3.

4 Other Features

This section briefly presents other features of the Eiffel-to-Boogie translation.

4.1 Default Frame Conditions

Frame conditions are necessary to reason modularly about heap-manipulating programs, but they are also an additional annotational burden for programmers. In several simple cases, however, the frame conditions are straightforward and can be inferred syntactically from postconditions. For a routine r, let mod_r denote the set of attributes mentioned in r's postcondition; mod_r is a set of (*reference*, *attribute*) pairs. The translation of Eiffel to Boogie implemented in AutoProof assumes that every attribute in mod_r may be modified (that is, mod_r is r's frame), whereas every other location in the heap is not modified. Since every non-pure routine already includes the whole *Heap* map in its **modifies** clause, the frame condition becomes the postcondition:

$$\textbf{ensures}\ (\forall\ o\colon \textbf{ref},\ f\colon\ \mathit{Field}\ \bullet\ (o,\ f) \notin \mathit{mod}_r \implies \mathit{Heap}[o,\ f]\ =\ \textbf{old}(\mathit{Heap}[o,\ f]))\,;$$

Such default frame conditions work well for routines mentioning only attributes of primitive types. For routines that manipulate composite data, such as arrays or lists, the default frame conditions are too coarse-grained, hence programmers have to supplement the default with more detailed annotations. Extending the support for automatically generated frame conditions is part of future work.

4.2 Routines Used in Contracts Pure by Default

The translation of routines marked as *pure* generates the frame condition **ensures** *Heap* =**old**(*Heap*) which specifies that the heap is unchanged. AutoProof implicitly assumes that every routine used in contracts is pure, and the translation reflects this assumption and checks its validity. While the Eiffel language does not require routines used in contract to be pure, it is a natural assumption which holds in practice most of the times, because the behavior of a program should not rely on whether contracts are evaluated or not. Therefore, including this assumption simplifies the annotational burden and makes using AutoProof easier in practice.

4.3 Agents (Function Objects)

The translation of Eiffel to Boogie supports *agents* (Eiffel's name for *function objects* or *delegates*). The translation introduces abstract predicates to specify routines that take function objects as arguments: some axioms link the abstract predicates to concrete specifications whenever an agent is initialized. The details of the translation of agents is described elsewhere [13].

5 Case Study

This section presents the results of a case study applying AutoProof to the verification of 11 programs listed in Table 1. For each example, the table reports its name, its size in number of classes and lines of code, the length (in lines of code) of the translation to Boogie, the time taken by Boogie to verify successfully the example, and the kind of Eiffel features mostly exercised by the example.

Example 1 is a set of routines presented in Meyer's book [12] when describing Eiffel's exceptions; Example 2 is a set of classes part of the EiffelStudio compiler runtime. To verify them, we extended the original contracts with postconditions to express the behavior when exceptions are triggered, and with rescue invariants (Section 2.2).[1] The most difficult part of verifying these example was inventing rescue invariants. Even when the examples are simple, the rescue invariants may be subtle, because they have to include clauses both for normal and for exceptional termination.

Examples 3–5 target polymorphism in verification. The *Expression* example is described in Section 3. The *Sequence* example models integer sequences with the deferred classes *SEQUENCE*, *MONOTONE_SEQUENCE*, and *STRICT_SEQUENCE*, and their effective descendants *ARITHMETIC_SEQUENCE* and *FIBONACCI_SEQUENCE*. The *Command* example implements the command design pattern with a deferred class *COMMAND* and effective descendants that augment the postcondition of *COMMAND*'s deferred routine *execute*. The encoding of inheritance described in Section 3 is accurate but it also significantly increases the size of the Boogie translation and correspondingly the time needed to handle it. Since a translation that takes dynamic types into account is not always necessary, future work will introduce an option to have AutoProof translating contracts solely based on the static type of references.

Examples 6–8 use agents and are the same examples as in [13]. The *Formatter* example illustrates the specification of functions taking agents as arguments; the *Archiver* example uses an agent with closed arguments; the *Calculator* example implements the command design pattern using agents rather than subclasses.

Examples 9–11 combine multiple features: a cell class that stores integer values; a counter that can be increased and decreased; a bank account class with clients. These examples demonstrate other features of the translation, such as the usage of default frame conditions.

The source code of the examples is available at `http://se.ethz.ch/people/tschannen/boogie2011_examples.zip`. The experiments ran on a Windows 7 machine with a 2.71 GHz dual core Intel Pentium processor and 2GB of RAM.

6 Related Work

Tools such as ESC/Java [5] and Spec# [2] have made considerable progress towards practical and automated functional verification. Spec# is an extension of

[1] As the implementation in AutoProof of translation of exceptions is currently underway, these two examples were translated by hand.

	Example name	Classes	LOC Eiffel	LOC Boogie	Time [s]	Feature
1.	Textbook OOSC2	1	106	481	2.33	Exceptions
2.	Runtime ISE	4	203	561	2.32	Exceptions
3.	Expression	4	134	752	2.11	Inheritance
4.	Sequence	5	195	976	2.28	Inheritance
5.	Command	4	99	714	2.14	Inheritance
6.	Formatter	3	120	761	2.23	Agents
7.	Archiver	4	121	915	2.07	Agents
8.	Calculator	3	245	1426	9.73	Agents
9.	Cell / Recell	3	154	905	2.09	General
10.	Counter	2	97	683	2.02	General
11.	Account	2	120	669	2.04	General
	Total	**35**	**1594**	**8843**	31.36	

Table 1. Examples automatically verified with AutoProof

C# with syntax to express preconditions, postconditions, class invariants, and non-null types. Spec# is also a verification environment that verifies Spec# programs by translating them to Boogie—also developed within the same project. Spec# works on significant examples, but it does not support every feature of C# (for example, exceptions and delegates are not handled). Spec# includes annotations to specify frame conditions, which make proofs easier but at the price of an additional annotational burden for developers. On the contrary, AutoProof tries to rely on standard annotations whenever possible, which impacts on the programs that can be verified automatically.

Spec# has shown the advantages of using an intermediate language for verification. Other tools such as Dafny [9] and Chalice [10], and techniques based on Region Logic [1], follow this approach, and they also rely on Boogie as intermediate language and verification back-end, in the same way as AutoProof does.

Separation logic [15] is an extension of Hoare logic with connectives that define separation between regions of the heap, which provides an elegant approach to reasoning about programs with mutable data structures. Verification environments based on separation logic—such as jStar [4] and VeriFast [7]—can verify advanced features such as usages of the visitor, observer, and factory design patterns. On the other hand, writing separation logic annotations requires considerably more expertise than using standard contracts embedded in the programming language; this makes tools based on separation logic more challenging to use by practitioners.

7 Future Work

AutoProof is a component of EVE, the Eiffel Verification Environment, which combines different verification tools to exploit their synergies and provide a uniform and enhanced usage experience, with the ultimate goal of getting closer to the idea of "verification as a matter of course".

Future work will extend AutoProof and improve its integration with other verification tools in EVE. In particular, the design of a translation supporting the expressive model-based contracts [16] is currently underway. Other aspects for improvements are a better inference mechanism for frame conditions and intermediate assertions (e.g., loop invariants [6]); a support for interactive prover as an alternative to Boogie for the harder proofs; and a combination of AutoProof with the separation logic prover also part of EVE [18].

References

1. A. Banerjee, D. A. Naumann, and S. Rosenberg. Regional logic for local reasoning about global invariants. ECOOP. Springer-Verlag, 2008.
2. M. Barnett, R. Leino, and W. Schulte. The Spec# Programming System: An Overview. In *CASSIS*, volume 3362 of *LNCS*, pages 49–69. Springer, 2004.
3. A. Darvas and K. R. M. Leino. Practical reasoning about invocations and implementations of pure methods. In *FASE*, LNCS. Springer-Verlag, 2007.
4. D. Distefano and M. J. Parkinson. jStar: Towards Practical Verification for Java. In *Proceedings of OOPSLA*, pages 213–226, 2008.
5. C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *PLDI*, pages 234–245. ACM, 2002.
6. C. A. Furia and B. Meyer. Inferring loop invariants using postconditions. In *Fields of Logic and Computation*, volume 6300 of *Lecture Notes in Computer Science*, pages 277–300. Springer, 2010.
7. B. Jacobs, J. Smans, and F. Piessens. A quick tour of the VeriFast program verifier. In *Proceedings of APLAS 2010*, 2010.
8. K. Rustan M. Leino. This is Boogie 2. Manuscript KRML 178, 2008.
9. K. Rustan M. Leino. Dafny: an automatic program verifier for functional correctness. LPAR'10. Springer-Verlag, 2010.
10. K. Rustan M. Leino and P. Müller. A basis for verifying multi-threaded programs. ESOP '09. Springer-Verlag, 2009.
11. K. R. M. Leino and P. Müller. Verification of equivalent-results methods. In *ESOP*, volume 4960 of *LNCS*, pages 307–321. Springer-Verlag, 2008.
12. B. Meyer. *Object-Oriented Software Construction.* Prentice Hall, second edition, 1997.
13. M. Nordio, C. Calcagno, B. Meyer, P. Müller, and J. Tschannen. Reasoning about Function Objects. In *Proceedings of TOOLS-EUROPE*, LNCS. Springer, 2010.
14. M. Nordio, C. Calcagno, P. Müller, and B. Meyer. A Sound and Complete Program Logic for Eiffel. In M. Oriol, editor, *TOOLS-EUROPE*, 2009.
15. P. W. O'Hearn, H. Yang, and J. C. Reynolds. Separation and information hiding. In *POPL '04*, pages 268–280, 2004.
16. N. Polikarpova, C. A. Furia, and B. Meyer. Specifying reusable components. In *Proceedings of VSTTE'10*, volume 6217 of *Lecture Notes in Computer Science*, pages 127–141. Springer, 2010.
17. J. Tschannen. Automatic verification of Eiffel programs. Master's thesis, Chair of Software Engineering, ETH Zürich, 2009.
18. S. van Staden, C. Calcagno, and B. Meyer. Verifying executable object-oriented specifications with separation logic. In *Proceedings of ECOOP'10*, volume 6183 of *Lecture Notes in Computer Science*, pages 151–174. Springer, 2010.