Automatic Verification of Advanced Object-Oriented Features: The AutoProof Approach

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#, Dafny, 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 discusses some techniques used in AutoProof, a verification tool that translates Eiffel programs to Boogie and uses the Boogie verifier to prove them. In an effort to be usable with real programs, AutoProof fully supports several advanced object-oriented features including polymorphism, inheritance, and function objects. AutoProof also adopts simple strategies to reduce the amount of annotations needed when verifying programs (e.g., frame conditions). The paper illustrates the main features of AutoProof's translation, including some whose implementation is underway, 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 [8]—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 [10], 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; they should require minimal annotational effort besides writing ordinary pre and postconditions of routines (methods); and they should give valuable feedback when verification fails.

The present paper describes some traits of 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 [25]. At the time of writing, AutoProof does not implement the translation of exceptions described in the paper, but its inclusion 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 available for download at:

`http://se.inf.ethz.ch/research/eve/`

A version of AutoProof is also integrated into CloudStudio [21] web-based multi-language integrated development environment, available online at:

`http://cloudstudio.ethz.ch/`.

Outline. Section 2 introduces the Boogie intermediate language to make the paper self-contained. Section 3 presents the Boogie translation of Eiffel's exception handling primitives; Section 4 describes a translation of conforming inheritance that supports polymorphism; Section 5 discusses a technique to verify function objects. Section 6 sketches other features of the translation, such as the definition of "default" frame conditions; Section 7 illustrates the examples verified in the case study; Section 8 presents the essential related work, and Section 9 outlines future work.

2 A Short Introduction to Boogie

Boogie is a language for verification [10], as well as an automated verifier that takes programs written in the Boogie language as input. AutoProof verifies Eiffel programs by translating them into the Boogie language and then by calling

the Boogie verifier on the translation. The Boogie verifier generates verification conditions for the input program, and supports different prover back-ends (e.g., Z3 and Simplify) to discharge them. For readers unfamiliar with Boogie, this section describes the essential features of the Boogie language used in the rest of the paper.

The Boogie language offers two kinds of constructs: a simple imperative modular programming language—used to translate the source program (Eiffel, in our case)—and a specification language based on first-order logic—used to define specification elements and background logic theories needed to support complex specifications.

Boogie's specification language is a typed first-order logic with arithmetic. The basic types include Booleans (**bool**) and mathematical (unbounded) integers (**int**); the type constructors support the definition of derived types. Line 1 in Figure 1 declares a new type *person*, which Boogie treats as a fresh sort for variables. The specification language supports the definition of global variables and constants, functions (in the sense of mathematical logic), and axioms. Line 2, for example, declares a global constant *eve* of type *person*. Lines 3 and 4 declare two functions *age* and *can_vote*. Lines 5 and 6 introduce two axioms about the declared items: *age* is defined as 23 for argument *eve*; and *can_vote* is true precisely for persons whose age is greater than or equal to 18.

```
1  type person;
2  const eve: person;
3  function age(p: person) returns (int);
4  function can_vote(p: person) returns (bool);
5  axiom (age(eve) = 23);
6  axiom (∀ p: person • can_vote(p) ⟺ age(p) ≥18);
```

Fig. 1. Some definitions in Boogie's specification language.

Boogie's programming language supports the definition of procedures. Each procedure has a signature, which may include a specification in terms of preconditions (**requires**), postconditions (**ensures**), and frame clauses (**modifies**). The specification clauses contain formulas in Boogie's specification language. Postconditions, in addition, supports the usage of the **old** keyword to evaluate expressions in the state *before* a procedure was called. Modifies clauses, instead, define a procedure's *frame*, that is the set of global variables the procedure may modify. Pre- and postconditions may be marked as **free**, which prevents the generation of proof obligations based on them: a **free** assertion is assumed to hold whenever convenient, but need not be checked when required.

Procedure implementations use standard imperative constructs (assignments, conditionals, loops, jumps, and procedure calls) with the usual semantics. To write nondeterministic programs, Boogie's programming language includes the **havoc** command, which assigns a nondeterministically chosen value to its argu-

ment variables. To constrain the effects of **havoc** and to express intermediate verification conditions, Boogie's programming language also offers **assert** and **assume** statements. Both take an arbitrary formula F as argument. The program state of every execution reaching an **assert** F must satisfy F; otherwise, verification fails. Conversely, the verification process can assume that F holds of the program state whenever an execution reaches **assume** F, which "shapes" the nondeterministic behavior when convenient.

Figure 2 shows the specification and implementation of a procedure *vote* to cast a vote, demonstrating Boogie syntax.

```
var votes: int;

procedure vote(p: person);
  requires can_vote(p);
  ensures votes = old(votes) + 1;
  modifies votes;

implementation vote(p: person) {
  votes := votes + 1
}
```

Fig. 2. A Boogie procedure *vote*: specification and implementation.

3 Exceptions

Eiffel's exception handling mechanism is different than most object-oriented programming languages such as C# and Java.[1] This section presents Eiffel's mechanism (Section 3.1), discusses how to annotate exceptions (Section 3.2), and describes the translation of Eiffel's exceptions to Boogie (Section 3.3) with the help of an example (Section 3.4).

3.1 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 will try to restore the object state to a condition where the routine can execute normally. To this

[1] In related work, we have formalized the semantics of Java exceptions [16] and compared it against Eiffel's [17].

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 3 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. 3. An Eiffel routine with exception handler.

3.2 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** if and only if 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 3 says that *failed* is **False** if and only if 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** after the execution of s_2 (in the latter case, s_1 is not executed anymore). To reason about such implicit loops, we introduce a *rescue invariant* [20,22]; the rescue invariant holds after the first execution of s_1 and after each execution of $s_2\,;\, s_1$. A suitable rescue invariant of routine *attempt_transmission* is:

rescue invariant
not *ExcV* **implies not** *failed*
(*failures* $<$ *max_attempts*) **implies not** *failed*

3.3 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, and adds ExcV to the modifies clause of every procedure. Then, a routine with body s_1 and rescue clause s_2 becomes in Boogie:

$\nabla(s_1, excLabel)$
excLabel: **while** (*ExcV*)
invariant $\nabla(I_{rescue})$;
{
ExcV := **false**;
Retry := **false**;
$\nabla(s_2, endLabel)$
if (¬*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.

The exception handling semantics is only superficially similar to having control-flow breaking instructions such as *break* and *continue*—available in languages other than Eiffel—inside standard loops: the program locations where the

control flow diverts in case of exception are implicit, hence the translation has to supply a conditional jump after every instruction that might trigger an exception. This complicates the semantics of the source code, and correspondingly the verification of Boogie code translating routines with exception handling.

3.4 An Example of Exception Handling in Boogie

Figure 4 shows the translation of the example in Figure 3. To simplify the presentation, Figure 4 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 24–38 maps the loop induced by the **rescue** clause, and its invariant (lines 25 and 26) is the rescue invariant.

4 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 5 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 evaluation always yields a nonnegative value stored in attribute *last_value*, which is set as side effect (see Section 6.1). 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. Similar issues occur when a descendant class *weakens* a some routine r's precondition with an **require else** clause.

The verification goal is proving that, after the invocation *e.eval* (in class *ROOT*), *eval*'s postcondition in class *CONST* holds, which subsumes the **check** statement in the caller. Reasoning about the invocation only based on the static type *EXP* of the target e guarantees the postcondition $last\_value \geq 0$, which is however too weak to establish that *last_value* is exactly 5.

Other approaches, such as Müller's [15], have targeted these issues in the context of Hoare logics, but they usually are unsupported by automatic program verifiers. In particular, 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 nonpure routines: the call $e.eval$ only establishes $e.last\_value \geq 0$, not the stronger $e.last\_value = 5$ that follows from e's dynamic type *CONST*, unless an explicit cast redefines the type *CONST*. The rest of the section describes the solution implemented in AutoProof, which handles contracts of redefined routines.

```
1  var max_attempts: int;
2  var failed :bool;
3  var transmitted:bool;
4
5  procedure unsafe_transmit (m: ref);
6        free requires ExcV = false;
7        modifies ExcV, transmitted;
8        ensures ExcV ⟺ ¬ transmitted;
9
10 procedure attempt_transmission (m: ref);
11       free requires ExcV = false;
12       modifies ExcV, transmitted, max_attempts, failed;
13       ensures ExcV ⟺ failed ;
14
15 implementation attempt_transmission (m: ref)
16 {
17   var failures : int;
18   var Retry: bool;
19   entry:
20        failures := 0; Retry := false;
21        failed := false;
22        call unsafe_transmit (m); if (ExcV) { goto excL; }
23        excL:
24            while (ExcV)
25                invariant ¬ExcV ⟹ ¬ failed ;
26                invariant (failures < max_attempts) ⟹ ¬ failed ;
27            {
28                ExcV := false; Retry := false;
29                failures := failures + 1;
30                if ( failures < max_attempts) {
31                    Retry := true;
32                } else {
33                    failed := true;
34                }
35                if (¬ Retry) {ExcV := true; goto endL;}
36                failed := false
37                call unsafe_transmit (m); if (ExcV) { goto excL; }
38            }
39       endL: return;
40 }
```

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

```
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. 5. Nonnegative integer expressions.

4.1 Polymorphism in Boogie

The Boogie translation implemented in AutoProof can handle polymorphism appropriately even for non-pure routines; it is based on a methodology for agents [19] and on a methodology for pure routines [3,12]. The rest of the section discusses how to translate postconditions and preconditions of redefined routines in a way that accommodates polymorphism, while still supporting modular reasoning.

Postconditions. 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 includes the translation $\nabla_{post}(X.r)$ of r's postcondition clause syntactically declared in class X:[2]

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 $\bullet$
  \$**type**(c) <: $Y \Longrightarrow$(*post.X.r*(*h1*, *h2*, c, r) $\implies$ $\nabla_{post}(Y.r)$));

The function \$**type** returns the type of a given reference; hence the postcondition predicate *post.X.r* implies an actual postcondition $\nabla_{post}(Y.r)$ according to c's dynamic type.

In addition, for each redefinition of r in a descendant class Z, the translation defines a fresh Boogie procedure $Z.r$ with corresponding postcondition predicate $post.Z.r$ and axioms for all of Z's descendants.

Preconditions. Eiffel also supports *weakening of preconditions*. Therefore, the precondition of a routine can also depend on the dynamic type. We use a similar translation as for the postcondition. Given a routine r of type X, a precondition predicate is generated and used in the signature of the generated Boogie procedure:

function *pre.X.r*(*h*: *HeapType*; c: **ref**) **returns** (**bool**);

which predicates over one heap and a reference c to the current object. r's precondition in Boogie references the function $pre.X.r$, and it includes the translation $\nabla_{pre}(X.r)$ of r's precondition originally declared in class X:

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

Conversely to the postcondition, establishing r's precondition is a responsibility of callers of r; clients have to establish the precondition determined by the dynamic type—captured by the function $pre.X.r$—, whereas the precondition originally given in X is given as a **free requires** and is only used to prove r's implementation.

axiom ($\forall$ *h*: *HeapType*; c: **ref** $\bullet$
  $h[c,$ **type**$]$ <: $Y \Longrightarrow (\nabla_{pre}(Y.r) \implies pre.X.r(h,\ c)))$;

To establish $pre.X.r$, it is enough to establish any of the clauses $\nabla_{pre}(Y.r)$.

[2] The translation differs for calls to **Precursor** (**super** in Java and **base** in C#).

```
1  function post.EXP.eval(h1, h2: HeapType; c: ref) returns (bool);
2
3  procedure EXP.eval(current: ref);
4    free ensures post.EXP.eval(Heap, old(Heap), current);
5    ensures Heap[current, last_value] ≥ 0;
6    // precondition and frame condition omitted
7
8  axiom (∀ h1, h2: HeapType; o: ref •
9    $type(o) <: EXP ⟹
10     (post.EXP.eval(h1, h2, o) ⟹ (h1[o, last_value] ≥ 0)) );
11 axiom (∀ h1, h2: HeapType; o: ref •
12   $type(o) <: CONST ⟹
13     (post.EXP.eval(h1, h2, o) ⟹ h1[o, last_value] = h1[o, value]) );
14
15 implementation ROOT.main (Current: ref) {
16     var e: ref;
17   entry:
18       // translation of: create {CONST} e.make (5)
19     havoc e;
20     assume Heap[e, $allocated] = false;
21     Heap[e, $ allocated] := true;
22     assume $type(e) =CONST;
23     call CONST.make(e, 5);
24       // translation of e.eval
25     call EXP.eval(e);
26       // translation of: check e.last_value = 5 end
27     assert Heap[e, last_value] = 5;
28     return;
29 }
```

Fig. 6. Boogie translation of the Eiffel classes in Figure 5.

4.2 An Example of Polymorphism with Postconditions

Figure 6 shows the essential parts of the Boogie translation of the example in Figure 5. 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*.

5 Agents (Function Objects)

Eiffel programs can use *agents*—called function objects, closures, or delegates in other languages. Supporting agents in verification poses a number of challenges; Section 5.1 illustrates the main such challenges while succinctly describing Eiffel agents' syntax and semantics with an example. Then, Section 5.2 discusses how to specify agents; Sections 5.3 and 5.4 describe the translation of agents implemented in AutoProof; and Section 5.5 presents how the translation handles framing. This section is based on our previous work [19].

5.1 An Example Using Agents

We illustrate the challenges of verifying agents using an example by Leavens et al. [9] adapted for Eiffel. Consider a class *FORMATTER* that collects routines operating on paragraphs. The class includes, among others, the routine *align_left* (*p*: *PARAGRAPH*) that aligns to the left the text in paragraph *p*, passed as argument. For some details of its implementation that we need not delve into, the routine requires that the paragraph is not already aligned to the left; if executed correctly, it ensures that *p* is changed so that it is left aligned. Assuming class paragraph includes a (pure) routine *left_aligned* that returns true when called on paragraphs that are left aligned, we can write *align_left*'s specification in *FORMATTER* as:

```
class FORMATTER

  align_left (p: PARAGRAPH)
    require
      not p.left_aligned
    do
      -- Operations on p
    ensure
      p.left_aligned
    end

end
```

Class *FORMATTER* includes other routines that operate on paragraphs, such as *align_right*, *justify*, and *add_margin*, each with its proper specification.

Clients of class *FORMATTER* can apply any given routine of the class to objects of type *PARAGRAPH*. To this end, class *PARAGRAPH* offers a routine *format* that takes a generic routine of class *FORMATTER*—wrapped in an **agent**—and applies it to the current object of class *PARAGRAPH*. This is how *format* can be written in Eiffel:

```
class PARAGRAPH

  format (proc: PROCEDURE [FORMATTER, PARAGRAPH];
          f: FORMATTER)
    do
      proc. call (f, Current)
    end

end
```

The first argument has type *PROCEDURE* [*FORMATTER*, *PARAGRAPH*]; this denotes an **agent**, whose target class is *FORMATTER*, with an argument of class *PARAGRAPH*:[3] in other words, if f has type *FORMATTER*, p has type paragraph, and *proc* wraps some routine m in *FORMATTER*, $f.m\ (p)$ is a type-correct call. Agent invocation uses the different syntax shown in the example above: *proc*. *call* (*f*, **Current**) calls the routine wrapped by *proc* on the target *f*, passing it the **Current** object of class *PARAGRAPH* as argument.

This showed how agents are invoked. Let us now demonstrate agent creation in Eiffel with a routine *apply_align_left* that calls *align_left* on a paragraph through *format*:

```
apply_align_left (f: FORMATTER; p: PARAGRAPH)
  require
    not p. left_aligned
  do
    p. format (agent {FORMATTER}.align_left , f )
  ensure
    p. left_aligned
  end
```

The expression **agent** {*FORMATTER*}.*align_left* denotes an agent that wraps routine *align_left* of class *FORMATTER*.[4] This agent definition does not bind the wrapped routine to a specific target or to an argument p; therefore, we call it an agent with *open arguments*. In contrast, the expression **agent** *f. align_left* (*p*) denotes an agent with *closed arguments*: the target is bound to f and the argument to p.

Verifying *apply_align_left* boils down to proving the correctness of the call *p.format*. This, in turn, requires: 1) having a specification of *format*; 2) being

[3] For illustration purposes, we slightly simplify Eiffel's syntax and we limit ourselves to the case of single-argument routines. The generalization is humdrum.

[4] The keyword **agent** is necessary to disambiguate between a function object wrapping *align_left* and the invocation of *align_left* .

able to discharge *format*'s precondition by means of *apply_align_left* 's and to establish *apply_align_left* 's postcondition from *format*'s; 3) specifying *format*'s frame and deduce *apply_align_left* 's frame from it. In conformance with the general verification style of AutoProof (and Boogie), we should handle these problems *modularly*: *format*'s specification and correctness proof should be independent of how *format* is used by clients (and, in particular, which agents it receives as argument). These are the challenges of verifying function objects: Section 5.2 discusses how to write specifications for agents; Sections 5.3 and 5.4 shows how AutoProof uses agents; and Section 5.5 deals with agent framing.

5.2 Specifying Agents

Agents are abstract placeholders for routines; the actual routine attached to an argument of type agent is, in general, known only dynamically. In fact, the purpose of agents is providing a generic container of routines; the specification of agents must conform to the same level of abstraction.

In Eiffel, variables referring to agents all belong to class *ROUTINE* or some of its descendants (such as *PROCEDURE* in the *FORMATTER* example). Specification of agents can then use the functions *precondition* and *postcondition* of the class, which return a Boolean expression respectively corresponding to the **require** and **ensure** clause of the actual routine wrapped by an agent. In the running example, we can use these functions to specify *format* parametrically with respect to its argument *proc* of agent type:

```
format (proc: PROCEDURE [FORMATTER, PARAGRAPH];
          f: FORMATTER)
  require
      proc.precondition (f, Current)
  do
      proc.call (f, Current)
  ensure
      proc.postcondition (f, Current)
  end
```

Using the same mechanism, we can also specify the generic pre- and postcondition of *call* in class *ROUTINE*:

```
call (target: ANY; p: ANY)
  require
      Current.precondition (target, p)
  ensure
      Current.postcondition (target, p)
```

AutoProof's translation uses such generic specifications to reason modularly about the correctness of programs using agents, as described in the following subsections.

5.3 Agents in Boogie: Open Arguments

This section describes the basics of AutoProof's translation of Eiffel agents to Boogie; the presentation focuses on agents with open target and arguments. Section 5.4 outlines how AutoProof deals with closed arguments.

Translating agent specification. The translation introduces two uninterpreted Boolean functions to model the precondition and postcondition of agents:

function $\$precondition(agent,\ target,\ argument\colon \textbf{ref},\ h\colon HeapType)$
returns(**bool**);
function $\$postcondition(agent,\ target,\ argument\colon \textbf{ref},\ h,\ h_0\colon HeapType)$
returns(**bool**);

The function arguments represent references to objects for the agent, its target, and its arguments, plus a copy of the heap h and, for postconditions, the "old" heap h_0 before the agent was invoked.

Translating agent invocation. Consider three reference variables a, t, and p, respectively representing an agent, a target object, and an argument object in the Boogie translation. The agent invocation $a.\,call\ (t,\ p)$ is translated as follows:

assert $\$precondition(a,\ t,\ p,\ Heap)$;
$h_0 := Heap$;
havoc *Heap*;
assume $\$postcondition(a,\ t,\ p,\ Heap,\ h_0)$;

That is, verify that a's precondition holds of the current heap; save the heap to h_0 and nondeterministically change its content; assume that the new heap satisfies a's postcondition.

Translating agent creation. When an agent is created, we must bind the placeholders $\$precondition$ and $\$postcondition$ to the actual pre- and postcondition of the routine wrapped by the agent, so that they can be used in the correctness proofs to reason about agent usages. If a represents in Boogie a reference attached to an agent created as **agent** pr from some routine pr, the following two assumptions bind pr's pre- and postcondition (represented in Boogie by functions $pre.pr$ and $post.pr$):

assume $\forall t,\ p\colon \textbf{ref},\ h_1\colon HeapType \bullet$
$\$precondition(a,\ t,\ p,\ h_1) = pre.pr(t,\ p,\ h_1)$;
assume $\forall t,\ p\colon \textbf{ref},\ h_1, h_2\colon HeapType \bullet$
$\$postcondition(a,\ t,\ p,\ h_1,\ h_2) = post.pr(t,\ p,\ h_1,\ h_2)$;

5.4 Agents in Boogie: Closed Arguments

This section discusses the translation of agents with either closed target or closed argument. The generalization to closed target *and* argument is straightforward.

Translating agent specification. The translation introduces two uninterpreted Boolean functions with different signature to model the precondition and postcondition of agents with closed arguments:

function $\$precondition_1$(*agent*, *arg*: **ref**, *h*: *HeapType*) **returns**(**bool**);
function $\$postcondition_1$(*agent*, *arg*: **ref**, *h*, h_0: *HeapType*) **returns**(**bool**);

The argument *arg* represents either a reference to the open target (if the agent's argument is closed) or a reference to the open argument (if the agent's target is closed). The other function arguments are as in Section 5.1.

Translating agent invocation. Consider two reference variables a and p, respectively representing an agent (with closed target) and an argument object in the Boogie translation. The agent invocation *a. call* (*p*) is translated as:

assert $\$precondition_1(a, p, Heap)$;
$h_0 := Heap$;
havoc *Heap*;
assume $\$postcondition_1(a, p, Heap, h_0)$;

The same translation works if a has closed argument and open target, and p translates a reference to a target object.

Translating agent creation. If a represent in Boogie a reference attached to an agent created as **agent** *u.pr* from some routine *pr* and target *u*, the translation generates the following two assumptions:

assume $\forall p$: **ref**, h_1: *HeapType* •
$\$precondition_1(a, p, h_1) = pre.pr(u, p, h_1)$;
assume $\forall p$: **ref**, h_1, h_2: *HeapType* •
$\$postcondition_1(a, p, h_1, h_2) = post.pr(u, p, h_1, h_2)$;

Similarly, if b represent in Boogie a reference attached to an agent created as **agent** *pr* (*v*) from some routine *pr* and argument *v*, the translation generates the following two assumptions:

assume $\forall t$: **ref**, h_1: *HeapType* •
$\$precondition_1(a, t, h_1) = pre.pr(t, v, h_1)$;
assume $\forall t$: **ref**, h_1, h_2: *HeapType* •
$\$postcondition_1(a, t, h_1, h_2) = post.pr(t, v, h_1, h_2)$;

5.5 Framing of Agents

Eiffel does not offer explicit support to specify *frame conditions*, that is the portion of the heap that a routine may modify. In principle, this is not necessary in many cases, because we can express changed and unchanged entities in postconditions. Following this intuition, AutoProof offers a simple mechanism that can generate frame conditions in the simplest cases; we describe it in Section 6.1.

When proving programs with agents, however, the kind of annotations necessary to express frame conditions in postconditions become cumbersome. This is essentially due to the fact that the frame of an agent depends, in general, also on its dynamically attached target and arguments, which need to be used in its frame specification. Therefore, we introduce **modify** clauses in Eiffel to specify agent framing. In practice, these are implemented as **note** annotations, which does not require changing the parser and guarantees compatibility with any version of the language.

Specifying frame conditions. Following what we did for pre- and postconditions in Section 5.2, we equip class *ROUTINE* with a function *modifies* that returns the list of locations modified by the actual routine wrapped by an agent.[5] **modify** clauses can then include both lists of object references that may be directly modified by a routine and lists of calls to *modifies* functions. The rest of the presentation considers agents with open target and arguments; the modifications necessary to deal with closed arguments are described in a technical report [18]. In the running example, we can use this function to specify *format*'s frame as being whatever *proc* may modify when called on target *f* and argument **Current**:

```
format (proc: PROCEDURE [FORMATTER, PARAGRAPH];
        f: FORMATTER)
  modify
    proc.modifies (f, Current)
```

Translating modify clauses. The translation introduces an uninterpreted function $*modify*:

```
function $modify(agent, target, arg: ref, h: HeapType,
                 obj: ref, fid: FieldId) returns(bool);
```

The function arguments represent references to objects for an agent, its target, and its arguments, a copy of the heap, as well as an additional generic reference *obj* to an object and the identifier *fid* of one of its attributes (fields). Intuitively, if $*modify*(*a*, *t*, *p*, *h*, *o*, *f*) is true, the call *a*.*call* (*t*, *p*) may modify the value of *o*.*f* in the heap *h*.

AutoProof translates a generic Eiffel **modify** clause of some routine *r*:

modify $a_1.modifies\ (t_1, p_1)$, ..., $a_m.modifies\ (t_m, p_m)$, o_1, ..., o_n

into a postcondition clause of *r*'s translation to Boogie:

ensures $\forall o$: **ref**, v: *FieldId* •
$(\neg\ \$modify(a_1, t_1, p_1, Heap, o, v) \wedge \cdots \wedge \neg\$modify(a_m, t_m, p_m, Heap, o, v) \wedge o \neq o_1 \wedge \cdots \wedge o \neq o_n) \implies Heap[o, v] = \mathbf{old}(Heap)[o, v]$

[5] Since *modifies* is only used in **modify** clauses, which are not part of the Eiffel language, we need not provide any actual implementation of *modifies*.

That is, all objects and attributes not explicitly mentioned in the **modify** clause (o_1, ...,o_n) and not modified by the agents (*modifies*) are certainly not changed by routine r. For example, the *format*'s **modify** clause translates to:

ensures $\forall o$: **ref**, v: *FieldId* •
$\neg$ \$*modify*(*proc*, f, *current*, *Heap*, o, v) $\implies$ *Heap*[o, v] = **old**(*Heap*)[o, v]

Frame conditions from agent creation. Finally, when an agent is created, we bind the placeholder \$*modify* to the actual locations modified by the routine wrapped by the agent. Similarly to what we showed in Section 5.3 for pre- and postconditions, if a represents in Boogie a reference attached to an agent created as **agent** pr from some routine pr, the translation generates the assumption:

assume $\forall t$, p, o: **ref**, v: *FieldId*, h_1: *HeapType* •
\$*modify*(a, t, p, h_1, o, v) = *modify.pr*(t, p, h_1, o, v);

where *modify.pr* represents the encoding of pr's frame in Boogie (whose details are straightforward).

Current limitations. The translation of agent framing currently implemented in AutoProof does not support the specification of fine-grained frame disjointness properties. In particular, it is not possible to specify that the locations modified by two agents are disjoint. This is often necessary when reasoning about agents working on composite data structures. The Eiffel library class *LIST*, for example, offers a routine *do_all* that takes an agent passed as argument and applies it to every element of the list. Reasoning about *do_all* requires to distinguish between when the agent is applied to different elements of the list; the simple convention of **modify** clauses, however, does not offer this level of granularity. In our technical report [18], we suggest a mechanism to express such non-interference properties; its implementation in AutoProof is part of future work.

6 Other Features

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

6.1 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 the 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 clause:

ensures $\forall o$: **ref**, f: *FieldId* •
$(o,\ f) \notin mod_r \implies Heap[o,\ f] = \mathbf{old}(Heap[o,\ f])$;

To ensure soundness in the presence of inheritance, the translation always uses the postcondition of the original routine definition to infer the frame of the routine's redefinitions.

The frame conditions inferred by AutoProof work well for routines whose postconditions only mention attributes of primitive type. For routines that manipulate more complex data, such as arrays or lists, the default frame conditions are too coarse-grained; hence programmers have to supplement them with more detailed annotations. Extending the support for automatically generated frame conditions is part of future work.

6.2 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.

7 Case Study

This section presents the results of a case study applying AutoProof to the verification of the 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 [14] when describing Eiffel 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 3.2).[6] 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 4. The *Sequence* example models integer sequences with

[6] 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

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 [7] 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 4 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, we have introduced an option to have AutoProof translating contracts solely based on the static type of references. This speeds up verification in the most common cases, while still having the option to use the more complex encoding when necessary.

Examples 6–8 use agents and are the same examples as in [19]. 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.

8 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 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, delegates are not handled, and exceptions can only be checked at runtime). Spec# includes annotations to specify frame conditions, which make proofs easier but at the price of an additional annotational burden for developers. To ease the annotational overhead, Spec# adds a default frame condition that includes all attributes of the target object. This solution has the advantage that the frame can change with routine redefinitions to include attributes introduced in the subclasses. AutoProof follows a different approach and 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 [11] and Chalice [13], 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 [23] 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 [8]—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.

In our previous work [26], we have presented a verification tool that integrates static proofs and dynamic testing techniques.

9 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 [24] 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 [28].

Acknowledgments. This work has been partially funded by the SNF grant LSAT (200020-134974) and by the Hasler foundation on related projects. A

preliminary version of this work has been presented at the First International Workshop on Intermediate Verification Languages (Boogie'11), held in Wrocław, Poland, in August 2011 and is available as technical report [27].

References

1. A. Banerjee, D. A. Naumann, and S. Rosenberg. Regional logic for local reasoning about global invariants. In *In European Conference on Object Oriented Programming*, ECOOP. Springer-Verlag, 2008.
2. M. Barnett, K. R. M. 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. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison Wesley, 1994.
8. B. Jacobs, J. Smans, and F. Piessens. A quick tour of the VeriFast program verifier. In *Proceedings of APLAS 2010*, 2010.
9. G. T. Leavens, K. R. M. Leino, and P. Müller. Specification and verification challenges for sequential object-oriented programs. *Formal Aspects of Computing*, 19(2):159–189, 2007.
10. K. R. M. Leino. This is Boogie 2. Technical report, Microsoft Research, 2008.
11. K. R. M. Leino. Dafny: an automatic program verifier for functional correctness. In *Proceedings of the 16th international conference on Logic for programming, artificial intelligence, and reasoning*, LPAR-16, pages 348–370. Springer-Verlag, 2010.
12. K. R. M. Leino and P. Müller. Verification of equivalent-results methods. In *ESOP*, volume 4960 of *LNCS*, pages 307–321. Springer-Verlag, 2008.
13. K. R. M. Leino and P. Müller. A basis for verifying multi-threaded programs. In *Proceedings of the 18th European Symposium on Programming Languages and Systems*, ESOP '09, pages 378–393. Springer-Verlag, 2009.
14. B. Meyer. *Object-Oriented Software Construction.* Prentice Hall, second edition, 1997.
15. P. Müller. *Modular Specification and Verification of Object-Oriented Programs*, volume 2262 of *LNCS*. Springer-Verlag, 2002.
16. P. Müller and M. Nordio. Proof-transforming compilation of programs with abrupt termination. In *SAVCBS '07: Proceedings of the 2007 conference on Specification and verification of component-based systems*, pages 39–46, 2007.
17. M. Nordio. *Proofs and Proof Transformations for Object-Oriented Programs.* PhD thesis, ETH Zurich, Switzerland, 2009.
18. M. Nordio, C. Calcagno, B. Meyer, and P. Müller. Reasoning about Function Objects. Technical Report 615, ETH Zurich, 2008.
19. M. Nordio, C. Calcagno, B. Meyer, P. Müller, and J. Tschannen. Reasoning about Function Objects. In *Proceedings of TOOLS-EUROPE*, LNCS, pages 79–96. Springer, 2010.

20. M. Nordio, C. Calcagno, P. Müller, and B. Meyer. A Sound and Complete Program Logic for Eiffel. In M. Oriol, editor, *TOOLS-EUROPE*, volume 33 of *Lecture Notes in Business and Information Processing*, pages 195–214, 2009.
21. M. Nordio, H.-C. Estler, C. A. Furia, and B. Meyer. Collaborative software development on the web, 2011. arXiv:1105.0768v3.
22. M. Nordio, P. Müller, and B. Meyer. Proof-Transforming Compilation of Eiffel Programs. In R. Paige and B. Meyer, editors, *TOOLS-EUROPE*, Lecture Notes in Business and Information Processing, pages 316–335. Springer-Verlag, 2008.
23. P. W. O'Hearn, H. Yang, and J. C. Reynolds. Separation and information hiding. In *POPL '04*, pages 268–280, 2004.
24. 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.
25. J. Tschannen. Automatic verification of Eiffel programs. Master's thesis, Chair of Software Engineering, ETH Zurich, 2009.
26. J. Tschannen, C. A. Furia, M. Nordio, and B. Meyer. Usable verification of object-oriented programs by combining static and dynamic techniques. In *Proceedings of the 9th International Conference on Software Engineering and Formal Methods (SEFM '11)*, LNCS, pages 382–398. Springer, 2011.
27. J. Tschannen, C. A. Furia, M. Nordio, and B. Meyer. Verifying Eiffel programs with Boogie. In *First International Workshop on Intermediate Verification Languages (BOOGIE)*, 2011. Available at `http://arxiv.org/abs/1106.4700`.
28. 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.