Reasoning about Function Objects

Martin Nordio[1], Cristiano Calcagno**[23], Bertrand Meyer[1], Peter Müller[1], and Julian Tschannen[1]

[1] ETH Zurich, Switzerland
{Martin.Nordio, Bertrand.Meyer, Peter.Mueller, julian.tschannen}@inf.ethz.ch
[2] Monoidics Ltd
[3] Imperial College, London, UK
ccris@doc.ic.ac.uk

Abstract. Modern object-oriented languages support higher-order implementations through function objects such as delegates in C#, agents in Eiffel, or closures in Scala. Function objects bring a new level of abstraction to the object-oriented programming model, and require a comparable extension to specification and verification techniques. We introduce a verification methodology that extends function objects with auxiliary side-effect free (pure) methods to model logical artifacts: preconditions, postconditions and modifies clauses. These pure methods can be used to specify client code abstractly, that is, independently from specific instantiations of the function objects. To demonstrate the feasibility of our approach, we have implemented an automatic prover, which verifies several non-trivial examples.

1 Introduction

Object-oriented design makes a clear choice in dealing with the basic duality between data and operations: it bases system architecture on the object, more precisely the object types as represented by classes, and attaching operations to one such class. Functional programming languages, on the other hand, use functions as the primary compositional elements. The two paradigms are increasingly borrowing from each other: functional programming languages such as OCaml integrate object-oriented ideas, and a number of object-oriented languages now offer a mechanism to package operations (routines, methods) as objects. In the dynamically typed world, the idea goes back at least to Smalltalk with its blocks; among statically typed languages, C# has introduced *delegates*, Eiffel *agents*, and Scala *closures*.

The concept of agent or delegate is, in its basic form, very simple, with immediate applications. A typical one, in a Graphical User Interface system, is for some part of a system to express its wish to observe (in the sense of the Observer pattern [12]) events of a certain type, by registering a procedure to be executed in response:

** This work was done while visiting ETH Zurich.

US_map.left_click.subscribe (**agent** *show_state_votes*)

This indicates that whenever a *left_click* event occurs on the map, the given procedure *show_state_votes* should be executed. The routine *subscribe* takes as argument a function object representing a procedure with two integer arguments. Since the function object is a formal argument, *subscribe* does not know which exact procedure, such as *show_state_votes*, it might represent; but it can call it all the same, through a general procedure *call* applicable to any function object, and any target and argument objects.

Function objects appear in such examples as a form of function pointers as available for example in C and C++. But they go beyond this first analogy. Firstly, they are declared with a signature and hence provide a statically typed mechanism, whereas a function pointer just denotes whatever is to be found in the corresponding memory address. Secondly, a function object represents a routine abstraction, and can be subject to dynamic dispatch when the receiver is an open argument[4].

Function objects have proved attractive to object-oriented programmers, but they also raise new verification challenges. To address these problems, we introduce a specification and verification technique. Our approach uses side-effect free (pure) routines to specify abstractly the pre- and postconditions of function objects. These pure routines can be used to specify client code independently from specific function objects. Using previous work on encoding pure routines in Boogie [8,22], these routines are encoded as mathematical functions, which represent the function object's pre- and postcondition. The basic idea is that to prove a property of a function object call, it suffices to prove that the abstract precondition of the function object holds before the invocation; then we can assume the abstract postcondition of the function object holds after its invocation.

The main contributions of this paper are: (1) a verification methodology for function objects, and (2) an automatic verifier for function objects. The verifier takes an Eiffel program, translates it to Boogie2 [16], and then proves the Boogie2 code using the Boogie verifier [2]. We demonstrate the practicality of our approach with a suite of examples, including one previously described as an open problem, and more function objects intensive programs which implement graphical user interfaces. Although we focus on Eiffel agents, we believe that the same ideas apply to similar mechanisms in other languages, such as C# delegates.

Outline. Section 2 presents example applications of agents and their verification challenges. Section 3 describes the verification methodology. This methodology is extended to framing in Section 4. Section 5 applies the methodology to the examples from Section 2. In Section 6, we show a set of examples that have been verified using the implemented automatic prover. Section 7 discusses related work; Section 8 summarizes the result and describes future work.

[4] An argument is *open* if it must be provided in the invocation of the agent.

2 Agent Examples and their Verification Challenges

In this section, we present some typical applications of agents.

2.1 Formatter

The first example comes from a paper by Leavens et al. [20] and is recouched in Eiffel below. It is of particular interest since they describe it as a verification challenge beyond current techniques. The class *FORMATTER* models paragraph formatting with two alignment routines. The class *PARAGRAPH* includes a procedure to format the current paragraph:

```
class FORMATTER
  align_left (p:PARAGRAPH)
    require
        not p. left_aligned
    do
        ... Operations on p ...
    ensure
        p. left_aligned
end

  align_right (p:PARAGRAPH)
    require
        not p. right_aligned
    do
        ... Operations on p ...
    ensure
        p. right_aligned
```

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

For illustration purposes, the routines *align_left* and *align_right* require that the paragraph is not left aligned and not right aligned, respectively. The routines *left_aligned* and *right_aligned* are pure routines (side-effect free) defined in the class *PARAGRAPH*, and return *true* if the paragraph is left aligned or right aligned, respectively.

In Eiffel, the contracts of a class are its invariant, and the precondition and postcondition that can be attached to any routine, with the respective keywords **invariant**, **require** and **ensure**. Each such clause involves an assertion written out as a sequence of boolean expressions. An absent contract clause is equivalent to one specificying *True*. In the routine *format*, the signature *proc: PROCEDURE [FORMATTER, PARAGRAPH]*[5] declares an agent *proc* with two open arguments (the target of type *FORMATTER* and a parameter of type *PARAGRAPH*). The agent *proc* is invoked using the procedure *call* (**Current** denotes the receiver object, *this* in C#).

[5] This is a simplification of the Eiffel syntax; the Eiffel declaration is *PROCEDURE[FORMATTER, TUPLE[PARAGRAPH]]*

An example of the use of the *format* routine is shown in the routine *apply_align_left*:

```
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 notation **agent** {*FORMATTER*}.*align_left* denotes a function object that represents the *align_left* routine of the class *FORMATTER*. The keyword **agent** is used to distinguish between the function object *align_left* and the invocation of the routine *align_left*.

The verification challenge in this case is to specify and verify the routine *format* in an abstract way, abstracting the pre and postcondition of the agent. Then, one should be able to invoke the routine *format* with a concrete agent, here *align_left*, and to show that the postcondition of *align_left* holds. If the *format* routine is called with another routine, say *align_right*, one should be able to show that the postcondition of *align_right* holds without modifying the proof of *format*. Another issue is framing; one should be able to express what the routine *format* modifies, but abstracting from the specific routines *align_left* and *align_right*. When the routine *format* is invoked using the agent *align_left*, we should be able to show that *format* only modifies the locations that *align_left* modifies.

2.2 Archive Example

This section describes the *archive* example presented by Leavens et al. [20] and proved by Müller and Ruskiewicz [23]. This example illustrates the application of agents with closed arguments[6].

Figure 1 presents the example encoded in Eiffel. The class *TAPE_ARCHIVE* defines a tape with a routine *store* which stores objects if the device is loaded. An application of agents is implemented in the routine *log* of class *CLIENT*, which calls the agent *log_file* with the string *s*. Finally, the class *MAIN* shows an example of the invocation of the routine *log*.

The invocation *log_file.call*(*s*) invokes the procedure *log_file* with the parameter *s*. The declaration *PROCEDURE[TAPE;ANY]*[7] indicates that *log_file* is an agent with closed argument of type *TAPE* and one open argument of type *ANY*. The target of the invocation is defined in the creation of the agent. In this example, the target object is *t* defined by ***agent*** *t.store*.

[6] Closed arguments are the arguments of an agent provided in the agent declaration.

[7] This is a simplification of the declaration in Eiffel. The declaration in Eiffel is *PROCEDURE[TAPE,TUPLE[ANY]]*.

```
class TAPE_ARCHIVE
  tape: TAPE
  is_loaded: BOOLEAN
    ensure
      Result = (tape /= void)

  make
    do
       create tape
    end

  store (o: ANY)
    require
      is_loaded
    do
      tape.save (o)
    end
  -- other routines
  --  omitted
end
```

```
class TAPE
  save(o: ANY) do ... end
    -- other routines omitted
end

class CLIENT
  log ( log_file :PROCEDURE[TAPE;ANY];
              s:STRING)
    do
      log_file . call(s)
    end
end
class MAIN
  main (c: CLIENT)
    local
      t: TAPE_ARCHIVE
    do
      create t.make
      c.log (agent t.store, "Hello World")
    end
end
```

Fig. 1. Archive example encoded in Eiffel.

The verification challenge in this case is to verify the routine *log* in an abstract way, and being able to show that the precondition of the agent *store* holds before its invocation. In the routine *log*, the methodology has to assume that the target is closed but the exact target is unknown.

3 Verification Methodology

A verification technique should address both the specification of routines that uses function objects and the verification of invocation of function objects. Section 3.1 considers the first issue; the remainder of this section examines the second one.

3.1 Specifying Function Objects

The difficulty of specifying the correctness of agents is that while a variable of an agent type represents a routine, it is impossible to know statically which routine that is. The purpose of agents is to abstract from individual routines. The specification must reflect this abstraction.

What characterizes the correctness of a routine is its precondition and its postcondition. For an agent, these are known abstractly through the functions

precondition and *postcondition* of class *ROUTINE* and its descendants. These functions enable us to perform the necessary abstraction on agent variables and expressions. The approach makes it possible for example to equip the routine *format* with a contract:

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

Note that the precondition of *format* uses the routine *precondition* to query the precondition of the procedure *proc*. Finally, we need to specify the routine *call* in the class *ROUTINE*. Its specification is the following:

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

3.2 Reasoning

This section describes the methodology to reason about agents with open arguments. This methodology is presented as a translation from Eiffel to Boogie2 [16]. The translation uses the basic Boogie2 instructions **assume**, **assert**, **havoc**, and assignment. In the following, we present the translation of agent initialization and agent invocation. The translation of other instructions such as assignments and routine invocation is similar to the translation applied in Spec#; for more details see [15]. The methodology is extended for closed arguments in Section 3.3; framing is handled in Section 4.

Agent Pre- and Postconditions. The methodology introduces two uninterpreted functions to model the pre- and postcondition of the agent. The function[8] *$precondition* takes three values (the agent, the target, and the parameter), and the current heap, and yields the evaluation of the agent's precondition. The function *$postcondition* takes a second heap to evaluate old expressions. The signatures of these functions are defined as follows[9]:

$$\$precondition : Value \times Value \times Value \times Heap \rightharpoonup Bool$$
$$\$postcondition : Value \times Value \times Value \times Heap \times Heap \rightharpoonup Bool$$

[8] We use the prefix $ for the mathematical functions to distinguish them from the Eiffel routines.

[9] $\rightharpoonup$ denotes partial functions.

Invoking Agents. Given an agent a, a target t, and an argument p, the agent is invoked using the Eiffel routine *call*. The translation of the agent invocation $a.call(t, p)$ first asserts the precondition of the agent, and then assumes its postcondition. The proof obligations are the following:

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

The current heap is denoted by *Heap*. The assignment $h_0 := Heap$ saves the current heap, then h_0 is used to evaluate old expressions in the postcondition of the agent. The **havoc** command assigns an arbitrary value to the heap.

Initializing Agents. The translation above asserts the abstract precondition of the agent. This abstract precondition could be any precondition of any procedure. Once the agent is initialized with a procedure pr, the methodology connects the abstract pre- and postcondition of the agent with the concrete pre- and postcondition of the procedure pr. Thus, if the precondition of pr holds, the prover will be able to show that the abstract precondition holds.

Given the agent initialization a := **agent** pr where pr is a procedure[10], the methodology generates the following assumptions:

assume $\forall t, p : ObjectId; h_1 : Heap : \$precondition(a, t, p, h_1) = \$pre_{pr}(t, p, h_1)$
assume $\forall t, p{:}ObjectId; h_1, h_2 : Heap : \$postcondition(a, t, p, h_1, h_2) = \$post_{pr}(t, p, h_1, h_2)$

where $\$pre_{pr}$ and $\$post_{pr}$ denotes the pre- and postcondition of the procedure pr, t the target object, and p the argument respectively; we assume that the agent variable a is a fresh variable.

The translation of agents to Boogie2 is based on the translation of pure routines [8,22]. The novel concepts are the introduction of the functions $\$precondition$ and $\$postcondition$ to model the agent pre- and postcondition, and the generation of assumptions for the initialization of the agent, which relates the pre- and postcondition of the agent with the concrete pre- and postcondition of the procedure.

3.3 Reasoning about Closed Arguments

To model closed arguments, we define two uninterpreted functions: $\$precondition_1$ and $\$postcondition_1$[11]. These functions are similar to the functions defined in the

[10] **agent** pr is an abbreviation for keeping all arguments open (including the target), as in **agent** $\{TYPE\}.pr(?)$.

[11] As a reminder, we assume that routines have only one parameter, although, the methodology can be extended easily.

section above but they take an agent with one closed argument (either closed target or closed parameter) and the heap(s), and yield the evaluation of the pre- and postcondition. The signatures are:

$$\$precondition_1 : Value \times Value \times Heap \rightharpoonup Bool$$
$$\$postcondition_1 : Value \times Value \times Heap \times Heap \rightharpoonup Bool$$

The translation for initializing agents, and invoking agents are similar to the section above; Figure 2 presents this translation.

Eiffel code	*Boggie2 code*
(A) $a.call(p)$	**assert** $\$precondition_1(a, p, Heap)$ $h_0 := Heap$ **havoc** $Heap$ **assume** $\$postcondition_1(a, p, Heap, h_0)$
(B) a := **agent** $t_1.pr$	**assume** $\forall p : ObjectId; h_1 : Heap :$ $\quad \$precondition_1(a, p, h_1) = \$pre_{pr}(t_1, p, h_1)$ **assume** $\forall p : ObjectId; h_1, h_2 : Heap :$ $\quad \$postcondition_1(a, p, h_1, h_2) = \$post_{pr}(t_1, p, h_1, h_2)$
where t_1 is the closed target, and pr a procedure	
(C) a := **agent** $pr(p_1)$	**assume** $\forall t : ObjectId; h_1 : Heap :$ $\quad \$precondition_1(a, t, h_1) = \$pre_{pr}(t, p_1, h_1)$ **assume** $\forall t : ObjectId; h_1, h_2 : Heap :$ $\quad \$postcondition_1(a, t, h_1, h_2) = \$post_{pr}(t, p_1, h_1, h_2)$
where p_1 is the closed parameter, and pr a procedure	

Fig. 2. Translation of Agents with Closed Arguments to Boogie2: (A) Agent Invocation with Closed Arguments; (B) Closed Target Initialization; (C) Closed Parameter Initialization.

Note that Eiffel does not distinguish between an agent with open target and an agent with open parameter. Both agents are declared with the same notation. Thus, the methodology uses the functions $\$precondition_1$ and $\$postcondition_1$ to express the precondition and postcondition with closed target and closed parameter, and then it uses the assumptions generated in the initialization of the agent.

4 Framing

A necessary part of a routine specification is the modifies clause, which defines the locations that are modified by the routine. The problem of defining these locations is known as *frame problem*. The frame problem has been addressed for example using dynamic frames [18], ownership [6], separation logic [31,26], and regional logic [1]. However, this problem has to be solved for agents. This section presents a solution for framing agents based on dynamic frames. As future work, we plan to investigate the integration with other techniques such as separation logic.

In Section 2.1 we have specified the routine *format*, however, one needs to specify what locations this routine modifies. A candidate solution for this problem is to assume that *format* modifies all the locations than can be accessed from the target and the arguments of the agent *proc*. However, this assumption is too strong since *format* may only modify a few attributes of *proc*'s target. Note that *format* can be invoked with different routines, and each routine might modify different locations.

To address the frame problem for agents, we adapt dynamic frames. Instead of using a set of locations as in Kassios's work [18], we introduce a routine *modifies* (in the source language), which takes an agent a, its target t and argument value p, and returns the locations modified by the agent a with target t and argument p. This function abstracts from the specific locations that the agent modifies. Thus, the modifies clause of *format* can be defined as follows (pre and postconditions are omitted):

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

This modifies clause expresses that the routine *format* modifies the locations that are modified by the procedure *proc*. Depending on the routine used to invoke *format*, the function *modifies* will yield a different set of locations.

Following, we describe the encoding of framing for agents with open arguments; framing for closed arguments is presented in our technical report [24].

Modifies Clauses. We have extended Eiffel with *modifies clauses*. Each routine contains a modifies clause which is defined as a comma separated list of locations. To express what locations are modified by an agent, we introduce the function *modifies*. The definition of *modifies clauses* and routine declarations is the following:

$$
\begin{array}{rcl}
\mathit{Mclause} & ::= & \mathit{Mclause}, \mathit{Mclause} \\
& | & \mathit{VarId} \\
& | & \mathit{modifies}(\mathit{VarId}, \mathit{VarId}, \mathit{VarId}) \\
\mathit{Routine} & ::= & \mathit{RoutineId}\ (\mathit{VarId} : \mathit{Type}) :\ \mathit{Type} \\
& & \quad \texttt{require} \\
& & \quad\quad \mathit{BoolExp} \\
& & \quad \texttt{modify} \\
& & \quad\quad \mathit{Mclause} \\
& & \quad \texttt{do} \\
& & \quad\quad \mathit{Instr} \\
& & \quad \texttt{ensure} \\
& & \quad\quad \mathit{BoolExp} \\
& & \quad \texttt{end}
\end{array}
$$

where *boolExp* are boolean expressions, *RoutineId* routine identifiers, *VarId* variable identifiers, and *Instr* instructions.

Encoding of Modifies Clauses. To encode the routine *modifies*, we introduce an uninterpreted function $\$\mathit{modifies}$ which takes an agent a, its target and argument values, the current heap, an object value o, and a field name f, and yields *true* if the agent a with its target and argument modifies the field f of the object o. The signature of this function is the following:

$$\$\mathit{modifies} : \mathit{Value} \times \mathit{Value} \times \mathit{Value} \times \mathit{Heap} \times \mathit{Value} \times \mathit{FieldId} \rightharpoonup \mathit{Bool}$$

Modifies clauses are encoded in a similar way to Spec#, but cosidering the mapping of the Eiffel function *modifies*. Modifies clauses are a list of applications of the function *modifies* and variable identifiers. Given the modifies clause in the source language:

$$\mathit{modifies}(a_1, t_1, p_1), ..., \mathit{modifies}(a_n, t_n, p_n), v_1, ..., v_m$$

this clause is encoded in Boogie2 as:

$$\textbf{ensures}\ \forall o : \mathit{ObjectId};\ \mathit{fId} : \mathit{FieldId} : \begin{pmatrix} \mathit{not}\ \$\mathit{modifies}(a_1, t_1, p_1, \mathit{Heap}, o, \mathit{fId}) \wedge ... \wedge \\ \mathit{not}\ \$\mathit{modifies}(a_n, t_n, p_n, \mathit{Heap}, o, \mathit{fId}) \\ \wedge\ o \neq v_1\ \wedge ...\ \wedge\ o \neq v_m \end{pmatrix} \Rightarrow\ \mathit{Heap}[o, \mathit{fId}] = \mathit{old}(\mathit{Heap})[o, \mathit{fId}]$$

For example, the modifies clause of the routine *format* is encoded as follows:

$$\textbf{ensures}\ \forall o : \mathit{ObjectId};\ \mathit{fId} : \mathit{FieldId} : \mathit{not}\ \$\mathit{modifies}(\mathit{proc}, f, \mathit{Current}, \mathit{Heap}, o, \mathit{fId}) \Rightarrow \mathit{Heap}[o, \mathit{fId}] = \mathit{old}(\mathit{Heap})[o, \mathit{fId}]$$

This property expresses that for all objects o, and all fields *fId* that are not modified by the agent *proc* with the target f and argument *Current*, the value

of the field $o.fId$ in the current heap is equal to the value of $o.fId$ in the old heap. The expression $Heap[o, fId]$ yields the value of the field fId of the object o in the current heap.

Initializing Agents. To address the frame problem for agents, we need to link the uninterpreted function $\$modifies(proc, t, p)$ with the locations that the routine *proc* modifies. We solve this by applying the same approach used to reason about agent pre- and postconditions. Thus, our methodology connects the uninterpreted function $\$modifies$ with the concrete set of locations that the agent modifies.

Given a procedure pr, the agent initialization a := **agent** pr generates the following assumptions:

$$\begin{array}{l}
\textbf{assume } \forall t, p : ObjectId; h_1 : Heap : \$precondition(a, t, p, h_1) = \$pre_{pr}(t, p, h_1) \\
\textbf{assume } \forall t, p : ObjectId; h_1, h_2 : Heap : \\
\qquad\qquad \$postcondition(a, t, p, h_1, h_2) = \$post_{pr}(t, p, h_1, h_2) \\
\textbf{assume } \forall t, p, o : ObjectId;\ fId : FieldId; h_1 : Heap : \\
\qquad\qquad \$modifies(a, t, p, h_1, o, fId) = \$modifies_{pr}(t, p, h_1, o, fId)
\end{array}$$

The assumptions for the functions $\$precondition$ and $\$postcondition$ are the same assumptions as described in Section 3.2. The third assumption relates the uninterpreted function $\$modifies$ with the modifies clause of pr. The function $\$modifies_{pr}$ yields *true* if the procedure pr modifies the field $o.fId$ for the target t and argument p. The definition of this function is generated from the modifies clause of the procedure pr.

For example, assuming that the routine *align_left* in the class *FORMATTER* (Section 2.1) modifies its argument p, then $modifies_{align\_left}$ is defined as

$$\$modifies_{align\_left}(Current, p, h, o, fId) \triangleq (h[o] = p)$$

Limitations. The current implementation of modifies clauses is not powerful enough to express some non-interference properties. One can express that an agent a modifies a set of locations s, and an agent b modifies another set of locations r, however, we cannot express that these locations are disjoint.

The same problem arises when verifying agents with open targets. An example of the use of open target is the routine *do_all* defined in the class *LIST* of the Eiffel base library. The *do_all* routine takes an agent with open target, and invokes the agent for all elements of the list. To verify the routine *do_all*, one needs to reason about non-interference at an abstract level, because the invocation of the agent for the *ith* element of the list might violate the precondition of the agent for the *jth* element of the list. To address this problem, a mechanism to support non-interference reasoning is required, as discussed in our technical report [24]. Extending our implementation to support this mechanism is part of future work.

5 Applications

In this section we study the applicability of our methodology to a range of examples which illustrate challenging aspects of reasoning about function objects.

5.1 Formatter Example

To verify the routine *format*, the methodology generates the following Boogie2 code[12]:

format(*proc* : *PROCEDURE*[*FORMATTER*, *PARAGRAPH*]; *f* : *FORMATTER*)
1 **assume** $*precondition*(*proc*, *f*, *Current*, *Heap*)
2 **assert** $*precondition*(*proc*, *f*, *Current*, *Heap*)
3 $h_0 := Heap$
4 **havoc** *Heap*
5 **assume** $*postcondition*(*proc*, *f*, *Current*, *Heap*, h_0)
6 **assume** $\forall o : ObjectId;\ fId : FieldId :$
not $*modifies*(*proc*, *f*, *Current*, *Heap*, *o*, *fId*) $\Rightarrow$ $Heap[o, fId] = h_0[o, fId]$
7 **assert** $*postcondition*(*proc*, *f*, *Current*, *Heap*, h_0)
8 **assert** $\forall o : ObjectId;\ fId : FieldId :$
not $*modifies*(*proc*, *f*, *Current*, *Heap*, *o*, *fId*) $\Rightarrow$ $Heap[o, fId] = h_0[o, fId]$

The pre- and postcondition of *format* are translated in the lines 1 and 7, respectively. The modifies clause of *format* is translated in line 8. The agent invocation is translated in the lines 2-6. This translation assumes the postcondition and the modifies clause of *call* in lines 5 and 6. The proof is straightforward since the **assume** and **assert** instructions in lines 1 and 2, lines 5 and 7, and lines 6 and 8 refer to the same heap.

The most interesting case in the verification of function object is the verification of clients that use function objects, such as *apply_align_left*. The application of the methodology to this routine generates the Boogie2 code presented in Figure 3.

Similar to the previous example, lines 1 and 11 are generated by the translation of the pre- and postcondition; line 12 is the translation of the modifies clause. The declaration **agent** {*FORMATTER*}.*align_left* generates lines 2-5. The precondition and postcondition of the routine *align_left* is denoted by $\$pre_{align\_left}$ and $\$post_{align\_left}$ respectively; the modifies clause of *align_left* is denoted by $\$modifies_{align\_left}$. The invocation of the routine *format* produces lines 6-10. The current heap is stored in h_0 in line 7 to be able to evaluate the postcondition in line 9.

The key points in the proof are the **assert** instructions at lines 6, 11 and 12. By the definition of $\$pre_{align\_left}$, $\$post_{align\_left}$, and $\$modifies_{align\_left}$ we know:

[12] To simplify the presentation, we use the signature of the function in Eiffel.

$apply\_align\_left(f : FORMATTER; p : PARAGRAPH)$

1 **assume not** $p.\$left\_aligned$
2 $a := \mathbf{agent}\{FORMATTER\}.align\_left$
3 **assume** $\forall t_1, p_1 : ObjectId; h : Heap :$
$\$precondition(a, t_1, p_1, h) = \$pre_{align\_left}(t_1, p_1, h)$
4 **assume** $\forall t_1, p_1 : ObjectId; h, h' : Heap :$
$\$postcondition(a, t_1, p_1, h, h') = \$post_{align\_left}(t_1, p_1, h, h')$
5 **assume** $\forall t_1, p_1, o : ObjectId;\ fId : FieldId; h : Heap :$
$\$modifies(a, t_1, p_1, h, o, fId) = \$modifies_{align\_left}(t_1, p_1, h_1, o, fId)$
6 **assert** $\$precondition(a, f, p, Heap)$
7 $h_0 := Heap$
8 **havoc** $Heap$
9 **assume** $\$postcondition(a, f, p, Heap, h_0)$
10 **assume** $\forall o : ObjectId;\ fId : FieldId :$
$not\ \$modifies(proc, f, p, Heap, o, fId) \Rightarrow Heap[o, fId] = h_0[o, fId]$
11 **assert** $p.\$left\_aligned$
12 **assert** $\forall o : ObjectId;\ fId : FieldId :$
$o \neq p \Rightarrow Heap[o, fId] = h_0[o, fId]$

Fig. 3. Proof obligations of the routine *apply_align_left*.

$$\forall t_1, p_1 : ObjectId; h : Heap : \$pre_{align\_left}(t_1, p_1, h) = \ not\ p_1.\$left\_aligned \quad (1)$$
$$\forall t_1, p_1 : ObjectId; h, h' : Heap : \$post_{align\_left}(t_1, p_1, h, h') = \ p_1.\$left\_aligned \quad (2)$$
$$\$modifies_{align\_left}(Current, p, h, o, fId) \ = h[o] \neq p \quad (3)$$

In particular, $\$pre_{align\_left}(f, p, Heap) = \ not\ p.\$left\_aligned$. Then, the assertion at line 6 is proven using the assumptions at lines 1 and 3, and (1). The assertion at line 11 is proven in a similar way using the assumptions at lines 4 and 9, and (2). Finally, the assertion at line 12 is proven in a similar way using the assumptions at lines 5 and 10, and (3).

5.2 Archive Example

In the archive example, the most interesting proof is the proof of the routine *main*. The routine *log* is interesting to show how to specify and prove closed arguments. To prove these routines, we apply the methodology described in Section 3.3 (to simplify the example, we omit the the translation for framing). The proof for the routine *log* is similar to the proof of the *format* routine. The only change is the use of the function $\$precondtion_1$ which takes only three arguments (the procedure *log_file*, the string s and the heap). The generated proof obligations are the following:

$log(log\_file : PROCEDURE[ANY; TAPE];\ s : STRING)$
1 **assume** $\$precondition_1(log\_file, s, Heap)$
2 **assert** $\$precondition_1(log\_file, s, Heap)$
3 $log\_file.call(s)$

The translation of the routine *main* is as follows:

$main(c : CLIENT)$
1 **create** $t.make$
2 **assert** $t.\$is\_loaded$
3 $a :=$ **agent** $t.store$
4 **assume** $\forall p_1 : ObjectId; h : Heap :$
$\$precondition_1(a, p_1, h) = \$pre_{store}(a, t, p_1, h)$
5 **assume** $\forall p_1 : ObjectId; h, h' : Heap :$
$\$postcondition_1(a, p_1, h, h') = \ \$post_{store}(a, t, p_1, h, h')$
6 **assert** $\$precondition_1(a, "HelloWorld", Heap)$
7 $c.log(a, "HelloWorld")$

The proof of routine *main* translates the agent in lines 3-5. The function $precondition_1$ is used to express the precondition of the agent with closed target. Using the assumption at line 4 and the knowledge of line 2, one can prove the **assert** instruction at line 7.

6 Experiments

We have implemented an automatic verifier for agents, called *EVE Proofs*, following the architecture of the Spec# verifier [2]. Given an Eiffel program, the tool generates a Boogie2 [16] file, and uses the Boogie verifier to prove the generated program. The tool is integrated in *EVE* [10] (the Eiffel Verification Environment), and it can be downloaded from `http://eve.origo.ethz.ch/`. Once the user has specified pre- and post-conditions, and invariants, the verification is completely automatic.

EVE Proofs translates each agent initialization into Boogie2 assumptions as described in Section 3 and Section 4. These assumptions are generated inside the body of the Boogie2 procedure corresponding to the Eiffel routine. Thus, Boogie only considers the agent properties inside the procedure where the agent is used.

Using *EVE Proofs*, we have automatically proven a suite of examples: the examples presented in Section 2, and several more agent-intensive programs to model graphical user interfaces. The examples can be downloaded from `http://se.ethz.ch/people/tschannen/examples.zip`. The experiments were run on a machine with a 2.71 GHz dual core Pentium processors with 2GB of RAM.

Table 1 presents the results of the experiments. For each example, the table shows the number of classes, agents, agent calls, and lines of code in Eiffel, as

Name	*Classes*	*Agents*	*Agent calls*	*LOC Eiffel*	*LOC Boogie2*	*Time [s]*
1. Formatter	3	2	2	116	414	1.57
2. Archiver	4	1	1	119	440	1.58
3. Command	3	2	4	120	435	1.61
4. Calculator	3	11	11	243	817	25.14
5. ATM	4	13	20	486	1968	73.72
6. Cell / Recell	3	4	4	151	497	1.71
7. Counter	2	2	4	96	356	1.53
8. Sequence	5	2	4	200	526	1.78
Total	27	37	50	1513	5453	108.64

Table 1. Examples automatically verified by *EVE Proofs*

well as the number of lines of the encoding in Boogie2. The last column shows the running time of Boogie (the dominant factor in the verification).

The formatter and archiver examples have been discussed in the previous sections. The third example is a typical implementation of the command pattern [12]. It defines a command class that uses an agent to store an action, which will be executed when the command's execute function is called. This pattern is also used in the *calculator* and *ATM* examples, which model applications using graphical user interfaces (GUI). The *calculator* example implements the GUI of a simple calculator with buttons for the digits, and basic arithmetic operations such as addition, subtraction, and multiplication. The *ATM* example implements a GUI for an ATM machine, and it also implements client code where a pin number is entered, and money is deposited and withdrawn from an account. These two examples are of particular interest because the GUI libraries in Eiffel typically use agents to react on events.

The ATM and calculator example make extensive use of the command pattern and therefore have more agents. Due to the increased number of agent calls and more difficult contracts, the proof of these examples takes significantly longer than the smaller examples. The ATM example is slower than the Calculator because the ATM example has more agent calls.

The cell/recell example is an extension of an example by Parkinson and Bierman [29] with agents. The *counter* example implements a simple counter class with increase and decrease operations. The last example defines a class hierarchy for integer sequences introducing an arithmetic and Fibonacci sequence.

Graphical user interfaces use agents intensively. We have performed some experiments to check the verification time in such applications. The results have shown that increasing the number of agent initializations in a routine slighlty increases the overall verification time of that routine. This time increase is due to the additional assumptions that are generated for each agent. The new assumptions are then used by Boogie, and thus slow down the verification. Since the verification methodology is modular, the assumptions are local to a specific routine. The increase of agent initializations in one routine does not affect the verification time of other routines.

7 Related Work

Jacobs [14] as well as Müller and Ruskiewicz [23] extend the Boogie verification methodology to handle C# delegates. They associate pre- and postconditions with each delegate type. When the delegate type is instantiated, they prove that the specification of the method refines the specification of the delegate type. At the call site, one has to prove the precondition and may assume the postcondition of the delegate. By contrast, the methodology presented here "hides" the specification behind abstract predicates. Callers will in general require the predicates to hold that they need in order to call an agent. The approach taken by Jacobs, Müller, and Ruskiewicz splits proof obligations into two parts, the refinement proof when the delegate is instantiated and the proof of the precondition when the delegate is called. This split makes it difficult to handle closed parameters, in particular, the closed receiver of C# delegates. Both previous works use some form of ownership [21] to ensure that the receiver of a delegate instance has the properties required by the method underlying the delegate. Our methodology requires only one proof obligation when the agent is called and can be generalized to several closed parameters more easily.

Parkinson and Bierman [28,29] introduce abstract predicates to verify object-oriented programs in separation logic. Abstract predicates are a powerful means to abstract from implementation details and to support information hiding and inheritance. Distefano and Parkinson [9] show the applicability of abstract predicates by implementing a tool to verify Java programs. The tool handles several design patterns such as the visitor pattern, the factory pattern, and the observer pattern. The predicates we use for the preconditions and postconditions of agents are inspired by abstract predicates. Even though Parkinson and Bierman's work and Distefano and Parkinson's work do not handle function objects, we believe that the ideas presented in this paper also apply to their setting.

Birkedal et al. [4] present higher-order separation logic, a logic for a second-order programming language, and use it to verify an implementation of the Observer pattern [19]. In contrast to separation logic, the methodology presented in this paper is designed to work with standard first-order theorem provers.

Contracts have been integrated into higher-order functions. Findler et al. [11] integrate contracts using a typed lambda calculus with assertions for higher-order functions. Honda et al. [13,3] introduce a sound compositional program logic for higher-order functions. Régis-Gianas and Pottier [30] develop a Hoare logic for a call-by-value programming language equipped with higher-order functions. Kanig and Filiatre [17] present a tool to verifier higher-order functions. The tool uses an intermediate language; the tool is intended to be used for verification tools targeting ML-like programming languages. Function objects in object-oriented languages are more complex than higher-order functions in functional languages because of the heap and side-effects. Although the pre- and postconditions of agents are side-effect free, agent calls are not: agent calls can access the heap, and can modify any attribute. This makes the verification of function objects harder compared to functional programming languages.

Börger et al. [5] present an operational semantics of C# including delegates. The semantics is given using abstract state machines. However, this work does not describe how to apply this model to specify and verify C# programs.

Schoeller [32] has developed an automatic verifier for a subset of Eiffel. The tool generates Boogie code, and uses the Boogie verifier to prove the generated Boogie program. However, Schoeller's tool does not handle agents. Paige and Ostroff [27], and Nordio et al. [25] have formalized semantics for a subset of Eiffel, but these works do not include agents. Our encoding of the routines *precondition* and *postcondition* is based on previous work on pure routines by Darvas and Leino [8,7], and Leino and Müller [22].

8 Conclusions and Future Work

We have introduced a verification methodology to verify higher-order functions, and we have implemented an automatic verifier for function objects. The verifier takes an Eiffel program, translates it to Boogie2, and uses the Boogie verifier to prove the generated code. Our experiments with automatic proofs indicate that the methodology is able to specify and verify function objects by introducing side-effect free routines which model abstractly the pre- and postcondition of the function objects. The experience so far suggests that a complete verification chain leading to fully automatic verification of object-oriented programs with function objects is possible.

Although presented in Eiffel, the verification methodology is not dependent on a specific programming language; we see no major obstacles in applying it to other languages supporting function objects.

As future work we plan to extend the framing methodology to handle non-interference. In particular, we plan to extend the implementation to be able to prove library classes such as *linked lists*. Furthermore, we plan to investigate how to apply a similar methodology to generics (in particular *constrained generics*).

Acknowledgements

We thank Stephan van Staden and Manuel Oriol for their insightful comments on drafts of this paper. Calcagno was supported by EPSRC.

References

1. A. Banerjee, D. Naumann, and S. Rosenberg. Regional Logic for Local Reasoning about Global Invariants. In *ECOOP*, volume 5142 of *LNCS*. Springer-Verlag, 2008.
2. M. Barnett, R. Leino, and W. Schulte. The Spec# Programming System: An Overview. In *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, volume 3362 of *LNCS*, pages 49–69. Springer, 2004.
3. M. Berger, K. Honda, and N. Yoshida. A logical analysis of aliasing in imperative higher-order functions. In *ICFP '05: Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*, pages 280–293. ACM, 2005.

4. B. Biering, L. Birkedal, and N. Torp-Smith. BI-hyperdoctrines, higher-order separation logic, and abstraction. *ToPLAS*, 2008.
5. E. Börger, N. G. Fruja, V. Gervasi, and R. F. Stärk. A high-level modular definition of the semantics of c#. *Theor. Comput. Sci.*, 336(2-3):235–284, 2005.
6. D. Clarke and S. Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. In *OOPSLA '02*, volume 37, pages 292–310. ACM Press, 2002.
7. Á. Darvas. *Reasoning About Data Abstraction in Contract Languages*. PhD thesis, ETH Zurich, Switzerland, 2009. To appear.
8. A. Darvas and K. R. M. Leino. Practical reasoning about invocations and implementations of pure methods. In *FASE*, LNCS. Springer-Verlag, 2007.
9. D. Distefano and M. J. Parkinson. jStar: Towards Practical Verification for Java. In *OOPSLA '08: Proceedings of the 23rd ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 213–226, 2008.
10. EVE: Eiffel Verification Environment. http://eve.origo.ethz.ch.
11. R. B. Findler and M. Felleisen. Contracts for higher-order functions. *SIGPLAN Not.*, 37(9):48–59, 2002.
12. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison Wesley, 1994.
13. K. Honda, N. Yoshida, and M. Berger. An observationally complete program logic for imperative higher-order frame rules. In *LICS '05: Proceedings of the Symposium on Logic in Computer Science*, pages 260–279, USA, 2005. IEEE Computer Society.
14. B. Jacobs. *A Statically Verifiable Programming Model for Concurrent Object-Oriented Programs*. PhD thesis, Katholieke Universiteit Leuven, 2007.
15. K. Rustan M. Leino. Specification and verification of object-oriented software. Marktoberdorf International Summer School 2008, lecture notes.
16. K. Rustan M. Leino. This is boogie 2. Technical Report Manuscript KRML 178, Microsoft Research, 2008.
17. J. Kanig and J.-C. Filliâtre. Who: A Verifier for Effectful Higher-order Programs. In *ACM SIGPLAN Workshop on ML*, Edinburgh, Scotland, UK, 2009.
18. I. T. Kassios. Dynamic Frames: Support for Framing, Dependencies and Sharing Without Restrictions. In *FM 2006: Formal Methods*, pages 268–283, 2006.
19. N. Krishnaswami, J. Aldrich, and L. Birkedal. Modular verification of the subject-observer pattern via higher-order separation logic. In *FTJP*, 2007.
20. 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.
21. K. R. M. Leino and P. Müller. Object invariants in dynamic contexts. In M. Odersky, editor, *ECOOP*, volume 3086 of *LNCS*, pages 491–516. Springer-Verlag, 2004.
22. K. R. M. Leino and P. Müller. Verification of equivalent-results methods. In *ESOP*, volume 4960 of *LNCS*, pages 307–321. Springer-Verlag, 2008.
23. P. Müller and J. N. Ruskiewicz. A modular verification methodology for C# delegates. In U. Glässer and J.-R. Abrial, editors, *Rigorous Methods for Software Construction and Analysis*, To appear.
24. M. Nordio, C. Calcagno, B. Meyer, and P. Müller. Reasoning about Function Objects. Technical Report 615, ETH Zurich, 2008.
25. M. Nordio, C. Calcagno, P. Müller, and B. Meyer. A Sound and Complete Program Logic for Eiffel. In M. Oriol, editor, *TOOLS-EUROPE 2009*, volume 33 of *Lecture Notes in Business and Information Processing*, pages 195–214, 2009.
26. P. W. O'Hearn, H. Yang, and J. C. Reynolds. Separation and information hiding. In *POPL '04*, pages 268–280, 2004.

27. R. Paige and J. Ostroff. ERC: an Object-Oriented Refinement Calculus for Eiffel. *Formal Aspects of Computing*, 16:51–79, 2004.
28. M. Parkinson and G. Bierman. Separation logic and abstraction. In *POPL '05*, volume 40, pages 247–258. ACM, 2005.
29. M. J. Parkinson and G. M. Bierman. Separation logic, abstraction and inheritance. In *POPL '08*, pages 75–86. ACM, 2008.
30. Y. Régis-Gianas and F. Pottier. A hoare logic for call-by-value functional programs. In *MPC '08: Proceedings of the 9th international conference on Mathematics of Program Construction*, pages 305–335. Springer-Verlag, 2008.
31. J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, 2002.
32. B. Schoeller. *Making classes provable through contracts, models and frames.* PhD thesis, ETH Zurich, 2007.