Reasoning about Function Objects

Martin Nordio[1], Cristiano Calcagno[2], Bertrand Meyer[1], and Peter Müller[3]

(1) ETH Zurich, Switzerland
{Martin.Nordio, Bertrand.Meyer}@inf.ethz.ch

(2) Imperial College, London, UK
ccris@doc.ic.ac.uk

(3) Microsoft Research, USA
mueller@microsoft.com

ETH Technical Report Nr. 615

February 2009

Abstract

Modern object-oriented languages support higher-order implementations through function objects such as delegates in C#, agents in Eiffel, or function objects 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 equips each function object with side-effect free (pure) methods for its pre- and postcondition, respectively. These pure methods can be used to specify client code relatively to the contract of the function object. We demonstrate the expressiveness of our approach through several non-trivial examples. It can be combined with any verification technique that supports pure methods, as illustrated by our experiments with Spec#.

Contents

1 Introduction

Object-oriented design makes a clear choice in dealing with the basic duality between objects and operations (data and functions): it bases system architecture on the object, more precisely the object types as represented by classes, and attaching any operation 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 *function objects*.

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 [7]) events of a certain type, by registering a procedure to be executed in response:

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 an agent representing a procedure with two integer arguments, and the mouse coordinates x and y. Since the agent 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 agent, and any target and argument object.

Agents (we will stay with this term but much of the discussion applies to other language variants) 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. First, 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. Next, an agent represents a true routine abstraction with an operation to call the underlying routine.

These mechanisms have proved attractive to object-oriented programmers but they also raise new verification challenges: how do we prove programs taking advantage of them? These challenges have been solved for functional languages. However, these solutions cannot be applied to object-oriented languages with their use of the heap and side effects.

To answer these requirements we introduce a specification and verification technique. Our approach uses side effect free (pure) routines to specify the pre- and postcondition of agents. To specify routines that take agents as arguments, we use these pure routines. Using previous work on pure routines [6, 13], these routines are encoded as mathematical functions, which yields the value of the agent pre- and postcondition. The basic idea, developed in the following sections, is that to prove a property of an agent call, $a.call(t, arg)$[1], it suffices to prove that the precondition of the agent a holds before the invocation, and then we can assume that the postcondition of a holds.

The three main technical contributions are: the idea of using pure routines to model the agent pre- and postcondition; a specification and verification methodology for function objects; and the demonstration of the approach's practicality through a set of proofs, of a sequence of examples of increasing difficulty, including one previously described as an open problem.

Although we focus on Eiffel agents, it should be straightforward to apply the results to mechanisms addressing similar goals in other languages, in particular, C# delegates. One restriction, however, is that any target language must be equipped or extended with contracts to enable formal reasoning; in particular, the approach relies on the assumption that it is possible to query a function object for its precondition and postcondition.

Section 2 presents example applications of agents and their verification challenge. Section 3 describes the verification method. Section 6 applies the method to the examples from Section 2. Section 7 reports the application of these proofs through an automatic prover. Section 8 discusses related work; Section 9 summarizes the result and describes future developments.

[1] To simplify the notation, we use a slight variant of the Eiffel.

2 Agent Examples and their Verification Challenge

We present some typical applications of agents. To simplify the notation, we assume agents are procedures and have at most one argument.

2.1 Formatter

The first example comes from a paper by Leavens et al. [11] 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
end

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

```
align_right (p: PARAGRAPH)
  require not p. right_aligned
  do
    ... Operations on p ...
  ensure p.right_aligned
  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. The signature *proc: PROCEDURE [FORMATTER, PARAGRAPH]* declares a procedure *proc* with two open arguments (the target of type *FORMATTER* and a parameter of type *PARAGRAPH*). Open arguments are the arguments provided in the invocation of the agent. An example of the use of the *format* routine is shown in the routine *apply_align_left*. This routine is implemented as follows:

```
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 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, we 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*, we should be able to show that the postcondition of *align_left* holds without modifying the proof of *format*.

2.2 Multi-Level Undo-Redo

The command pattern [7] can be used to implement multi-level undo-redo mechanisms. The standard implementation uses a class *COMMAND* with features *execute* and *cancel*. This example involves a history list, of type *LIST [COMMAND]*, such that is possible to undo all previously recorded commands through the following routine:

```
undo_all ( history_list : LIST [COMMAND] )
 do
```

```
    history_list . do_if(agent {COMMAND}.cancel, agent {COMMAND}.cancelable)
end
```

The routine *cancelable* returns true if the command can be canceled; in other words it satisfies the precondition of *cancel*. Iteration routines such as *do_if*, *do_all*, *for_all*, and *exists* are available to all list classes through their declaration in the ancestor class *LINEAR*, where a typical declaration is:

```
do_if (f: PROCEDURE [ANY]; test: PREDICATE [ANY])
    do
        from start until after loop
            if test.item (item) then f.call(item) end
            forth
        end
    end
```

The declaration *f: PROCEDURE [ANY]* indicates that *f* can come from any class and takes no arguments besides the target; similarly for *test*. (*ANY* is the most general class, from which all classes descend; cf. "Object" in Java.) The loop follows a standard scheme moving a cursor: *start* brings the cursor to the first element, *forth* moves it by one position, *after* indicates whether the cursor is past the last element, and *item* gives the element at cursor position.

The verification challenge in this example is to reason about the pre- and postcondition of the agent applied to several objects (in this example the elements of the list), where each agent invocation changes the properties of a single object. Verifying these kinds of examples is challenging because the invocation of the agent on one target might change the properties of the other targets. The use of multiple targets also illustrates one of the differences between agents and delegates in C#: applying an agent to multiple objects requires the ability to pass function objects with open target.

2.3 Archive Example

In this section we describe the *archive* example presented by Leavens et al. [11] and proved by Müller and Ruskiewicz [14]. This example illustrates the application of agents with closed arguments (closed arguments are the arguments of an agent provided at declaration of the agent).

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

```
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[ANY;TAPE];
                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
```

The invocation *log_file.call*(*s*) invokes the procedure *log_file* with the parameter *s*. The declaration *PROCEDURE[ANY;TAPE]*[2] indicates that *log_file* is a procedure 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*.

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*. The ability to query an agent object for its precondition and postcondition is important for the verification framework, and must be available or emulated in the transposition of the present work to any other object-oriented language.

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. First, we introduce the functions[3] *$precondition* and *$postcondition* to model the agents pre- and postcondition. Then, we show the assumptions and assertions that are generated when an agent is initialized and called. The methodology is extended for framing in Section 4. Sections 3.3 and 4.2 extend the methodology for closed arguments.

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

[3]We use the prefix $ in the mathematical functions to distinguish them with the Eiffel routines.

3.2.1 Agent Pre- and Postconditions.

The methodology uses two functions to model the pre- and postcondition of the agent. The function *$precondition* takes three values (the agent, the target and the parameter) and the current heap, and yields the evaluation of the agent's precondition. The function *$postcondition* takes a second heap to evaluate old expressions. The signature of these functions are defined as follows:

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

3.2.2 Initializing Agents.

Given the agent initialization a := **agent** pr where pr is a procedure, 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.

3.2.3 Invoking Agents.

Invoking an agent a with target t and argument p, $a.call(t, p)$, first asserts the precondition of the agent, and then assumes the postcondition. The proof obligations are the followings:

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 the postcondition. The **havoc** command assigns an arbitrary value to the heap.

This translation is based on the translation of pure routines [6, 13]. 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.

3.2.4 Noninterference.

Agents can be declared with open arguments. If the target is open, the same agent can be invoked with different target objects. To reason about these invocations, we need the notion of noninterference. Given a partial function $f : Heap \rightharpoonup X$, we call *footprints* the elements of its domain. In this work we only consider functions f such that, for each heap h on which f is defined, there exists a (unique) minimal sub-heap h_0, so that the value of f on lager heaps is completely determined from its value on h_0. Functions of type $Heap \rightharpoonup X$ are obtained from preconditions and bodies of agents by fixing the target and parameter, and from postconditions by also fixing the old heap (we are interested in the footprint expressed in terms of the new heap only).

Given functions $f : Heap \rightharpoonup X$ and $g : Heap \rightharpoonup Y$, we write the noninterference predicate $f \# g : Heap \rightarrow Bool$ which returns *true* iff both f and g are defined and their minimal footprints are disjoint.

We now lift the disjointness predicate $\#$ to objects. Let C be a class and F the set of state functions (preconditions, postconditions with fixed pre-heap, and bodies of features) which it provides. For $o, o' \in ObjectId$ objects of class C, we define

$o \# o' : Heap \rightarrow Bool$
$(o \# o')(h) \triangleq \forall v, v' \in Value, \forall f, f' \in F.\ (f(o, v) \# f'(o', v'))(h)$

The role of the # predicate is to generalize from concrete mechanisms for establishing noninterference, namely ownership [5, 12], separation logic [18, 16], regional logic [1]. The idea is that each such formalism is sufficiently expressive to imply instances of $o\#o'$ facts on a per-example basis.

3.3 Reasoning about Closed Arguments

The above two sections present a methodology to reasoning about agents with open arguments. In this section, we extend the methodology to agents with closed arguments. Framing for agents with closed arguments is omitted here, however, it is presented in Section 4.2.

To model closed arguments, we introduce two functions: $\$precondition_1$ and $\$postcondition_1$[4]. These functions yield the evaluation of pre- and postcondition of an agent with one closed argument (either closed target or closed parameter). The function $\$precondition_1$ takes two values (the agent and the open argument) and the current heap, and yields the evaluation of the precondition of the agent. The function $\$postcondition_1$ takes also a second heap to evaluate old expressions. The signature of the functions are defined as follows:

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

To handle arbitrary number of arguments in a routine, say n, the methodology can be extended by adding the functions $\$precondition_0 ... \$precondition_n$ and the functions $\$postcondition_0 ... \$postcondition_n$. The functions

$$\$precondition_i : Value^{i+i} \times Heap \rightharpoonup Bool$$
$$\$postcondition_i : Value^{i+i} \times Heap \times Heap \rightharpoonup Bool$$

can be used to model agents with i open arguments.

3.3.1 Initializing Agents.

To handle closed arguments, the methodology generates new assumptions using the functions $\$precondition_1$ and $\$postcondition_1$. In the following, we present these assumptions for closed target and closed arguments.

Closed Target. Given the agent initialization a := **agent** $t_1.pr$ where t_1 is the closed target, and pr a procedure, the methodology generates the following assumptions:

assume $\forall p : ObjectId; h_1 : Heap : \$precondition_1(a, p, h_1) = \$pre_{pr}(t_1, p, h_1)$
assume $\forall p : ObjectId; h_1, h_2 : Heap : \$postcondition_1(a, p, h_1, h_2) = \$post_{pr}(t_1, p, h_1, h_2)$

These assumptions quantify only over one parameter, p. The target object t_1 is known, and it is used in the function $\$pre_{pr}$. The difference with the assumptions generated for open arguments (Section 3.2) is that the assumptions for open arguments quantify over both the target and the parameter.

Closed Parameter. Given the agent initialization a := **agent** $pr(p_1)$ where p_1 is the closed parameter, and pr a procedure, the methodology generates the following assumptions:

assume $\forall t : ObjectId; h_1 : Heap : \$precondition_1(a, t, h_1) = \$pre_{pr}(t, p_1, h_1)$
assume $\forall t : ObjectId; h_1, h_2 : Heap : \$postcondition_1(a, t, h_1, h_2) = \$post_{pr}(t, p_1, h_1, h_2)$

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

3.3.2 Invoking Agents.

The invocation of an agent with closed arguments takes as arguments the agent and the open parameter. Given the agent a which declares a procedure with one open argument (it can be open target or open parameter) the agent invocation $a.call(p)$ with argument p, defines the following proof obligations:

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

where *Heap* denotes the current heap.

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 open arguments, and then it uses the assumptions generated in the initialization of the agent. An example of the application of open arguments is presented in Section 6.3.

4 Framing

This section presents a solution for framing. First, framing is solved for agents with open arguments, and then the methodology is extended for agents with closed arguments.

4.1 Framing for Agents with Open Arguments

One of the most interesting part of routines' 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 solved for example using dynamic frames [9, 19]. However, this problem has to be solved for routines that take other routines as arguments (agents). For example, in the routine *format* presented in Section 2.1, one need to define what locations this routine modifies:

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

A candidate solution to this problem is to assume that *format* modifies the target 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 any routine, and each routine might modify different locations.

To solve the frame problem for agents, we adapt dynamic frames. Instead of using a set of locations as in Kassios's work [9], we define a routine *modifies* (in the source language) which takes an agent a, its target and argument's values, and returns the locations modified by the agent a with target t and argument p. 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 of the routine used to invoke *format*, the function *modifies* will yield a different set of locations.

4.1.1 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 routines declarations is the following:

$$
\begin{array}{lll}
\mathit{modifies\_clause} & ::= & \mathit{modifies\_clause}, \mathit{modifies\_clause} \\
 & & \mid \mathit{VarId} \\
 & & \mid \mathit{modifies}(\mathit{VarId}, \mathit{VarId}, \mathit{VarId}) \\
\mathit{routine} & ::= & \mathit{RoutineId}\ (\mathit{VarId} : \mathit{Type}) : \ \mathit{Type} \\
 & & \quad \texttt{require}\ \mathit{boolExp} \\
 & & \quad \texttt{modify}\ \mathit{modifies\_clause} \\
 & & \quad \texttt{do} \\
 & & \quad\quad \mathit{instr} \\
 & & \quad \texttt{ensure}\ \mathit{boolExp} \\
 & & \quad \texttt{end}
\end{array}
$$

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

4.1.2 Encoding of Modifies Clauses.

To encode the modifies clauses, we introduce a function $\$\mathit{modifies}$ which takes an agent a, its target and argument's 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:

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

For example, the modifies clause of the routine *format* can be encoded using this function as follows:

free 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 object o, and all field *fId* that are not modified by the agent *proc* with the target f and argument *Current*, then the value of the field $o.\mathit{fId}$ in the current heap is equal to the value of $o.\mathit{fId}$ in the old heap. A *free ensures postcondition* is a postcondition that is assumed by the callers, and it does not have to be proven when the implementation is verified. The expression $\mathit{Heap}[o, \mathit{fId}]$ yields the value of the field *fId* of the object o in the current heap, and *Heap* denotes the current heap.

Generalizing, modifies clauses are list of application of the function $\$\mathit{modifies}$ and variable identifiers. Given the modifies clause:
$\$\mathit{modifies}(a_1, t_1, p_1), ..., \$\mathit{modifies}(a_n, t_n, p_n), v_1, ..., v_m$

this clause is encoded as:

free ensures $\forall o : \mathit{ObjectId};\ \mathit{fId} : \mathit{FieldId} :$

$$\left(\begin{array}{l} \mathit{not}\ \$\mathit{modifies}(a_1, t_1, p_1, \mathit{Heap}, o, \mathit{fId}) \\ \wedge ... \wedge \\ \mathit{not}\ \$\mathit{modifies}(a_n, t_n, p_n, \mathit{Heap}, o, \mathit{fId}) \\ \wedge\ o! = v_1\ \wedge\ o! = v_m \end{array}\right) \Rightarrow\ \mathit{Heap}[o, \mathit{fId}] = \mathit{old}(\mathit{Heap})[o, \mathit{fId}]$$

4.1.3 Initializing Agents.

To solve the frame problem for agents, one need to link the function $\$\mathit{modifies}(\mathit{proc}, t, p)$ with the locations that the routine *proc* modifies. We solve this by applying the same approach to

reasoning about agent's pre- and postcondition. Thus, our methodology generates assumptions of the function $\$modifies$, when the agent is initialized. Given a procedure pr, the agent initialization $a := \textbf{agent}\ pr$ generates the following assumptions:

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

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

The assumptions for the functions $\$precondition$ and $\$postcondition$ are the same assumptions described in Section 3.2. The third assumption relates the 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. 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 follows:

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

Generalizing, the function $modifies_{pr}$ takes a target object, an argument, and the object id and field id. The definition of this function can be generated from the modifies clause of each procedure pr.

4.2 Framing for Agents with Closed Arguments

4.2.1 Modifies Clauses.

To define the locations that an agent with closed arguments modifies, we introduce a function $modifies_1$. This function takes an agent a and its open argument, and returns the locations modified by the agent a with the argument p. The definition of the modifies clause is extended as follows:

$$\begin{array}{lll} modifies\_clause & ::= & modifies\_clause, modifies\_clause\\ & & \mid VarId\\ & & \mid modifies(VarId, VarId, VarId)\\ & & \mid modifies_1(VarId, VarId)\end{array}$$

Using the function $modifies_1$, the modifies clause of the routine *log* (Section 2.3) can be defined as follows:

```
log ( log_file : PROCEDURE [ANY;TAPE];
          s: STRING)
    modify
        modifies1( log_file ,s)
    do
        log_file . call(s)
    end
```

4.2.2 Encoding of Modifies Clauses.

The encoding of the modifies clauses follows the same ideas of the above section. We define a function $\$modifies_1$ which takes an agent a, its open argument's value, the current heap, an object value o, and a field name fId, and yields true if the agent a with argument p modifies the field $o.fId$. The signature of this function is the following:

$$\$modifies_1 : Value \times Value \times Heap \times Value \times FieldId \rightharpoonup Bool$$

For example, the modifies clause of *log* can be encoded as follows:

free ensures $\forall o : ObjectId;\ fId : FieldId :$
$not\ \$modifies_1(log\_file, s, Heap, o, fId) \Rightarrow Heap[o, fId] = old(Heap)[o, fId]$

4.2.3 Initializing Agents.

To conclude with the framing for closed arguments, the methodology generates assumptions in a similar way to the above section. We describe the assumptions generated for initialization of agents with closed target, closed parameter is analogous.

Given the agent initialization a := **agent** $t_1.pr$ where t_1 is the closed target, and pr a procedure, the methodology generates the following assumptions:

assume $\forall p : ObjectId; h_1 : Heap : \$precondition_1(a, p, h_1) = \$pre_{pr}(t_1, p, h_1)$
assume $\forall p : ObjectId; h_1, h_2 : Heap :\ \$postcondition_1(a, p, h_1, h_2) = \$post_{pr}(t_1, p, h_1, h_2)$

assume $\forall o, p : ObjectId;\ fId : FieldId :$
$\$modifies_1(a, p, o, fId) = \$modifies_{pr}(t_1, p, o, fId)$

5 Extending the Methodology for Functions

In the above section, we have described a verification methodology for agents assuming the agents are prodecures. In this section, we extend the methodology to function agents. We describe it using agents with open arguments, however, the methodology for agents with closed arguments is analogous.

5.1 Basics

Since functions return a result, they can access to an special variable *Result*. For example, a function *sum* that adds two integers can be write as follows:

```
sum (op1, op2: INTEGER): INTEGER
    do
        Result := op1 + op2
    ensure
        Result := op1 + op2
    end
```

If we apply the same approach defined in the above sections, one would introduce a function $postcondition_{sum}$ to express the postcondition of *sum*. This function takes only the arguments and the target of the object. However, if we invoke $x := sum(4, 5)$ we would need to replace *Result* by x. To model this, we introduce a new argument to the function postcondition. Then, we can write the postcondition of the invocation $x := sum(4, 5)$ as $postcondition_{sum}(Current, 4, 5, x)$.

The same problem raise with function agents: one need to add the result value of the agent. Thus, we introduce a new function $postcondition_R$ which takes an extra parameter: the result. Its signature is the following:

$$\$postcondition_R : Value \times Value \times Value \times Value \times Heap \times Heap \rightharpoonup Bool$$

5.1.1 Initializing Agents.

Given the agent initialization a := **agent** f, then the methodology generates the following assumptions:

assume $\forall t, p : ObjectId; h_1 : Heap : \$precondition(a, t, p, h_1) = \$pre_f(t, p, h_1)$
assume $\forall t, p, r : ObjectId; h_1, h_2 : Heap :$
$\$postcondition_R(a, t, p, r, h_1, h_2) = \$post_f(t, p, r, h_1, h_2)$

where $\$pre_f$ and $\$post_f$ denotes the pre- and postcondition of the function f, t the target object, p the argument, and r the result of the function. Note that $post_f$ takes an extra argument: the result of the function.

5.1.2 Invoking Agents.

Agent functions are invoked using the routine *item*. The invocation of an agent a with target t and argument p, $x := a.item(t, p)$, first asserts the precondition of the agent, and then assumes the postcondition hold with result x. The proof obligations are defined as follows:

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

5.2 Framing for Functions

Invocations of agent functions can be used in contracts. For example, a routine r can define the following precondition:

```
r (f: FUNCTION [ANY, ANY; BOOLEAN])
    require
        f.item (t,  p)
```

To solve the frame problem, one need to know what are the locations the agent f reads. To know the locations that the agent f reads, we introduce the function $\$reads$. This function takes an agent a, its target and argument's values, the current heap, an object value o, and a field name f, and yields true if the agent a with target t and argument p reads the field f of the object o. The signature of this function is the following:

$$\$reads : Value \times Value \times Value \times Heap \times Value \times FieldId \rightharpoonup Bool$$

Then, the location that the expression $f.item(t, p)$ reads is defined by:

$$\$reads(f, t, p)$$

5.2.1 Initializing Agents.

Applying the same approach to reason about framing for procedures, we generate assumptions for the function $\$reads$ when the agent is initialized. Given the agent initialization a := **agent** f, then the methodology generates the following assumptions:

assume $\forall t, p : ObjectId; h_1 : Heap : \$precondition(a, t, p, h_1) = \$pre_f(t, p, h_1)$
assume $\forall t, p : ObjectId; h_1, h_2 : Heap : \$postcondition(a, t, p, h_1, h_2) = \$post_f(t, p, h_1, h_2)$

assume $\forall o, t, p : ObjectId;\ fId : FieldId :$
$\$reads(a, t, p, o, fId) = \$reads_f(t, p, o, fId)$

where $\$reads_f$ yields true if the function f reads the field $o.fId$ for the target t and argument p.

This concludes the methodology to reasoning about functions objects. The methodology uses four side-effect free (pure) functions to express the agent pre and postcondition, and to express what the agent reads and modifies. Next section presents applications of the methodology.

6 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.

6.1 Formatter Example

In this section, we show how to verify the *formatter* example presented in Section 2.1. This routine generates the following proof obligations:

$format(proc : PROCEDURE[FORMATTER, PARAGRAPH];\ f : FORMATTER)$
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 **assert** $\$postcondition(proc, f, current, Heap, h_0)$

The agent invocation is translated in the lines 2-5. The pre- and postcondition of *format* are translated in the lines 1 and 6 respectively. The proof is straightforward since the assume and assert instructions in lines 1-2 and 5-6 refer to the same heap.

The most interesting case in the verification of function object is the verification of clients that use function object, such as *apply_align_left*. Applying our methodology to this routine generates the following assumptions and assertions:

$apply\_align\_left(f : FORMATTER; p : PARAGRAPH)$
1 **assume not** $p.\$left\_aligned$
2 $a :=$ **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 **assert** $\$precondition(a, f, p, Heap)$
6 $h_0 := Heap$
7 **havoc** $Heap$
8 **assume** $\$postcondition(a, f, p, Heap, h_0)$
9 **assert** $p.\$left\_aligned$

Similar to the previous example, lines 1 and 9 are generated by the translation of the pre- and postcondition. The declaration **agent** $\{FORMATTER\}.align\_left$ generates lines 2-4. The precondition and postcondition of the routine *align_left* is denoted by $\$pre_{align\_left}$ and $\$post_{align\_left}$ respectively. The invocation of the routine *format* produces lines 5-8. The current heap is stored in h_0 in line 6 to be able to evaluate the postcondition in line 8.

The key points in the proof are the assert instructions at lines 5 and 9. By the definition of $\$pre_{align\_left}$ and $\$post_{align\_left}$, we know:

$$\forall t_1, p_1 : ObjectId; h : Heap : \$pre_{align\_left}(t_1, p_1, h) = \ not\ p_1.\$left\_aligned \quad (1)$$

$$\forall t_1, p_1 : ObjectId; h, h' : Heap : \$post_{align\_left}(t_1, p_1, h, h') = \ p_1.\$left\_aligned \quad (2)$$

In particular, $\$pre_{align\_left}(f, p, Heap) = \ not\ p.\$left\_aligned$. Then, the assertion at line 5 is proven using the assumptions at lines 1 and 3, and (1). The assertion at line 9 is proven in a similar way using the assumptions at lines 4 and 8, and (2).

6.2 Multi-Level Undo-Redo Example

In this section, we discuss how to prove the routine *do_if* presented in Section 2.2. The proof of routine *cancel_all* is similar to *apply_align_left*. Due to space restrictions, the proof of *cancel_all* is omitted (but it is presented in our technical report [15]).

The first step is to give an specification of the routine *do_if*. The idea of $do\_if(f, test)$ is to execute f on all the elements which satisfy test. There is, however, a problem in specification of this routine in the EiffelBase library: *test* can be arbitrary and it might not imply that the precondition of f holds. A first try at improving the contract of the routine *do_if* could be the following:

```
do_if (f: PROCEDURE [ANY]; test: FUNCTION [ANY])
  require
      forall 1≤i≤count: test(ith(i)) implies f.precondition(ith(i))
  ensure
      forall 1≤i≤count: ( old test(ith(i)) implies f.postcondition(ith(i)) )
```

This contract uses two more features of the class *LINEAR*: *ith* and *count*. Function *ith* returns the i-th element of the current structure, and attribute *count* contains the length of the structure.

However, this improved contract is still not sufficient. Consider list $l = [c, c]$ where c satisfy the query *cancelable*. Consider an invocation of *do_if* with the agents *cancel* and *cancelable*. Since we know that the precondition of *cancel* holds for all the elements of the list l, we can invoke the routine. Thus, the first agent is invoked. But assume that this invocation breaks the property *cancelable*, then since the list contains two repeated elements, the second agent invocation does not satisfy *cancelable* and an exception is trigged.

The root of the problem lies in the fact that the invocation of the agent on one element of the list could break the precondition of the next element. To prevent the problem we must impose further conditions. We do that using the noninterference predicate # presented in Section 3.2. These assertions are then treated as proof obligations which need to be *discharged* by appropriate mechanisms in the target language. Example mechanisms are richer type systems based on ownership [2, 5] (as in our experiments with Spec#), or richer program logics based on separation logic [18, 16]. In other words, the # operator specifies that objects do not interfere (they occupy disjoint memory in case of separation logic, or they belong to different contexts in case of ownership). Here, we use the property that agent invocations only modify the target object, and that noninterference still holds after the invocation. Using this extension, we can have another go at writing the contract for *do_if*:

```
do_if (f: PROCEDURE [ANY]; test: FUNCTION [ANY])
  require
      forall 1≤i≤count: test(ith(i)) implies f.precondition(ith(i)) and
      forall 1 ≤ i < j ≤ count: ith(i) # ith(j)
  ensure
      forall 1≤i≤count: ( old test(ith(i)) implies f.postcondition(ith(i)) )
```

The new precondition says in addition that there is no interference between the elements of the list. We now present a sketch of the proof of the routine *do_if*. Section 7 shows how this example is encoded and automatically proved in Spec#.

Let *do_if_pre* be the precondition of the routine *do_if* and *loop_invariant* be the loop invariant defined as follows:

$$
\begin{aligned}
&do\_if\_pre \triangleq\ Inv(\alpha)\ \wedge\ forall\ 1 \le i < j \le count : ith(i) \# ith(j)\ \wedge \\
&\quad forall\ 1 \le i \le count : ((\alpha_i.\$test \Rightarrow \$precondition(f, \alpha_i, Heap)) \wedge (\alpha_i.\$test\ = \beta_i)) \\
&loop\_invariant \triangleq\ 1 \le j \le count + 1\ \wedge\ forall\ j \le i < k \le count : ith(i) \# ith(k)\ \wedge \\
&\quad forall\ 1 \le i \le j - 1 : (\beta_i \Rightarrow \$postcondition(f, \alpha_i, Heap, h_0)\ \wedge \\
&\quad forall\ j \le i \le count :\ (\alpha_i.\$test\ \Rightarrow \$precondition(f, \alpha_i, Heap))
\end{aligned}
$$

We use the auxiliary variable α to represent the current structure, which is a sequence of length *count*. The i-th element is denoted α_i. The expression $Inv(\alpha)$ denotes the invariant of the class parameterized by the sequence α. To translate away the **old** operator, we introduce auxiliary variable β, also denoting a sequence of length *count*. In the precondition, we assume $\alpha_i.\$test = \beta_i$. In the postcondition, expression **old** $\alpha_i.test$ is translated as β_i.

Figure 1 presents the sketch of the proof of the routine *do_if*. The auxiliary variable j represents the index of the current structure. After the invocation *f.call(item)*, we have to show that there is still no interference between the elements of the list. This is exactly the property of the operator #, which we have introduced in Section 3.2. The assertion at line 6 is proved using the loop invariant. The loop invariant is reestablished using the property of the # operator.

```
do_if (f : PROCEDURE[ANY]; test : FUNCTION[ANY])
  do
2   assume do_if_pre
    h0 := Heap
4   from start until after loop
      if test(item) then
6       assert $precondition(f, αj, Heap)
        h1 := Heap
8       f.call(item)
        assume $postcondition(f, αj, Heap, h1)
10    end
      forth
12    assert loop_invariant
    end
14  assert ( Inv(α) ∧ forall 1 ≤ i < k ≤ count : ith(i)#ith(k) ∧
           (forall 1 ≤ i ≤ count : (βj ⇒ $postcondition(f, αj, Heap, h0))) )
  end
```

Figure 1: Sketch of the proof of the routine *do_if*.

6.3 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. 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):

```
log(log_file : PROCEDURE[ANY; TAPE]; s : STRING)
  1  assume $precondition1(log_file, s, Heap)
  2  assert $precondition1(log_file, s, Heap)
  3  log_file.call(s)
```

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, we can prove the assert instruction at line 7.

```
main(c : CLIENT)
  1  create t.make
  2  assert t.$is_loaded
  3  a := agent t.store
  4  assume ∀p1 : ObjectId; h : Heap :
           $precondition1(a, p1, h) = $pre_store(a, t, p1, h)
  5  assume ∀p1 : ObjectId; h, h' : Heap :
           $postcondition1(a, p1, h, h') = $post_store(a, t, p1, h, h')
  6  assert $precondition1(a, "HelloWorld", Heap)
  7  c.log(a, "HelloWorld")
```

7 Automatic Proofs

The goal of the verification effort is to support automatic verification of programs using function objects. Although an implementation of a complete tool chain is still work in progress, we were able to perform significant experiments through the Boogie [3] verifier.

To apply Boogie, we converted the Eiffel examples into Spec# [3]. Although the main reason for this decision is that our Eiffel proof framework was not yet able to meet our needs, another motivation was to check the language independence of the proof technique. After translation, we used the Boogie verifier to prove the correctness of the examples. For the present work the trans-

lation from Eiffel to Spec# was performed manually. The translation is mostly straightforward, the only aspect whose automation would require work is the annotation of modifies clauses (and annotations related to Spec#'s ownership system).

Proving the examples requires having an implementation of the Eiffel class *ROUTINE* in Spec#. We have written a Spec# interface *Routine* with methods *precondition*, *postcondition* and *call*. To pass agent expressions as arguments, we provide a class, implementing the *Routine* interface, for every agent expression. For example, in the formatter example, the translation of **agent** {*FORMATTER*}.*align_left* relies on a class *AlignLeftRoutine* implementing *Routine*; one may then pass an object of type *AlignLeftRoutine* to represent the original **agent** {*FORMATTER*}.*align_left*.

The routines *precondition*, *postcondition* and *call* in the class *AlignLeftRoutine* are implemented by invoking the precondition, postcondition and the routine *align_left* itself. The method *postcondition* needs extra work to eliminate the **old** keyword. We solved this problem by introducing an extra field in the class. After addition of suitable modifies clauses to express what locations the method modifies, Boogie was able to prove the Spec# versions of all the previous examples (Section 6) automatically.

The example of Section 6.2 requires a proof of noninterference. To prove this example in Spec#, we used ownership [5, 12]. The technique represents the structure of the class *LINEAR* using an array, where each element is owned by the current structure (elements are *rep*), and all array elements are different. This made the automatic proof of the Spec# version possible after addition of type annotations. The example of Section 6.3 is encoded in a similar way to the *formatter* adding the functions $precondition_1$ and $postcondition_1$.

Using the same techniques we have proved a significant number of other examples, including all those presented by Leavens et al. [11]. The details of our proofs using Spec# can be found in our technical report [15].

8 Related Work

Jacobs [8] as well as Müller and Ruskiewicz [14] 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 [12] 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, avoids the complications and restrictions of ownership and can be generalized to several closed parameters more easily.

Parkinson and Bierman [17] 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. They are similar to the predicates that we use for the preconditions and postconditions of agents. Even though Parkinson and Bierman's work does 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 [10]. In contrast to separation logic, the methodology presented in this paper works with standard first-order theorem provers.

A key issue of reasoning about object-oriented programs is framing, that is, how to conclude which heap changes affect which predicates. In this paper, we simply assumed a noninterference predicate # without prescribing a particular way of enforcing it. Suitable candidates are separation

logic [18, 16], dynamic frames [9, 19], or regions [1]. Separation logic offers separating conjunction to express noninterference. Both dynamic frames and regions effect specifications for predicates and routines.

Our encoding of the routines *precondition* and *postcondition* is based on previous work on pure routines by Darvas and Leino [6], and Leino and Müller [13].

9 Conclusions and Future Work

We have introduced a verification methodology to verify higher-order functions. To invoke function objects, we use the ordinary call mechanism through the special routine *call* of class *ROUTINE*, with its contract given by precondition and postcondition. Our attempts at automatic proofs (preceded by manual translation) suggest that the methodology is able to specify and verify function objects by introducing side effect free routines which model 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. Clearly a number of links must still be filled to make this chain a reality, in particular removing the limitations and simplifications described in this article, and automating the steps that are still currently manual. This work is now proceeding as part of the development of an Eiffel Verification Environment (EVE) at ETH.

Although tried out on 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, an increasingly popular mechanism for modern programming languages, object-oriented or not.

References

[1] A. Banerjee, D. Naumann, and S. Rosenberg. Regional Logic for Local Reasoning about Global Invariants. In *ECOOP*, volume 5142 of *LNCS*, pages 387–411. Springer-Verlag, 2008.

[2] M. Barnett, R. Deline, M. Fähndrich, R. Leino, and W. Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology (JOT)*, 3(6):27–56, 2003.

[3] 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.

[4] B. Biering, L. Birkedal, and N. Torp-Smith. BI-hyperdoctrines, higher-order separation logic, and abstraction. *ToPLAS*, 2008. To appear.

[5] D. Clarke and S. Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. In *OOPSLA '02*, volume 37, pages 292–310. ACM Press, November 2002.

[6] A. Darvas and K. R. M. Leino. Practical reasoning about invocations and implementations of pure methods. In *FASE*, volume 4422 of *LNCS*, pages 336–351. Springer-Verlag, 2007.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison Wesley, 1994.

[8] B. Jacobs. *A Statically Verifiable Programming Model for Concurrent Object-Oriented Programs.* PhD thesis, Katholieke Universiteit Leuven, 2007.

[9] I. T. Kassios. Dynamic Frames: Support for Framing, Dependencies and Sharing Without Restrictions. In *FM 2006: Formal Methods*, pages 268–283, 2006.

[10] N. Krishnaswami, J. Aldrich, and L. Birkedal. Modular verification of the subject-observer pattern via higher-order separation logic. In *Formal Techniques for Java-like Programs*, 2007.

[11] 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.

[12] K. R. M. Leino and P. Müller. Object invariants in dynamic contexts. In M. Odersky, editor, *European Conference on Object-Oriented Programming (ECOOP)*, volume 3086 of *LNCS*, pages 491–516. Springer-Verlag, 2004.

[13] K. R. M. Leino and P. Müller. Verification of equivalent-results methods. In *ESOP*, volume 4960 of *LNCS*, pages 307–321. Springer-Verlag, 2008.

[14] 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.

[15] M. Nordio, C. Calcagno, B. Meyer, and P. Müller. Reasoning about Function Objects. Technical report, ETH Zurich, 2008. available at http://www.se.inf.ethz.ch/people/nordio/files2/NordioCalcagnoMeyerMueller08.pdf.

[16] P. W. O'Hearn, H. Yang, and J. C. Reynolds. Separation and information hiding. In *POPL '04*, pages 268–280, 2004.

[17] M. J. Parkinson and G. M. Bierman. Separation logic, abstraction and inheritance. In *POPL '08*, pages 75–86. ACM, 2008.

[18] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, 2002.

[19] J. Smans, B. Jacobs, and F. Piessens. Implicit dynamic frames. In *Formal Techniques for Java-like Programs*, 2008.

A More Applications

A.1 Formatter Example using Closed Arguments

This section shows a variant of the formatter example presented in Sections 2.1 and 6.1 using closed arguments. The *format* routine takes an agent with closed target of type *FORMATTER*, and open parameter of type *PARAGRAPH*:

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

The routine *apply_align_left* is implemented using closed target as follows:

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

To prove these routines, we apply the methodology described in the above section. The proof for the routine *format* is similar to the previous proof. The only change is the use of the function *target*:

$format(proc : PROCEDURE[FORMATTER, PARAGRAPH];\ f : FORMATTER)$

1 **assume** $precondition(proc, current, h_0)$
2 **assert** $precondition(proc, current, h_0)$
3 $proc.call(current)$
4 **assume** $postcondition(proc, current, h_0, h_1)$
5 **assert** $postcondition(proc, current, h_0, h_1)$

The new proof for the routine *apply_align_left* adds a new assumption (line 3). This assumption is used to proof the assert instructions at lines 6 and 9.

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

1 **assume not** $p.\$left\_aligned$
2 $a :=$ **agent** $f.align\_left$
3 **assume** $\forall p_1 : ObjectId; h : Heap :$
$\$precondition(a, p_1, h) = \$pre_{align\_left}(f, p_1, h)$
4 **assume** $\forall p_1 : ObjectId; h, h' : Heap :$
$\$postcondition(a, p_1, h, h') = \$post_{align\_left}(f, p_1, h, h')$
5 **assert** $\$precondition(a, p, Heap)$
6 $h_0 := Heap$
7 $p.format(a, f)$
8 **assume** $\$postcondition(a, p, Heap, h_0)$
9 **assert** $p.\$left\_aligned$

A.2 Fold Left

We have presented three examples that show the challenges of verifying object functions. The first example, the *formatter* example, illustrates how to prove function objects in an abstract way, using the pure methods *precondition* and *postcondition*. The second one, the multi-level undo-redo example, shows the application of agents to a sequence of objects, where every invocation affects the state of a single object. To prove this example, we use the noninterference predicate #. Finally,

the last example, the archive example, illustrates the problem of verifying function objects with open arguments. To fulfill this goal, we introduce the pure methods *target* and *parameter*.

This section describes a forth example, typical of higher-order languages, where the invocation on each element of a list produces a result which is dependent on the result of the previous invocation. This example can be found in your technical report [15].

The *fold_left* routine of class *LIST* takes as arguments a procedure which implements a binary operation in-place on the receiver object, and an initial object. The call, where represents the list, executes the following steps

$$Result := init;\ f.call(Result, o_1);\ \ldots\ f.call(Result, o_n);$$

If for example $f(n)$ belongs to a class with an integer field *val* and the body of f is $val := val+n$, the final result will be $((init + o_1) + \cdots o_n)$. The Eiffel implementation of *fold_left* is the following.

```
fold_left  (f: PROCEDURE [ANY, ANY], init: ANY) : ANY
 do
   from start; Result := init until after loop
     f. call (Result, item)
     forth
   end
 end
```

The verification challenge here is finding an abstract way to describe the state change after each application of *f*, and being able to show that the state change is applied n times, where n is the length of the list.

Proof of the Fold Left Example

The first difficulty in giving a specification for this example is finding an abstract way to describe the state change after each application of the procedure *f*. Our solution is to define a function $compute : G \times G \rightarrow G$ to model the abstract state transitions, and a redefined class *PROCEDURE2* for function objects which manipulate abstract states.

The next challenge is to specify the result of repeated applications of *call* on the elements of the list. This would normally require the use of inductive predicates. We introduce an auxiliary[5] recursive function *inv* to compute the result of applying the agent to the first n elements of the list:

$$\begin{aligned} &inv : Integer \times G \rightarrow G \\ &inv(0, v) = v \\ &inv(n, v) = ith(n).compute(inv(n-1, v), ith(n)) \end{aligned}$$

Class *PROCEDURE2* redefines the specification of procedure *call* to model the intuition that invoking *call* corresponds to performing an abstract state transition.

```
call  (invok: ANY, p: ANY)
  ensure then invok = invok.compute (old invok, p)
```

The **ensure then** clause extends the postcondition of the parent with the requirement that the effect of *call* is described by function *compute*.

As in the *do_if* example, noninterference assertions $ith(i)\#ith(j)$ are proof obligations that agent invocations on one element of the list do not affect properties of other elements of the list. A sketch of the proof, including the loop invariant, is presented in Figure 2. The only interesting step is the implication used to re-establish the loop invariant, which uses one unrolling of the definition of *inv*.

[5] auxiliary predicates and functions are purely proof artifacts, and must be side-effect free so that they don't change the semantics of the program.

```
fold_left (f : PROCEDURE2[ANY, ANY]; init : ANY) : ANY
  do
    start
    Result := init
    assert Result = inv(0, init)
    until
      after
    loop
      assert Result = inv(index, init)
      f.call(Result, item)
      assume Result = Result.compute(inv(index, init), item)
      forth
      assert Result = inv(index, init)
    end
    assert Result = inv(count, init)
  end
```

Figure 2: A sketch of the proof of routine *fold_left*.

B Experiments in Spec#

This section presents the encoding into Spec# of the examples described in Section 2 and Appendix A. The formatter example, the multi-level undo-redo and the archiver example presented in Section 2 are encoded to Spec# in Sections B.1, B.2 and B.3. Then, Section B.4 proves the fold left example described in Appendix A. Finally, SectionsB.5 and B.6 shows more experiments in Spec# of applications of agents.

B.1 Formatter Example

To be able to encode the examples into Spec#, we need to write an interface *Routine*. This interface models a routine with the routines *precondition*, *postcondition* and *call*. After creating the interface *Routine*, we need to extend this class to model routines of target type T. The interface *RoutineT* models a routine of target object T. For instance, in the formatter example, the routine *format* takes a routine of target type *Formatter*. Thus, the Spec# class *ProcedureFormatter* models a procedure with target type *Formatters*. This class declares the methods *pre*, *post*, *call* and *getTarget* as follows:

```
using System; using Microsoft.Contracts;

/*
The class ProcedureFormatter defines an Eiffel procedure with target type Formatters.
*/
public interface ProcedureFormatter {
    [Pure] Formatters getTarget();

    [Pure] bool pre();

    [Pure] bool post();

    void call ()
        requires this.pre() ;
        modifies this.*;
        ensures this.post() && getTarget()==old(getTarget());
}
```

To create agent expressions, the source language uses the expression **agent** {T}.f. To avoid extending the Spec# verifier and adding a new translation of agent expressions, agent expressions are modeled using a class *ProcedureTF* where T is the type of the target object and F is the routine. For example, the class *ProcedureFormatterAlignLeft* models the agent expression **agent** {Formatter}.AlignLeft. Thus, an agent **agent** {Formatter}.AlignLeft can be created using an object of type *ProcedureFormatterAlignLeft*.

The class *ProcedureFormatterAlignLeft* implements an agent AlignLeft. The methods *pre*, *post* and *call* are implemented by invoking the precondition and postcondition of align left. These methods are implemented using the methods *alignLeftPre*, *alignLeftPost* and *alignLeft* respectively. The class is implemented in Spec# as follows:

```
public class ProcedureFormatterAlignLeft: ProcedureFormatter {
    [Rep] public Formatters! target;
    [Rep] public Paragraph! argument;

    public ProcedureFormatterAlignLeft([Captured] Formatters! t, [Captured] Paragraph! p)
        ensures target == t && p==argument;
    {
```

```
        target = t;
        argument = p;
    }

    [Pure] public Formatters getTarget()
        ensures result == target;
    {
        return target;
    }

    [Pure] public bool pre()
        ensures result == target.alignLeftPre();
    {
        return (target.alignLeftPre());
    }

    [Pure] public bool post()
        ensures result == (target.alignLeftPost());
    {
        return target.alignLeftPost();
    }

    public virtual void call ()
    {
        expose(this)
        {
            target.alignLeft(argument);
        }
    }
}
```

Similar to class *ProcedureFormatterAlignLeft*, the class *ProcedureFormatterAlignRight* implements an agent *AlignLeft*. The implementation is the following:

```
public class ProcedureFormatterAlignRight: ProcedureFormatter {
    [Rep] public Formatters! target;
    [Rep] public Paragraph! argument;

    public ProcedureFormatterAlignRight([Captured] Formatters! t, [Captured] Paragraph! p)
        ensures target == t && p==argument;
    {
        target = t;
        argument = p;
    }

    [Pure] public Formatters getTarget()
        ensures result == target;
    {
        return target;
    }

    [Pure] public bool pre()
        ensures result == target.alignRightPre();
    {
```

```
        return (target.alignRightPre());
    }

    [Pure] public bool post()
        ensures result == (target.alignRightPost());
    {
        return target.alignRightPost();
    }

    public virtual void call ()
    {
        expose(this)
        {
            target.alignRight(argument);
        }
    }
}
```

The class *Formatters* implements the formatters *alignLeft* and *alignRight*. The precondition and postcondition of these methods are implemented in the methods *alignLeftPre*, *alignLeftPost* and *alignRightPre*, *alignRightPost* respectivally. These pre- and postconditions are implemented in these methods to be able to implement the classes *ProcedureFormatterAlignLeft* and *ProcedureFormatterAlignRight*. The formatter is implemented as follows:

```
public class Formatters {
    [SpecPublic] [Rep] bool align; // true: left; false: right

    public void alignLeft(Paragraph p)
        requires alignLeftPre();
        ensures alignLeftPost();
    {
        align = true;
    }

    [Pure] public bool alignLeftPre()
        ensures result == (!align);
    {
        return !align;
    }

    [Pure] public bool alignLeftPost()
        ensures result == (align);
    {
        return align;
    }

    public void alignRight(Paragraph p)
        requires alignRightPre();
        ensures alignRightPost();
    {
        align = false;
    }

    [Pure] public bool alignRightPre()
```

```
        ensures result == (align);
    {
        return align;
    }

    [Pure] public bool alignRightPost()
        ensures result == (!align);
    {
        return !align;
    }
}
```

The class *Paragraph* implements a paragraph with a method format. This method takes a procedure f with target of type *Formatter* (*ProcedureFormatter*! f) and invokes it. The method *format* requires $f.pre$ and ensures $f.post$.

```
public class Paragraph {
    char [][] text;
    int width;

    public void format(ProcedureFormatter! f)
        requires f.pre();
        modifies f.*;
        ensures f.post() && f==old(f) &&
            f.getTarget() == old(f.getTarget()) ;
    {
        f.call();
    }
}
```

Finally, we present an example of the use of class *Paragraph*. First, *format* is called with the agent align left. Spec# was able to prove that the postcondition of the agent align left holds after the invocation. Then, the method *format* is called with the agent align right, and Spec# was also able to prove that the postcondition of align right holds after the invocation.

```
public class UsingParagraph {
    [SpecPublic] [Rep] public Paragraph! p;
    [SpecPublic] [Rep] public Formatters! f1, f11;
    [SpecPublic] [Rep] public Formatters! f2;

    public UsingParagraph ()
    {
        p = new Paragraph();
        f1 = new Formatters ();
        f11 = new Formatters ();
        f2 = new Formatters ();
    }

    public void exampleParagraph()
    {
        expose (this) {
            Formatters! f = new Formatters ();
            Paragraph! p2 = new Paragraph();
```

```
            assume f.alignLeftPre();
            ProcedureFormatterAlignLeft pfal = new ProcedureFormatterAlignLeft (f,p2);

            p.format (pfal);

            assert pfal.target.alignLeftPost();

            Formatters! f2 = new Formatters ();
            Paragraph! p3 = new Paragraph();
            ProcedureFormatterAlignRight pfal2 = new ProcedureFormatterAlignRight (f2,p3);

            assume f2.alignRightPre();
            p.format (pfal2);

            assert pfal2.target.alignRightPost();
        }
    }
}
```

B.2 Multi-Level Undo-Redo

The process of encoding and proving the multi-level undo-redo example is similar to the formatter example. First, we implement the class *ProcedureCommand* which models procedures with target of type *Command*.

The precondition and postcondition of the procedure is represented with the methods pre and post. The method call invokes the current procedure. It requires pre holds and ensures post. The class is implemented as follows:

```
using System; using Microsoft.Contracts;

public interface ProcedureCommand {

    [Pure] bool pre();

    [Pure] bool post();

    void call ()
        requires this.pre() ;
        ensures this.post() ;
}
```

The class *ProcedureCommandExecute* represents the agent *execute* (declared in the class *Command*). A target of type *Command* is declared in this class. The method *pre* returns *target.executePre*, and *post* returns *executePost*. The method *call* invokes the execute method. The implementation is the following:

```
public class ProcedureCommandExecute: ProcedureCommand {
    [Rep] public Command! target;


    public ProcedureCommandExecute([Captured] Command! t)
        ensures target == t;
    {
        target = t;
    }
```

```
    [Pure] public bool pre()
        ensures result == target.executePre();
    {
        return (target.executePre());
    }

    [Pure] public bool post()
        ensures result == (target.executePost());
    {
        return target.executePost();
    }

    public virtual void call ()
    {
        expose(this)
        {
            target.execute();
        }
    }
}
```

The class *ProcedureCommandUndo* represents the agent undo. This class is implemented in a similar way to ProcedureCommandExecute. The implementation is the following:

```
public class ProcedureCommandUndo: ProcedureCommand {
    [Rep] public Command! target;


    public ProcedureCommandUndo([Captured] Command! t)
        ensures target == t;
    {
        target = t;
    }

    [Pure] public Command! getTarget()
        ensures ( result == target);
    {
        return target;
    }

    [Pure] public bool pre()
        ensures result == target.undoPre();
    {
        return (target.undoPre());
    }

    [Pure] public bool post()
        ensures result == (target.undoPost());
    {
        return target.undoPost();
    }

    public virtual void call ()
    {
```

```
        expose(this)
        {
            target.undo();
        }
    }
}
```

The class *Command* implements a command with the operations *execute* and *undo*. This class stores the current number of executions in the field *executions* and the maximum number of executions in the field *maxExecutions*. The precondition and postcondition of the method *execute* is implemented in the methods *executePre* and *executePost* respectively. The pre- and postcondition of *undo* is implemented in a similar way. The class *Command* is implemented in Spec# as follows:

```
public class Command {
    public int executions;
    [SpecPublic] int oldExecutions;
    [SpecPublic] int maxExecutions;

    public Command ()
        ensures (executions == 0) && (maxExecutions == 20);
    {
        executions = 0;
        maxExecutions = 20;
    }

    public virtual void execute ()
        requires executePre();
        modifies oldExecutions, executions;
        ensures executePost();
    {
        oldExecutions = executions;
        executions ++;
    }

    [Pure] public bool executePre ()
        ensures result == (executions < maxExecutions);
    {
        return executions < maxExecutions;
    }

    [Pure] public bool executePost ()
        ensures result == (executions == oldExecutions + 1);
    {
        return (executions == oldExecutions + 1);
    }

    public virtual void undo ()
        requires undoPre();
        modifies executions, oldExecutions;
        ensures undoPost();
    {
        oldExecutions = executions;
        executions   ;
    }

    [Pure] public bool undoPre ()
```

```
        ensures result == (executions > 0);
    {
        return executions > 0;
    }

    [Pure] public bool undoPost ()
        ensures result == (executions == oldExecutions 1);
    {
        return (executions == oldExecutions 1);
    }

    public virtual void dec ()
        modifies executions;
        ensures (executions == (old(executions)) 1) &&
                maxExecutions == old(maxExecutions);

    {
        executions ;
    }

    [Pure] public int getExc()
        ensures result == executions;
    {
        return executions;
    }
}
```

The class *Linear* implements the Eiffel class *LINEAR*. The current structure is stored in an array structure. The routine doOnce takes a generic procedure and invokes it.

```
public class Linear {
    [SpecPublic] [Rep] [ElementsRep] public ProcedureCommand[]! structure;

    public Linear ([Captured] ProcedureCommand! p1, [Captured] ProcedureCommand! p2, [
        Captured] ProcedureCommand! p3 )
        requires p1 != p2 && p2 != p3 && p1!=p3;
        ensures structure[0]==p1 && structure[1]==p2 && structure[2]==p3 &&
                p1==old(p1) && p2==old(p2) && p3==old(p3) &&
                structure.Length==3;
    {
        structure = new ProcedureCommand [3];
        structure [0] = p1;
        structure [1] = p2;
        structure [2] = p3;
    }

    public virtual void doOnce(ProcedureCommand! proc)
        requires proc.pre() ;
        modifies proc.*;
        ensures proc.post() &&
                proc == old(proc);
    {
        proc. call ();
    }

    public virtual void doAll2(ProcedureCommand! proc1, ProcedureCommand! proc2 )
```

```
        requires proc1.pre() && proc2.pre() ;
        requires proc1 != proc2;
        modifies proc2.*, proc2.*;
        ensures proc2.post() &&
                proc1.post() &&
                proc2 == old(proc2) &&
                proc1 == old(proc1);
    {
        proc1. call ();

        assert proc2.pre();
        proc2. call ();
        assert proc2.post();
    }

    invariant forall {int i in (0: structure.Length);
                      structure[i] != null};
    invariant forall {int i in (0: structure.Length), int j in (0: structure.Length);
                      i!=j ==> structure[i] != structure[j]} ;

}
```

The class *DoUndo* shows an example of the use of *LINEAR*. The method *executeOnce* invokes the agent *execute* using the method *doOnce*. The method *undoOnce* does the same using the agent *undo*.

The method *executeOnce* requires *c.executePre*. This method implements a two experiments. The first one, an object in a location different to *c* is called before calling the method *doOnce* with the agent *execute*. This experiment shows that calling other methods do not change the property of *c.executePre* and we can prove this program. The second experiments changes *c*. But the invocation *c.dec*() does violate the property *c.executePre*(). Thus we can also verify this program. The implementation is the following:

```
public class DoUndo {

    [SpecPublic] [Rep] Command! c1, c2;
    [Rep] Linear! l;

    public DoUndo ()
    {
        // init doOnce
        Command! cc1 = new Command();
        Command! cc2 = new Command();
        Command! cc3 = new Command();

        c1 = new Command();
        c2 = new Command();

        ProcedureCommandExecute! p1 = new ProcedureCommandExecute (cc1);
        ProcedureCommandExecute! p2 = new ProcedureCommandExecute (cc2);
        ProcedureCommandExecute! p3 = new ProcedureCommandExecute (cc3);
        l = new Linear(p1,p2,p3);
    }

    public virtual void executeOnce ()
        requires c2.getExc() > 2;
        modifies c2;
```

```
    {
        expose (this) {
            Command c = new Command();
            ProcedureCommandExecute! p1 = new ProcedureCommandExecute (c);
            assert c.executePre();

            c2.dec();
            c2.dec();

            l.doOnce(p1);
            assert p1.target.executePost();
        }

    }

    public virtual void UndoOnce ()
    {
        expose (this) {
            Command c = new Command();
            c.execute();
            assume c.executions==1;
            assert c.undoPre();
            ProcedureCommandUndo! p1 = new ProcedureCommandUndo (c);
            assert c.undoPre();

            l.doOnce(p1);
            assert p1.target.undoPost();
        }
    }
}
```

B.3 Archive Example

The class *ProcedureTapeDrive* represents the Eiffel class *PROCEDURE* for *TapeDrive*. The precondition and postcondition of the procedure are represented with the methods *pre* and *post*. The method *call* invokes the current procedure. The class can be implemented in Spec# as follows

```
using System; using Microsoft.Contracts;

public interface ProcedureTapeDrive {
    [Pure] TapeDrive! getTarget();

    [Pure] bool pre();

    [Pure] bool post();

    void call (string! o)
        requires this.pre() ;
        modifies this.*;
        ensures this.post();
}
```

Agents store is model using the class *ProcTapeStore*. The methods *pre*, *post* and *call* are implemented implemented in a similar way to the previous examples (by invoking the methods *StorePre*, *StorePost* and *Store* respectively).

```
public class ProcTapeStore: ProcedureTapeDrive {
    [SpecPublic] [Rep] TapeDrive! target;

    public ProcTapeStore([Captured] TapeDrive! t)
        ensures target == t;
    {
        target = t;
    }

    [Pure] public TapeDrive! getTarget()
        ensures result==target;
    {
        return target;
    }
    [Pure] public bool pre()
        ensures result == target.StorePre();
    {
        return (target.StorePre());
    }

    [Pure] public bool post()
        ensures result == (target.StorePost());
    {
        return target.StorePost();
    }

    public virtual void call (string! o)
    {
        expose(this)
        {
            target.Store((string) o);
        }
    }
}
```

The class *TapeDrive* implements a TapeDrive with methods store, eject and change media. The implementation is the following:

```
public class TapeDrive {
    [SpecPublic] [Rep] public bool IsLoaded ;
    [SpecPublic] [Rep] public bool IsStored ;

    public TapeDrive ()
        ensures IsLoaded && !IsStored;
    {
        IsLoaded = true;
        IsStored = false;
    }
    public void Store (object! p)
        requires StorePre();
        modifies IsStored;
        ensures StorePost();
    {
        // ...
        IsStored = true;
    }
```

```
[Pure] public bool StorePre()
    ensures result == IsLoaded;
{
    return IsLoaded;
}

[Pure] public bool StorePost()
    ensures result == (IsStored && IsLoaded);
{
    return (IsStored && IsLoaded);
}

public void Eject ()
    requires EjectPre();
    modifies IsLoaded;
    ensures EjectPost() ;
{
    IsLoaded = false ;

}

[Pure] public bool EjectPre()
    ensures result == IsLoaded;
{
    return IsLoaded;
}

[Pure] public bool EjectPost()
    ensures result == (!IsLoaded);
{
    return !IsLoaded;
}

public void ChangeMedia ()
    requires ChangeMediaPre();
    modifies IsLoaded;
    ensures ChangeMediaPost();
{
    expose (this) {
        IsLoaded = false ;
        /* exchange media */
        IsLoaded = true ;
    }
}

[Pure] public bool ChangeMediaPre()
    ensures result == IsLoaded;
{
    return IsLoaded;
}

[Pure] public bool ChangeMediaPost()
    ensures result == IsLoaded;
```

```
    {
        return IsLoaded;
    }
}
```

The class *Client* is based on the example presented by Leavens et al. [11]. Instead of using delelates, we use ordinary variables of type *PROCEDURE*. This variable has the precondition, postcondition and target of the delegate. The method *Log* is generic, so it can be invoked with any procedure (in this case a procedure of type *ProcedureTapeDrive*).

```
class Client {

    public void Log(ProcedureTapeDrive! logFile, string! s)
        requires logFile.pre();
        modifies logFile.*;
        ensures logFile.post() ;
    {
        logFile.call(s) ;
        //assert false;
    }
}
```

Finally, we present an example of the use of *TapeDrive*. We frist create a *TapeDrive*, and then we invoke the *Client.Log*. Due to the agent satisfies its constract, then Spec# is able to prove this program. However, if we uncomment the invocation *tapeDrive.Eject*() the verifier cannot prove the program (because it is not correct).

```
class Example {
    [SpecPublic] [Rep] Client! c ;

    public Example ()
    {
        c = new Client();
    }

    public void main ()
    {
        expose(this) {
            TapeDrive! tape = new TapeDrive();

            //tape.Eject();
            ProcTapeStore! archiver = new ProcTapeStore(tape);
            c.Log(archiver, "hello") ;
        }

    }
}
```

B.4 Abstraction

This section presents the example described in Appendix A.2. The class *Procedure*2 models a routine which satisfy the abstract property *inv*. The class is implemented as follows:

```
using System; using Microsoft.Contracts;

public interface Procedure2 {
```

```
    [Pure] object getTarget();

    [Pure] bool inv(int i);

    void call (int i)
        requires inv(i) ;
        modifies this.*;
        ensures  inv(i+1) && getTarget()==old(getTarget());
}

public class Counter {

    public int val;

    public Counter ()
        ensures val==1;
    {
        val = 1;
    }

    public void succ ()
        ensures val == old(val) + 1;
    {
        val++;
    }
}

public class Counter2 {
    public int posval;
    public bool positive;

    [Pure] public int val()
        ensures result == (positive ? posval :  posval);

    {
        if(positive) return posval;
        else return posval;
    }

    public Counter2()
        ensures posval==1 && positive==true;
    {
        posval = 1;
        positive=true;
    }

    public void succ()
        ensures val() == old(val()) + 1;
    {
        if(positive) posval++;
        else posval ;
    }
}

public class ProcGeneral: Procedure2 {
```

```
    [SpecPublic] [Rep] public Counter! target;

    public ProcGeneral([Captured] Counter! t)
        ensures target == t;
    {
        target = t;
    }

    [Pure] public object getTarget()
        ensures result==target;
    {
        return target;
    }

    [Pure] public bool inv(int i)
        ensures result == (i == target.val );
    {
        return (i==target.val);
    }

    public virtual void call (int i )
    {
        expose(this)
        {
            target.succ();
        }
    }
}
public class ProcGeneral2: Procedure2 {
    [SpecPublic] [Rep] public Counter2! target;

    public ProcGeneral2([Captured] Counter2! t)
        ensures target == t;
    {
        target = t;
    }

    [Pure] public object getTarget()
        ensures result==target;
    {
        return target;
    }

    [Pure] public bool inv(int i)
        ensures result == (i == target.val() );
    {
        return (i==target.val());
    }

    public virtual void call (int i )
    {
        expose(this)
        {
            target.succ();
```

```
        }
    }
}
```

```
public class Threetimes {
    public void three (Procedure2! p)
        requires p.inv(1);
        modifies p.*;
        ensures p.inv(4) && p.getTarget() == old(p.getTarget());
    {
        p.call(1);
        p.call(2);
        p.call(3);
    }
}
```

```
public class test {
    [SpecPublic] [Rep] Threetimes! t;

    public test()
    {
        t = new Threetimes();
    }

    // here is the test
    public void example ()
    {
        expose(this)
        {
            Counter! c2 = new Counter();
            ProcGeneral pg = new ProcGeneral (c2);
            assert pg.target==c2;
            t.three(pg);
            assert pg.target==c2;
            assert pg.inv(4);
            assert pg.target.val==4;
            assert c2.val==4;
        }
    }

    public void example2()
    {
        expose(this)
        {
            Counter2! c2 = new Counter2();
            ProcGeneral2 pg = new ProcGeneral2(c2);
            assert pg.target==c2;
            t.three(pg);
            assert pg.target==c2;
            assert pg.inv(4);
            assert pg.target.val()==4;
            assert c2.val()==4;
        }
    }
```

```
}
```

B.5 Bank Account Example

The class *ProcedureBank* represents the Eiffel class *PROCEDURE* for target objects of type *BankAccount*. The precondition and postcondition are represented with the methods *pre* and *post*. The method *call* invokes the current procedure. The class is implemented as follows:

```
using System; using Microsoft.Contracts;

public interface ProcedureBank {
    [Pure] BankAccount! getTarget();

    [Pure] bool pre(int v);

    [Pure] bool post(int v);

    void call (int v)
        requires this.pre(v) ;
        modifies this.*;
        ensures this.post(v) && getTarget()==old(getTarget());
}
```

The class *ProcedureBankDeposit* represent the agent *deposit* (declared in the class *BankAccount*). A target of type *BankAccount* is declared in this class. The method *pre* returns *target.depositPre*, and *post* returns *depositPost*. The method *call* invokes the deposit method. Its implementation in Spec# is the following:

```
public class ProcedureBankDeposit: ProcedureBank {
    [SpecPublic] [Rep] BankAccount! target;

    public ProcedureBankDeposit([Captured] BankAccount! t)
        ensures target == t;
    {
        target = t;
    }

    [Pure] public BankAccount! getTarget()
        ensures result==target;
    {
        return target;
    }

    [Pure] public bool pre(int v)
        ensures result == (target.depositPre(v));
    {
        return target.depositPre(v);
    }

    [Pure] public bool post(int v)
        ensures result == (target.depositPost(v));
    {
        return target.depositPost(v);
    }

    public virtual void call (int v)
```

```
    {
        expose(this) {
            target.deposit(v);
        }
    }


}
```

Agents withdraw is modeled using the class *ProcedureBankWithdraw*. This class implements the interface *ProcedureBank* by invoking the precondition and postcondition of *withdraw*. Its implementation is Spec# is the following:

```
public class ProcedureBankWithdraw: ProcedureBank {
    [SpecPublic] [Rep] BankAccount! target;

    public ProcedureBankWithdraw([Captured] BankAccount! t)
        ensures target == t;
    {
        target = t;
    }

    [Pure] public BankAccount! getTarget()
        ensures result==target;
    {
        return target;
    }

    [Pure] public bool pre(int v)
        ensures result == (target.withdrawPre(v));
    {
        return target.withdrawPre(v);
    }

    [Pure] public bool post(int v)
        ensures result == (target.withdrawPost(v));
    {
        return target.withdrawPost(v);
    }

    public virtual void call (int v)
    {
        expose(this) {
            target.withdraw(v);
        }
    }
}
```

Bank accounts are implemented using the class BankAccount. The field *oldBalance* is used to refer to the old balance in the postcondition of deposit and withdraw:

```
public class BankAccount {
    [SpecPublic] [Rep] int balance;
    [SpecPublic] [Rep] int oldBalance;

    public BankAccount()
    {
```

```
        balance = 0;
        oldBalance = 0;
    }

    public void withdraw (int v)
        requires withdrawPre(v);
        ensures withdrawPost(v);
    {
        oldBalance = balance;
        balance = balance   v;
    }

    [Pure] public bool withdrawPre(int v)
        ensures result == (balance > v);
    {
        return balance > v;
    }

    [Pure] public bool withdrawPost(int v)
        ensures result == (balance == (oldBalance  v));
    {
        return balance == oldBalance  v;
    }

    public void deposit (int v)
        requires depositPre(v);
        ensures depositPost(v);
    {
        oldBalance = balance;
        balance = balance + v;
    }

    [Pure] public bool depositPre(int v)
        ensures result == (v > 0);
    {
        return v > 0;
    }

    [Pure] public bool depositPost(int v)
        ensures result == (balance == (oldBalance + v));
    {
        return balance == oldBalance + v;
    }
}
```

The class *Transfer* implements a simple transfer function. The function takes a procedure and invokes it with the argument *val*. This procedure can be instantiated with any procedure in bank account such that the procedure has one integer argument:

```
public class Transfer {
    public virtual void doTransfer(ProcedureBank! proc, int val)
        requires proc.pre(val) ;
        modifies proc.*;
        ensures proc.post(val) &&
                proc == old(proc) &&
                old (proc.getTarget())==proc.getTarget();
```

```
    {
        proc. call( val);
    }
}
```

Finally, we present an example of the use of the *Transfer* class. We first instantiate the transfer with the *deposit* procedure, and then we do the same with the procedure *withdraw*. Spec# has been able to show that the pre and postcondition of the agent holds before and after the agent invocation. This example is implemented as follows:

```
public class testTransfer {

    public virtual void transferOnce (int v)
    {
        expose (this) {
            BankAccount! b = new BankAccount ();
            Transfer! t = new Transfer();

            assume b.depositPre(v);
            ProcedureBankDeposit pbd = new ProcedureBankDeposit (b);
            t. doTransfer(pbd,v);
            assert pbd.getTarget() . depositPost(v);


            BankAccount! bb = new BankAccount ();
            assume bb.withdrawPre(v);
            ProcedureBankWithdraw pbd2 = new ProcedureBankWithdraw (bb);
            t. doTransfer(pbd2,v);
            assert pbd2.getTarget() . withdrawPost(v);
        }

    }

}
```

B.6 Simple Agent Invocation

This example implements a single agent invocation. In the method callingSingleAngent, the procedure p is assigned with the agent execute or undo depending whether b is true or not. This method ensures the postcondition of execute or undo holds (depending if b is true or false).

```
using System; using Microsoft. Contracts;


public class SingleAgent {
    [SpecPublic] [Rep] Command! c1;
    [SpecPublic] [Rep] Command! c2;

    public SingleAgent ()
    {
        c1 = new Command();
        c2 = new Command();
    }

    public virtual void callingSimpleAgent(bool b)
    {
```

```
        expose(this) {
            ProcedureCommand! p;
            if (b)
            {
                Command c = new Command();
                p = new ProcedureCommandExecute (c);
                assert c.executePre();
                assume p.pre();
                p.call();

                assert p.post();

            }
            else
            {
                Command c = new Command();
                p = new ProcedureCommandExecute (c);
                assert c.executePre();
                assume p.pre();
                p.call();

                assert p.post();
            }

            assert p.post();
        }
    }


}
```