JOURNAL OF OBJECT TECHNOLOGY
Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering. ©JOT, 2004

Vol. 3, No. 4
Special issue: TOOLS USA 2003

Conformance of agents in the Eiffel language

Philippe Ribet, **Cyril Adrian**, **Olivier Zendra** and **Dominique Colnet**
LORIA (INRIA - CNRS - University Henri Poincaré) Vandœuvre-lès-Nancy Cedex FRANCE

In Eiffel, the notion of agent makes it possible to describe and manipulate computation parts (i.e. operations) like ordinary data. Operations may be partially described, may be passed as ordinary data and may have their execution delayed. Agents are very convenient for many purposes, such as going through data structures and implementing call-backs in graphical libraries.
Although they can be seen as normal objects, they convey specific issues, pertaining to standard conformance rules for generic types. To get rid of existing problems, this paper proposes an adaptation of conformance rules for agents that provides much more flexibility while retaining all the benefits of a strong static typing system.

1 INTRODUCTION

Agents were introduced in the context of the Eiffel language in 1999, as an extension [DHM+99] for the previous definition of Eiffel [Mey92]. Although a number of details were provided for their typing, we realized when implementing agents in SmartEiffel[1], The GNU Eiffel Compiler (http://SmartEiffel.loria.fr) in Summer 2001 that major issues remained. Being among the first to implement agents in an Eiffel compiler and to actually use them (for iterators, for a graphical library still under work, etc.) enabled us to gather significant experience in this area, and made us find solutions to the uncovered issues.

This paper aims at presenting those solutions. It is organized as follows. First, section 2 presents the concept of agents in Eiffel. Section 3 then explains the severe issues that arise when using the usual conformance rules with agents. The solution we suggest is detailed in section 4; its goal is to tackle these issues and allow agents to smoothly merge with all other Eiffel concepts. Finally, section 5 concludes.

2 AGENTS: PRESENTATION

Overview

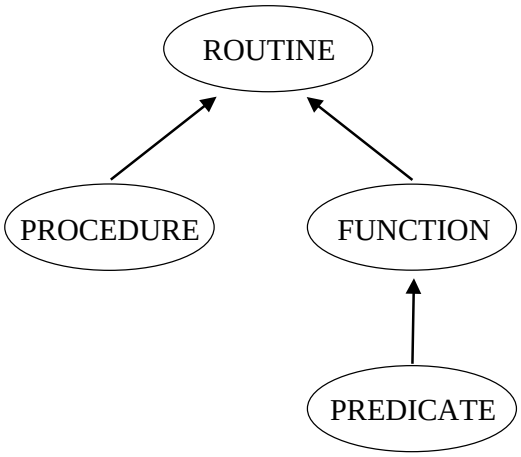
The agent mechanism [DHM+99, Mey00] gives the Eiffel object-oriented language the ability to handle operations, or commands, as such, like in functional program-

[1] Previously named SmallEiffel.

Cite this article as follows: Philippe Ribet, Cyril Adrian, Olivier Zendra, Dominique Colnet: "Conformance of agents in the Eiffel language", in *Journal of Object Technology*, vol. 3, no. 4, April 2004, Special issue: TOOLS USA 2003, pages 125–143, http://www.jot.fm/issues/issue_2004_04/article7

ming languages.

Agents are a new type of objects that allow to store code to be executed and data in an object, named the agent. In Eiffel, four types of agents exist: an abstract (deferred) type *ROUTINE*, and three concrete types *PROCEDURE*, *FUNCTION* and *PREDICATE*. The following figure shows their inheritance relationship:



Agents are a way of storing operations for later execution. They are objects. As such, they can be stored, compared to Void, or passed around to other software components. The operation stored in the agent may then be executed whenever the component decides. The most common uses of agents comprise delayed calls, multiple calls (on different values), lazy evaluation, and so on.

Using agents in Eiffel

Standard method calls are executed exactly "where they are written in the code". An agent, while having a syntax similar to a simple feature call, does not immediately call the method. Instead, an object is created, to be stored and used later. Only when it is used, will the agent trigger the feature call.

The created object has, like any Eiffel object, a well defined static type. The agent type (be it *ROUTINE* or one of its heirs as shown above) may be used to declare entities, such as attributes, feature parameters or a feature result.

For illustration, we use in this paper examples inspired from graphical user interface (GUI) programming. As the following example shows, GUI usage widely benefits from the power of the agent mechanism:

```
do
    ...
    my_window.when_pointer_move(agent print_coordinates(?,?)) --(1)
    ...
end
```

```
print_coordinates(x, y: INTEGER) is
   do
      io.put_integer(x) ; io.put_character(' ')
      io.put_integer(y) ; io.put_new_line
   end
```

The expression starting with the **agent** keyword on line (1) creates a new object, actually an agent object, instead of executing the print_coordinates method immediately. Note the pair of question marks (?,?) which denote the fact that the arguments of print_coordinates are not yet given. Still at line (1) this newly created agent object is passed to the when_pointer_move method to be memorized by the my_window object of class *WINDOW*. Thus the operation saved by the my_window object may be executed many times, with different arguments (e.g. each time the mouse pointer moves inside the *WINDOW*).

The *WINDOW* class is in charge of the agent memorization as well as the agent execution when the move event occurs. The following extract of the *WINDOW* class shows how to declare an attribute to store the agent (2) and then the usage of the call feature (3) to launch the execution:

```
delayed_action: PROCEDURE[ANY, TUPLE[INTEGER, INTEGER]]      -- (2)

when_pointer_move(action: like delayed_action) is
   do
      delayed_action := action
   end

pointer_move_dispatch(x, y: INTEGER) is
   do
      delayed_action.call([x, y])                            -- (3)
   end
```

The method when_pointer_move saves the agent, while the method pointer_move_dispatch executes it using the mouse pointer coordinates as arguments.

It is interesting to focus on the *PROCEDURE*[*ANY*, *TUPLE*[*INTEGER*, *INTEGER*]] type. It is the type of an agent that stores a procedure. This procedure may belong to (be defined in) any class. Such an agent has two open arguments of type *INTEGER*. An open argument is an argument whose value is unknown when the agent is created, but is provided at call time (when the agent is executed). Conversely, an agent may also have closed arguments, that is arguments known when the agent is created. Hence, the agent object does not only refer to a routine (as a mere function pointer would in other languages), but also contains additional information that becomes available as arguments of the executed routine. For example, the previous code can be changed this way:

```
    do
        ...
        my_window.when_pointer_move(
             agent position_message("Pointer moved to ", ?, ?) )
        my_window.when_left_click(
             agent position_message("Left click at ", ?, ?) )
        ...
    end

position_message(text: STRING; x, y: INTEGER) is
    do
        io.put_string(text)    ; io.put_integer(x)
        io.put_character(' ') ; io.put_integer(y)
        io.put_new_line
    end
```

Open arguments are symbolized by question marks, while closed arguments are directly stored in the agent object. This powerful system makes it possible to have delayed calls with values specific to each call (open arguments) and values specific to each agent but common to all executions of this agent (closed arguments). This mechanism is secure because each argument type is checked at compile time.

In the above example, the `pointer_move_dispatch` procedure will thus use the same call on all agents, even though some executed procedures require two formal arguments and others need three.

Another capability that makes agents very useful is that any existing method can be turned into an agent, without any change to its code. We could use for example `io.put_string`:

```
    do
        ...
        my_window.when_close( agent io.put_string("Bye bye%N") )
        ...
    end
```

Common agent use cases

The first goal of agents is to delay calls. Here is one example of such a use: let's imagine you have a dog, which is able to do what you tell it do do at noon. You may tell it to eat, to walk, to sleep, to get the newspaper...

Below is the code preparing the action the dog will do at lunch time (we ask the dog to eat `some_food` then). Follows an extract of the *DOG* class showing what it does with such an instruction (see the `do_lunch` action).

```
    do
        ...
        my_dog.at_lunch( agent my_dog.eat(some_food) )
        ...
    end

class DOG -- Extract

    at_lunch_action: PROCEDURE[ANY, TUPLE]

    at_lunch(action: PROCEDURE[ANY, TUPLE]) is
        do
            at_lunch_action := action
        end

    eat(food: FOOD) is
        do
            ...
        end

    do_lunch is
        do
            at_lunch_action.call([])
        end

end -- class DOG
```

Another common use of agents is repetitive action on a collection. For example, *ARRAY* features the `do_all` method whose code is:

```
class ARRAY[E] -- Extract

    do_all(action: ROUTINE[ANY,TUPLE[E]]) is
          -- Apply 'action' to every item of 'Current'.
        local
            i: INTEGER
        do
            from i := lower until i > upper
            loop
                action.call([item(i)])                    -- (4)
                i := i + 1
            end
        end
```

The following example shows the basic use of do_all:

```
zoo: ARRAY[ANIMAL]

foo is
   do
      ...
      zoo.do_all( agent print_name(?))                    -- (5)
      ...
   end

print_name(item: ANIMAL) is
   do
      io.put_string( item.name ) ; io.put_new_line        -- (6)
   end
```

In the above example, we display the name of each animal in the zoo. Indeed when zoo, an *ARRAY*[*ANIMAL*], is asked to do_all on line (5), it calls the print_name agent once for each item it contains (line (4)). Thus, each *ANIMAL* in zoo is passed as an argument to print_name, which then prints its name (line (6)).

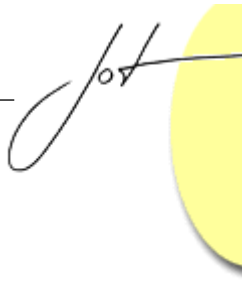
Agents also allow the receiver of the call (the target) to be an open argument. In this case, the open target is denoted by its type, like in the following adaptation of the previous example:

```
   do
      ...
      zoo.do_all( agent {ANIMAL}.do_lunch)
      ...
   end
```

Note that in this case, do_all is still used. Line (4) executes do_lunch using all the item(i) as successive *targets*, hence triggering do_lunch on every *ANIMAL*.

Agents are a way to pass code as an argument, but they may also be used for partial execution. In method calls, all parameters are evaluated before the call. If an actual parameter is **agent** object.method(arg), then it results in an agent object creation and method is not called on target object. Note that arg is evaluated when it is stored in the agent object. This agent may be executed later when requested. The following example shows partial code evaluation on an academic example:

```
   do
      ...
      ifthenelse( age > 12,
                  agent allow_access(current_discount),
```



```
                agent deny_access )
    ...
end

ifthenelse(cond: BOOLEAN; action1, action2: PROCEDURE[ANY, TUPLE]) is
   do
      if cond then
         action1.call([])
      else
         action2.call([])
      end
   end
```

All examples shown so far use *PROCEDURE*s; using agents of type *FUNCTION* is very similar. An agent of type *FUNCTION* requires one more generic parameter for the function result type. For example, with the following function definition in class *FOO*:

```
gt(item, value: INTEGER): BOOLEAN is
   do
      Result := item > value
   end
```

Various agent types may be used, as the following table shows:

Written Agent	Corresponding Agent Type
agent gt(?, ?)	*FUNCTION*[*FOO*, *TUPLE*[*INTEGER*, *INTEGER*], *BOOLEAN*]
agent gt(?, 3)	*FUNCTION*[*FOO*, *TUPLE*[*INTEGER*], *BOOLEAN*]
agent gt(1, ?)	*FUNCTION*[*FOO*, *TUPLE*[*INTEGER*], *BOOLEAN*]
agent gt(x, y)	*FUNCTION*[*FOO*, *TUPLE*, *BOOLEAN*]

Note that the *PREDICATE* type is just a shortcut for a *FUNCTION* with a *BOOLEAN* result type: *PREDICATE*[*A*, *B*] is equivalent to *FUNCTION*[*A*, *B*, *BOOLEAN*].

3 STANDARD EIFFEL CONFORMANCE RULES

Conformance usage

The Eiffel conformance rules are involved in assignments, be they direct or indirect.

For example, let's consider the following code:

```
a: A
b: B
```

```
...
foo(c: C; d: D) is
   do
      ...
   end
```

With these declarations, a direct assignment `a := b` is valid only if type B *conforms to* type A, while an indirect assignment `foo(a, b)` is valid only if type A *conforms to* type C and type B *conforms to* type D. Note that this is an indirect assignment because the call `foo(a, b)` assigns effective parameters to formal parameters: in order to initialize the formal parameters, `c := a; d := b` is performed when entering the `foo` routine.

The conformance rule in assignments is the base of the typing system. The goal is to be sure that *an object has a dynamic type which conforms to the static type of the entity used to access the object.*

The assignment attempt construct offers the possibility to write an assignment that would be invalid according to the previous rule based on static types. `a ?= b` is an assignment attempt. Such an instruction succeeds only if the dynamic type of the source `b` of the assignment conforms to the static type of the target `a`. The conformance rule is thus satisfied.

The next parts define the precise conformance rules in Eiffel.

Conformance with basic types

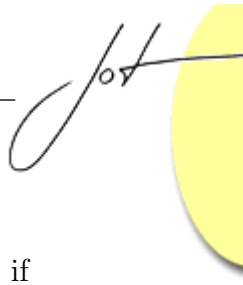
Very briefly, the main conformance rules are:

- any type conforms to itself,
- an expanded type conforms to the relative reference type,
- if types A and B are not expanded, and class B directly inherits from class A, then type B conforms to type A,
- the 'conforms to' property is transitive.

More details can be found in the Eiffel reference manual [Mey92].

Conformance with generic types

A type $G[F_1, F_2, ..., F_n]$ conforms to a type $G[E_1, E_2, ..., E_n]$ only if $\forall i \in [1..n]$ F_i conforms to E_i.



Conformance with *TUPLE* types

A type $TUPLE[F_1, F_2, ..., F_p]$ conforms to a type $TUPLE[E_1, E_2, ..., E_n]$ only if $n \leq p$ **and** $\forall i \in [1..n]$ F_i conforms to E_i.

Conformance with *ROUTINE* types

The *ROUTINE* types are generic types with more semantic. As *ROUTINE* types do not have their own conformance rules one may think that the *generic types* rules apply. We will hold true this assumption in this chapter, and show that we can make dogs eat tomatoes.

The *ROUTINE* type is a generic type with two formal type parameters: *ROUTINE*[*BASE*, *OPEN* –> *TUPLE*]. According to the conformance rule for generic types, a type $ROUTINE[B_1, O_1]$ conforms to a type $ROUTINE[B_2, O_2]$ only if B_1 conforms to B_2 and if O_1 conforms to O_2.

As mentioned in section 2, page 125, the *PROCEDURE* type inherits from *ROUTINE*, and has the same formal type parameters: *PROCEDURE*[*BASE*, *OPEN* –> *TUPLE*]. Its conformance rule is thus the same as for *ROUTINE*.

The *FUNCTION* type is a generic type with three formal type parameters: *FUNCTION*[*BASE*, *OPEN* –> *TUPLE*, *RESULT_TYPE*]. According to the conformance rule for generic types, a type $FUNCTION[B_1, O_1, R_1]$ conforms to a type $FUNCTION[B_2, O_2, R_2]$ only if B_1 conforms to B_2, O_1 conforms to O_2 and if R_1 conforms to R_2.

PREDICATE[*B*, *O*] type being just a shortcut for *FUNCTION*[*B*, *O*, *BOOLEAN*], and it has the same conformance rules as *ROUTINE*.

The next section shows these conformances within the context of various examples.

Applying conformance rules to examples: issues arise

Having precisely recalled the rules of conformance for the different types, we now apply these rules to an number of examples, in order to show that issues arise.

Let's start with the following code:

```
lunch_action: PROCEDURE[ANY, TUPLE[FOOD]]          -- (7)

do
    ...
    lunch_action := agent my_dog.eat(?)            -- (8)
    ...
    lunch_action.call([tomatoes])                  -- (9)
```

```
    ...
  end

class DOG -- Extract

  eat(meat: MEAT) is                                    -- (10)
    do
      ...
    end
```

If we consider line (8), the type of lunch_action is *PROCEDURE*[*ANY*, *TUPLE*[*FOOD*]] and the type of **agent** my_dog.eat(?) is *PROCEDURE*[*DOG*, *TUPLE*[*MEAT*]]. As *DOG* conforms to *ANY* and *TUPLE*[*MEAT*] conforms to *TUPLE*[*FOOD*] (because *MEAT* conforms to *FOOD*, being a subtype of it), the conformance rules of generic types implies that *PROCEDURE*[*DOG*, *TUPLE*[*MEAT*]] conforms to *PROCEDURE*[*ANY*, *TUPLE*[*FOOD*]]. Thus, according to the standard conformance rules, the assignment on line (8) is a valid one.

Using the same rules, line (9) is valid. The definition of call is call(o: *OPEN*); in this example, *OPEN* corresponds to *TUPLE*[*FOOD*], because of line (7). [tomatoes] is thus a valid argument.

As a consequence, the call on line (9) executes the eat method of class *DOG* (line (10)) with tomatoes as effective argument which does **not** conform to the formal argument type *MEAT*. This odd situation results in a very dangerous state, because the dynamic type of meat does not conform to its static type; this violates the conformance rule stated earlier. It seems reasonable to consider this a major problem.

Let us now formally demonstrate that the rule leads to conformance oddities. Our next example considers an agent as a delayed call. If an agent is executed where it is created (without any instruction between the agent creation instruction and the agent execution), then the agent call and its execution should have the same effect as a direct call and the properties should be similar.

Let's consider the following class definition:

```
class T -- Extract

  a: A
  b: B  -- with B conforming to A

  f(a1: A) is do ... end
  g(b1: B) is do ... end

  fa: PROCEDURE[T, TUPLE[A]] --      same type as 'agent f(?)'
  gb: PROCEDURE[T, TUPLE[B]] -- (11) same type as 'agent g(?)'
```

With the previous definition in mind, we now examine several calling sequences variants that should be equivalent:

```
  f(b)
⇕
  ( agent f).call([b])
⇕
  gb := agent f           -- (12)
  gb.call([b])            -- (13)
```

Writing the (12) assignment is appealing. Indeed, the type of gb, found on line (11), guarantees the method will be called with a parameter conforming to B (on line (13)). And the f method is precisely one that accepts such parameters (since it accepts A, to which B conforms). Thus, (13) should be valid and work as expected. However, the assignment on line (12) is forbidden by the current conformance rules in the language, because $PROCEDURE[T, TUPLE[A]]$ does not conform to $PROCEDURE[T, TUPLE[B]]$, since A does not conform to B.

This example makes it clear that the typing system may prevent writing perfectly valid calls, which is not satisfactory.

Let's now consider the following normally equivalent calling sequences:

```
  g(a)                    -- (14) invalid
⇕
  ( agent g).call([a])
⇕
  fa := agent g           -- (15)
  fa.call([a])            -- (16)
```

Line (14) is invalid, since method g requires a parameter of type B, not an A.

Conversely, line (15) is valid, because **agent** g is of type $PROCEDURE[T, TUPLE[B]]$, which conforms to $PROCEDURE[T, TUPLE[A]]$, the type of fa. Line (16) is also valid, since the provided parameter type is the one expected, A. But the result of running lines (15) and (16) is to execute the g method with a as parameter, which would be normally invalid in a direct call and may not reasonably be expected to succeed.

Thus, we can conclude that the normal conformance rules applied to routines makes it possible to defeat the typing system and perform invalid calls, which is an issue.

In the previous examples, we studied cases for the conformance of the parameters of the *TUPLE* type. But conformance with *TUPLE* types encompasses another

aspect: the number of parameters, or size of the tuple. The next examples pertain to this second aspect.

The code of the following example uses the one for the *WINDOW* class provided on page 127:

```
my_window.when_pointer_move( agent print_all )                -- (17)
my_window.when_pointer_move( agent print_x )                  -- (18)
my_window.when_pointer_move( agent io.put_string("move%N") ) -- (19)

print_x(x: INTEGER) is
   do
      io.put_integer(x) ; io.put_new_line
   end

print_all(x, y: INTEGER; t: TIME) is                          -- (20)
   do
      io.put_integer(x) ; io.put_character(' ')
      io.put_integer(y) ; io.put_character(' ')
      io.put_string(t.to_string) ; io.put_new_line
   end
```

On line (17), the type of **agent** print_all[2] is *PROCEDURE*[*T*, *TUPLE*[*INTEGER*, *INTEGER*, *TIME*]]. Since the argument type for when_pointer_move is *PROCEDURE*[*ANY*, *TUPLE*[*INTEGER*, *INTEGER*]] (see line (2) page 127), the indirect assignment on line (17) is allowed according to the previous conformance rules.

However problems are bound to arise when the agent is triggered and the call is executed. Indeed, the call on line (3) page 127 corresponds to action.call([x, y]) but the actually executed method print_all requires one more argument (line (20)). That is a dramatic error that is undetected by the type system, which is likely to cause trouble because this method needs information it will never get.

On line (18), the type of **agent** print_x is *PROCEDURE*[*T*, *TUPLE*[*INTEGER*]]. However, the argument type for when_pointer_move is *PROCEDURE*[*ANY*, *TUPLE*[*INTEGER*, *INTEGER*]] (line (2) page 127). The indirect assignment in line (18) is thus forbidden according to the previous conformance rules for *TUPLE*s seen on page 133. It may seem nonetheless acceptable, because more parameters are provided than required: the print_x method, when executed, will only use the first parameter.

Line (19) presents a case similar to that of line (18). This call is forbidden, but may be considered useful, with the executed method ignoring optional information it does not need.

[2] all arguments are open, is is a shortcut for agent print_all(?, ?, ?)

Conclusion

The above examples clearly show that the conformance rules presented in this section are not satisfactory. Some of these examples just point at some code that could be accepted but is not validated, thus proving the type system too cautious. This is safe, although not desirable for the sake of expressiveness.

However, a number of examples evidenced true issues, where incorrect code that is bound to fail is accepted by the type system. This is a major problem, that led us to design new, better rules, which are detailed in the next section.

4 NEW CONFORMANCE RULES

We demonstrated that not having rules for the special case of *ROUTINE* types (thus using the default "generic type" rules), was bound to raise many problems. In this chapter we will now present specific rules for the *ROUTINE* types and show how they solve those problems.

New rules definition

Our new conformance rules are specific to agents.

$ROUTINE[B_1,\ O_1]$ conforms to $ROUTINE[B_2,\ O_2]$ if the following two conditions hold:

- `BaseRule:` B_1 conforms to B_2
- `OpenRule:` O_2 conforms to O_1 (note the reversed conformance rule)

$FUNCTION[B_1,\ O_1,\ R_1]$ conforms to $FUNCTION[B_2,\ O_2,\ R_2]$ if the following three conditions hold:

- `BaseRule:` B_1 conforms to B_2
- `OpenRule:` O_2 conforms to O_1 (note the reversed conformance rule)
- `ResultRule:` R_1 conforms to R_2

Conformance for $PROCEDURE[B_1,\ O_1]$ and $PREDICATE[B_1,\ O_1]$ need the conditions `BaseRule` and `OpenRule` as for the *ROUTINE* type.

As indicated, our new conformance rules between O_1 and O_2 are the reverse of what they were in the normal ones (the conformance for generic types), erroneous conformance rules presented in section 3. All the other conformance rules (for basic types, generic types and *TUPLE* types) are unchanged and remain as they were in section 3.

Note that *the previous rules define conformance rules, this has nothing to do with covariance or contravariance.*

As a summary, a simple way to see and use these rules is to consider that when some method requires a formal argument of type *ROUTINE*[*BASE*, *OPEN*], the agent which will be provided as effective parameter is bound to be executed with arguments conforming to *OPEN*. So, providing a method able to handle such arguments is all that is necessary.

Applying new conformance to examples: issues are solved

This section details how our new conformance rules impact all the examples presented in section 3 and shows how they solve the issues that existed with the normal rules.

Let's start with the same example as the one presented on page 133:

```
lunch_action: PROCEDURE[ANY, TUPLE[FOOD]]

do
   ...
   lunch_action := agent my_dog.eat(?)      -- (21)
   ...
   lunch_action.call([tomatoes])            -- (22)
   ...
end

class DOG -- Extract

   eat(meat: MEAT) is
      do
         ...
      end
```

Line (21) is not valid anymore, because the type of **agent** my_dog.eat(?) is *PROCEDURE*[*DOG*, *TUPLE*[*MEAT*]] and does not conform to the type of lunch_action, which is *PROCEDURE*[*ANY*, *TUPLE*[*FOOD*]], according to our new reversed conformance rule OpenRule.

This code is thus now statically rejected. Line (22) is still type-valid, but since line (21) is not, lunch_action may not be an eat method that requires a *MEAT* as argument. Thus, thanks to our new conformance rules, the invalid code execution of page 133 is not possible anymore.

The next example considers an agent as a delayed call, and is based on rewriting code in different, but equivalent, ways. It was first shown on page 134.

```
class T -- Extract
```

```
a: A
b: B -- with B conforming to A

f(a1: A) is do ... end
g(b1: B) is do ... end

fa: PROCEDURE[T, TUPLE[A]] -- same type as 'agent f(?)'
gb: PROCEDURE[T, TUPLE[B]] -- same type as 'agent b(?)'
```

With the above definition for T, let's consider the same code variants as before:

```
  f(b)
⇕
  (agent f).call([b])
⇕
  gb := agent f           -- (23)
  gb.call([b])            -- (24)
```

Now, line (23) is allowed by our new conformance rules. Indeed, now, *PROCEDURE*[*T*, *TUPLE*[*A*]] conforms to *PROCEDURE*[*T*, *TUPLE*[*B*]], because `OpenRule` requires B to conform to A (which is trivially true). Line (24) will execute the `f` method with `b` as an argument, which is valid because `f` needs an argument conforming to A. This safe code is thus accepted now, which increases expressiveness.

Our third example, first shown on page 135, consists of the following normally equivalent code variants:

```
  g(a)                    -- invalid
⇕
  (agent g).call([a])
⇕
  fa := agent g           -- (25) now invalid
  fa.call([a])            -- (26)
```

Now, line (25) is not valid anymore, because `OpenRule` states that for *PROCEDURE*[*T*, *TUPLE*[*B*]] (the type of **agent** `g`) to conform to *PROCEDURE*[*T*, *TUPLE*[*A*]] (the type of `fa`), A must conform to B, which is of course not true.

This code is thus now correctly rejected. This is a safe situation, unlike that in section 3 because as explained there line (26) is valid and would execute method `g` with `a` as an effective argument while expecting a formal argument of type B. Our new rules thus prevent the bogus assignment of line (25) that leads to an invalid situation in line (26).

Our last series of examples, like in section 3, pertains to the conformance of *TUPLE* types with different number of parameters.

They rely on the code for the *WINDOW* class provided on page 127:

```
my_window.when_pointer_move( agent print_all )                -- (27)
my_window.when_pointer_move( agent print_x )                  -- (28)
my_window.when_pointer_move( agent io.put_string("move%N") )  -- (29)

print_x(x: INTEGER) is
   do
      io.put_integer(x) ; io.put_new_line
   end

print_all(x, y: INTEGER; t: TIME) is
   do
      io.put_integer(x) ; io.put_character(' ')
      io.put_integer(y) ; io.put_character(' ')
      io.put_string(t.to_string) ; io.put_new_line
   end
```

On line (27), the type of **agent** print_all is *PROCEDURE*[*T*, *TUPLE*[*INTEGER*, *INTEGER*, *TIME*]]. Since the argument type for when_pointer_move is *PROCEDURE*[*ANY*, *TUPLE*[*INTEGER*, *INTEGER*]] (see line (2) page 127), the indirect assignment on line (27) is not allowed anymore according to our new conformance rules. Indeed, OpenRule requires that *TUPLE*[*INTEGER*, *INTEGER*]] be conform to *TUPLE*[*INTEGER*, *INTEGER*, *TIME*]], which is not the case according to the conformance rules between *TUPLE* types. So this line is now statically rejected, which prevents reaching the problem previously explained on page 136.

On line (28), the type of **agent** print_x is *PROCEDURE*[*T*, *TUPLE*[*INTEGER*]], and the argument type for when_pointer_move is *PROCEDURE*[*ANY*, *TUPLE*[*INTEGER*, *INTEGER*]] (line (2) page 127). The indirect assignment on line (28) is now valid according to our new conformance rules that require *TUPLE*[*INTEGER*, *INTEGER*]] to conform to *TUPLE*[*INTEGER*], which is the case. When executed, the print_x method just ignores the extra available data. This gives additional expressiveness, compared to the normal rules, *with total safety*.

Line (29) is a case similar to that of line (28). This call is now valid as well, and safe. The executed method simply ignores all data, since it does not need any.

This capability to ignore some arguments might seem dangerous if it were allowed with immediate calls, but we think it is quite useful in the agent case for at least three reasons.

First, when working with a graphical system, it is easy to trace events as shown in our examples by just printing a message and ignoring all other data provided with this event. This is a very simple but *convenient way to debug* event-based systems.

Second, some data may be irrelevant for the action to execute. As an example, the method to execute when the user clicks on some button is in most cases inde-

pendent from the mouse pointer coordinates when the click is performed. *Our new rules allow to reuse existing methods* not specific to such an event coming from the graphical system and which did not care about the mouse pointer. We think that such cases are common in practice.

Finally, the ability to take into account only some of the arguments *helps software evolutions*. For example, a graphical system may evolve by providing more informations with the 'button clicked' event, say, by adding a time-stamp and a keyboard status. These extra pieces of information shall simply be ignored by any existing code, instead of breaking it all.

Conclusion

All these examples, that revealed problems with the normal conformance rules for agents in section 3, now work as expected with our new conformance rules.

Code that was needlessly rejected with the normal rules is now accepted, thus giving extra expressiveness to agents and their users, while maintaining security both at compile time (thanks to the type system) and at execution time.

This is an nice gain, but not the main one. Indeed, much more important is the fact that our new rules catch, at compile time, fatal errors that were completely undetected with the normal conformance rules. Hence erroneous code that was accepted, compiled and crashed at execution is now rejected. Our rules are thus safe, while the old ones were not.

5 CONCLUSION

In this paper, we showed that using generic types conformance rules with *ROUTINE* types was flawed. We provided examples to demonstrate that these rules could lead acceptable and safe code to be unnecessarily rejected, while erroneous code that had no way to work properly would be accepted.

To solve these issues, we added new specific conformance rules that allow agent assignment to become safe and accept more valid code as well. These specific rules do not change the typing of agent *per se*, but simply conformance on the second generic argument (relative to open arguments).

At first sight, these rules may seem a bit unintuitive to Eiffel developers. But they provide not only safety, but also extra expressiveness in a very natural and useable way, thus making it possible to develop with agents safely and easily.

To make the agents conformance rule easy for the user, he has to consider that if some method needs argument whose type is *ROUTINE*[*BASE*, *OPEN*], then the agent he will give as parameter is sure to be called with arguments conforming to *OPEN*, and then he needs to give an agent able to handle such arguments.

To summarize, the main arguments to adopt these specific conformance rules are:

- agents are not to be a way to defeat the typing system,
- conformance rules are maintained when the routine is executed,
- newly valid cases are useful and will be executed easily and safely,
- newly forbidden cases must be so, since they are error cases,
- if we consider agents as delayed calls, our new rules make it possible to delay any call that is valid as an immediate call; valid delayed calls using all parameters are also valid as immediate calls.

Thanks to the *TUPLE* type conformance rule and new *ROUTINE* conformance rule, delayed calls have one more capability than immediate calls: they may ignore some arguments. As explained, this property helps event debugging, software reuse and software evolutivity.

REFERENCES

[DHM+99] Paul Dubois, Mark Howard, Bertrand Meyer, Michael Schweitzer, and Emmanuel Stapf. From calls to agents. *Journal of Object-Oriented Programming (JOOP)*, 12(6), June 1999.

[Mey92] Bertrand Meyer. *Eiffel, The Language*. Prentice Hall, Englewood Cliffs, 1992. ISBN 0-13-247925-7.

[Mey00] Bertrand Meyer. Agents, iteration and introspection. Chapter 25 of ongoing work for the new *Eiffel, The Language* manual, May 2000. http://archive.eiffel.com/doc/manuals/language/agent/agent.pdf.

ABOUT THE AUTHORS

Philippe Ribet is software engineer in Toulouse, France. He works on graphic library design, using Eiffel's language power and works on SmartEiffel project. He can be reached at p.ribet@worldonline.fr.

Cyril Adrian is software engineer in Montbéliard, France. He joined the SmartEiffel team in summer 2002, working on the project in his free time. He developed a new installer, added the Acyclic Visitor design pattern, and worked on the first implementation of SCOOP. He can be reached at cyril.adrian@laposte.net. See also http://www.chez.com/cadrian/.

Olivier Zendra is a Researcher at INRIA-Lorraine / LORIA in Nancy, France. He works on the definition, compilation and optimization of object-oriented languages and on the SmartEiffel project. He can be reached at Olivier.Zendra@loria.fr. See also http://www.loria.fr/~zendra.

Dominique Colnet is Professor at the University of Nancy2. He is the original author of the GNU Eiffel compiler and the leader of the SmartEiffel team. He can be reached at Dominique.Colnet@loria.fr. See also http://www.loria.fr/~colnet.