Formalising Eiffel References and Expanded Types in PVS

Richard Paige[1], Jonathan Ostroff[2], and Phillip Brooke[3]

[1] Department of Computer Science, University of York, UK. paige@cs.york.ac.uk
[2] Department of Computer Science, York University, Canada. jonathan@cs.yorku.ca
[3] School of Computing, University of Plymouth, UK. philb@soc.plym.ac.uk

Abstract. Ongoing work is described in which a theory of Eiffel reference and expanded (composite) types is formalised. The theory is expressed in the PVS specification language, thus enabling the use of the PVS theorem prover and model checker to semi-automatically prove properties about Eiffel structures and programs. The theory is being used as the basis for automated support for the Eiffel Refinement Calculus.

1 Introduction

There is a definite need to be able to formally reason about *models* of object-oriented (OO) systems, e.g., as written in UML, in order to be able to analyse requirements and verify and validate design alternatives. Such techniques can let us discover errors, omissions, and ambiguities early in the system development process. Equally, it is vitally important to be able to reason formally about object-oriented *programs*, written in industrial-strength OO programming languages such as Java, C++, Eiffel, and Smalltalk. This is challenging for a number of reasons:

- Industrial-strength OO languages are typically very large, with numerous features that are not easily formalised, and for which formal reasoning can be difficult, such as pointers, deep loop exits, dynamic dispatch, and aggregate types.
- Industrial-strength OO languages are invariably complex, with sophisticated type systems that enable efficient execution and programming, but which are not necessarily designed for formal reasoning.
- Industrial-strength OO programs are usually very large, thus presenting a need for monotonic and compositional reasoning.

Despite these obvious difficulties, the need for reasoning about industrial-strength OO programs and programming languages remains: if engineers are to adopt reasoning techniques and tools, these mechanisms must support the languages and techniques that are applied in practice.

There is ongoing work on supporting reasoning about OO programs. Some of this focuses on defining new precise OO specification languages that can be translated into more widely used programming languages such as Java and C++. Perfect Developer [Es00] is an example of this, supporting formal OO specification, theorem proving, and code generation; it requires learning a new specification language, but thereafter

can generate code in a number of widely used languages, such as C++. The work of Cavalcanti et al [CN00] has produced a wp semantics for a subset of typical OO programming features; this semantics can thereafter be used to define a refinement calculus for transforming OO specifications into programs.

Other work has tackled the complexities of reasoning about OO programming languages head on. The LOOP project [BJ01] has focused on producing theories for the PVS system [CO95] that allow reasoning about Java programs; this should be contrasted with our work which is intended to support reasoning about specifications and transforming specifications into executable programs. The Extended Static Checker [LN00] provides a lightweight approach to verifying Java programs via source code annotation and a `lint`-like tool interface that detects many typical OO programming bugs at compile time, e.g., null reference access.

This paper reports on ongoing work on formalising a theory of reference and expanded types in the Eiffel programming language [Mey92]. A set of axioms expressing this theory was proposed in [PO03]. Meyer also has ongoing work on formalising parts of the Eiffel language using Hoare triples [Mey03]. This paper extends [PO03] by showing how the axioms can be formalised in the PVS language; the PVS theorem prover, model checker, and ground evaluator can thereafter be used to semi-automatically reason about and simulate specifications of Eiffel programs. Immediate applications of the PVS theory would be to support reasoning about aliasing in Eiffel, and about particularly complicated interactions between Eiffel reference types and expanded types. The PVS theories produced form the basis for automated support for the Eiffel Refinement Calculus [PO03].

We commence with a brief introduction to Eiffel and PVS, and attempt to provide some justification for combining the use of these two technologies in the calculus.

1.1 Eiffel

Eiffel is an object-oriented programming language and method [Mey97]; it provides constructs typical of the object-oriented paradigm, including classes, objects, inheritance, associations, composite ("expanded") types, generic types, polymorphism and dynamic binding, and automatic memory management.

However, Eiffel is not just a programming language — it also includes the notion of a *contract* to specify the duties of clients and suppliers. A valid Eiffel program may consist only of contracts – that is, it may possess no program code whatsoever – or it may be a combination of contract and code. or code only. It is in part because of its support for contracts that we have chosen Eiffel as the target language for the refinement calculus referenced in [PO03], and as the specification language to be partially formalised in PVS.

A short example of an Eiffel class is shown in Fig. 1. The class $CITIZEN$ inherits from $PERSON$ (thus defining a subtyping relationship). It provides several attributes, e.g., $spouse, children$ which are of reference type (in other words, $spouse$ refers to an object of type $CITIZEN$); these features are publicly accessible (i.e., are exported to ANY client). Attributes are by default of reference type; a reference attribute either points at an object on the heap, or is $Void$. The class provides one expanded attribute, $blood\_type$. Expanded attributes are also known as composite attributes; they are not

references, and memory is allocated for expanded attributes when memory is allocated for the enclosing object.

The remaining features of the class are routines, i.e., functions (like $single$, which returns $true$ iff the citizen has no spouse) and procedures (like $divorce$, which changes the state of the object). These routines may have preconditions (**require** clauses) and postconditions (**ensure** clauses), but no implementations. Routines may also have a **modifies** clause, which specifies those entities that may be changed by the routine – i.e., the *frame* of the routine. The purpose of the Eiffel Refinement Calculus is to transform such contracts into immediately executable Eiffel code, with a guarantee that the code is consistent with the contract. Finally, the class has an invariant, specifying properties that must be true of all objects of the class at stable points in time, i.e., before any valid client call on the object. While we have used predicate logic in specifying the invariant of $CITIZEN$, it should be observed that Eiffel does not support this exact syntax. It does possess a notion of *agent* that can be used to simulate quantifiers like the ones used in the example.

class $CITIZEN$ **inherit** $PERSON$
feature $\{ANY\}$
    $spouse : CITIZEN$
    $children, parents : SET[CITIZEN]$
    $blood\_type$: **expanded** $BLOOD\_TYPE$
    $single : BOOLEAN$ **is**
        ensure $Result = (spouse = Void)$
feature $\{BIG\_GOVERNMENT\}$
    $marry \cdots$
    $have\_child \cdots$
    $divorce$ **is**
        modifies $single, spouse$
        require $\neg\ single$
        ensure $single \wedge (\mathbf{old}\ spouse).single$
invariant
    $single \vee spouse.spouse = Current$;
    $parents.count \leq 2$;
    $\forall\, c \in children \bullet \exists\, p \in c.parents \bullet p = Current$
end

Fig. 1. Eiffel Class Interface

Other facilities offered by Eiffel, but not demonstrated here, include generic (parameterised) types, dynamic dispatch, multiple inheritance, static typing, and agents (function objects). We refer the reader to [Mey92] for full details.

1.1.1 Reference and Expanded Types in Eiffel Eiffel is a novel language in that it supports both user-defined reference variables and *expanded* variables. A reference variable may be attached to an object at run-time; thus, it is effectively a (safe) pointer

to a memory location. The pointer is "safe" in the sense that its address cannot be taken (easily), and pointer arithmetic cannot be applied to it. To use a reference variable requires two steps: declaration of the variable, and allocation/attachment of an object to the variable. This mechanism is somewhat cumbersome and inefficient for basic datatypes, such as integers. Thus, a shortcut is provided wherein a declaration of a variable i of type $INTEGER$ associates i directly with a storage location. This is roughly equivalent to stack allocation of memory in languages such as C and Pascal. Basic types, such as integers, are by default allocated in this way, but Eiffel provides a generalised mechanism by which any variable can be directly associated with a storage location; such variables are declared to be of expanded type. This allows programmers to avoid using reference mechanisms where they deem it appropriate; this should be contrasted with Java, which does not support a similar user-defined notion of expanded type. Interesting compatibility issues arise when using reference and expanded variables together in expressions and, particularly, assignment statements.

1.2 PVS

The PVS system [CO95] combines an expressive specification language with an interactive theorem prover and proof checker. The specification language is founded on classical typed higher-order logic with typical base types `bool`, `nat`, `int`, `real`, etc. It also provides function constructors of the form `[A->B]`. The type system of PVS is augmented with dependent types, abstract data types, and predicate subtypes, the last of which is a distinguishing characteristic of the specification language. The subtype `{x:A|P(x)}` consists of those elements of type `A` that satisfy the predicate `P`. The presence of predicate subtypes means that type checking of PVS specifications is undecidable in general, and thus requires use of the PVS theorem prover. Thus, the type checker generates proof obligations (called *type correctness conditions (TCCs)* in those cases where type conflicts cannot be resolved automatically. Frequently, large numbers of TCCs are discharged automatically by special stategies.

The PVS system has been used successfully in industrial projects, particularly for protocol and microprocessor verification. It is widely considered to be one of the most powerful theorem provers in use today. It is because of the power of the theorem prover, the expressiveness of its specification language, and the wealth of PVS libraries and expertise available in the research community, that we have targeted it as the means for providing automated support for the refinement calculus.

2 Overview of the Theory of Eiffel Reference Types

The paper [PO03] proposes a refinement calculus for Eiffel, which allows specifications – written in a pre- and postcondition style, using Eiffel's built-in support for such facilities – to be refined directly to Eiffel programs. The calculus is built atop Hehner's predicative programming theory [Heh03]. The calculus was produced by formalising the precisely stated semantics of Eiffel statements given in [Mey92] using Hoare logic. At the basis of this formalisation is a theory of Eiffel reference types, based on so-called

entity groups, which are the equivalence classes induced by the reference equality operator in Eiffel. This theory is used in formalising assignment statements, object **create** statements, and method calls. The theory, and its axioms, are based on careful analysis of the partial formalisation of Eiffel's semantics given in [Mey92].

We summarise elements of the theory here. Eiffel programs are made up of a number of classes, each of which possess *routines*; a routine is either a query (which returns a result without side effects) or a command (which changes the state of an invoking object). Routines may declare entities (variables) as local variables; entities may also be introduced as attributes of a class. Classes themselves are connected via associations and inheritance relationships.

Given routine r of class C, we let $r.\rho$ denote the set of reference entities that could appear in the routine body (including attributes of C, arguments and local variables of r, and syntactically legal dots and multi-dots expressions). If routine r is a query then $Result \in r.\rho$ because $Result$ is a predefined local variable of the query. Each entity in $r.\rho$ is potentially the subject of a creation instruction either directly in the body of r itself, or in a routine called by r.

Associated with each entity $e \in r.\rho$ there is a corresponding *entity group* which we denote by $\underline{e}$. Before the first creation statement in the body of r, the group is an empty set. After a $\mathbf{create}\, e$ instruction, $\underline{e} = \{e\}$, and this entity group is used to keep track of all reference entities in $r.\rho$ that point to the same object as e. We let $r.\pi$ denote all entity groups of routine r, and we require that these groups be disjoint (this equivalence relation will be formalized in the sequel).

The boolean expression $equal(e1, e2)$ is used for object equality. It can be defined recursively, provided that its occurrence in a program is syntactically legal; the formalisation is in [PO03]. An axiom is also needed that asserts that reference equality is at least as strong as object equality:

$$e1 \stackrel{r}{=} e2 \rightarrow equal(e1, e2) \tag{1}$$

The remaining axioms defining entity groups are as follows. Consider a routine r with bunch of entity groups $r.\pi$:

$$\forall\, \underline{ei}, \underline{ej} \in r.\pi \;\bullet\; \underline{ei} = \underline{ej} \;\vee\; \underline{ei} \cap \underline{ej} = \varnothing \tag{2}$$

Axiom (2) states that two entity groups are either disjoint or identical. Only reference entities are subject to aliasing, and hence only reference entities have associated groups. Thus if $e1$ and $e2$ point to the same object, then they are in the same group.

$$\forall\, \underline{e} \in r.\pi \bullet (\forall\, e1, e2 \in \underline{e} \bullet e1 \stackrel{r}{=} e2) \tag{3}$$

Axiom (3) states that if two entities are in the same entity group, they refer to the same object.

$$\forall\, \underline{e} \in r.\pi \bullet \forall\, e1, e2 \in \underline{e} \bullet \underline{e1} = \underline{e2} \tag{4}$$

(4) asserts that if two entities are in the same group, then the group can be referred to in expressions by either name.

$$\forall\, \underline{e} \in r.\pi \bullet \forall\, e \in \underline{e} \bullet e \neq \mathit{Void} \tag{5}$$

$$\forall\, e \in r.\rho \bullet (\forall\, \underline{e}' \in r.\pi \bullet e \notin \underline{e}') \equiv e = \mathit{Void} \tag{6}$$

$$\forall\, e \in r.\rho \bullet (e = \mathit{Void}) \equiv (\underline{e} = \varnothing) \tag{7}$$

Axiom (5) states that any entity in an group is attached to an object, and thus cannot be Void. Axiom (6) states that if an entity e is in no group, then it is Void. This is the state an entity is in after declaration, or after an assignment $e := \mathit{Void}$. Axiom (7) equates a Void reference with an empty entity group. It is perhaps not clear why entity groups have been used to formalise reference types in Eiffel, as opposed to obvious approaches, e.g., formalising a memory model with memory locations, pointers, and objects. One issue with the latter approach is that it can lead to long expressions (typically stating what values are stored in what memory locations) appearing in refinement steps. We desire to make the refinement calculus useful for carrying out refinement steps by hand and with automated assistance. Long expressions do not arise with entity groups, as they effectively let developers express only those entities that are of interest in a refinement step. Certainly, when using PVS it would be reasonable to specify an Eiffel memory model, and to define entity groups in terms of that memory model. In formulating the Eiffel memory model, we would envision using an approach similar to that for the Extended Static Checker for Java [LN00]. We are considering this approach in revisions to our theories, but for now are simply formalising the axioms directly so as to make it easier to validate and check our PVS formulation of the Eiffel refinement calculus.

The notion of entity groups can be used to define the semantics of instructions that use reference types in Eiffel such as the **create** statement, assignment and feature call. The $\mathbf{create}\ e$ instruction creates a new object and attaches it to entity e; any previous reference or attachment via entity e is lost; however, any other entities that referred to the object originally attached to e remain. This continued attachment is established by the previous axioms, specifically (4) and (5). The semantics of $\mathbf{create}\ e$ is given in Definition 1.

Definition 1. *[Entity creation semantics]* Given a reference entity e, the instruction $\mathbf{create}\ e$ is defined as

$$\begin{array}{l} \mathbf{modifies}\ e, t \\ \mathbf{ensure}\ \underline{e} = \{e\} \wedge \mathit{default}(e) \end{array}$$

where t represents the global clock (it is used in refining specifications to loops or recursive programs). The $\mathit{default}(e)$ clause asserts that each attribute $e.a$ is set to its default value on creation as described in [Mey97] (e.g., if a is a $\mathit{BOOLEAN}$ it is set to false, if it's a reference it is set to Void etc.). If the class $e.\mathit{type}$ has a creation routine r (whose purpose is to establish the class invariant), then we must execute this routine after the creation statement, i.e., $\mathbf{create}\ e;\ e.r$.

The type-compatible reference assignment statement $e1 := e2$ changes entity $e1$ to refer to the same object as entity $e2$. Definition 2 provides the semantics for the reference assignment (i.e., assigning references to references), leaving assignments involving both references and expanded types for the sequel.

Definition 2. *[Reference assignment semantics]* Suppose $e1$ and $e2$ are references and their declared types are compatible according to [Mey97], so that $e2$ can be assigned to $e1$. Then $e1 := e2$ is defined as

$$\begin{aligned} &\textbf{modifies } e1, t \\ &\textbf{ensure } e1 = e2 \end{aligned}$$

We now present the meaning of procedure calls, to illustrate how the theory can be applied to feature calls; query calls are formalised in [PO03]. Definition 3 provides the meaning of a targeted command call $e1.c(e2)$, where $e1$ and $e2$ are reference entities and c is a command of class $SUPPLIER$. The meaning of this call is supplied by the precondition and postcondition of c, targeted to the entities $e1$ and $e2$ (in the definition, $\backslash$ applied to any expression refers to the value of the expression evaluated in the prestate).

Definition 3. *[Targeted command call]* The call $e1.c(e2)$ for command $c(x1 : TYPE)$, which changes entities contained in $c.modifies$, and which is in a class having attribute a, means

$$\begin{aligned} &\textbf{modifies } c.modifies[a := e1.a, x1.a := e2.a] \\ &\textbf{require } e1 \neq Void; \\ &\qquad c.pre[x1 := e2, a := e1.a, Current := e1] \\ &\textbf{ensure } c.post[\backslash a := \backslash e1.a, \backslash Current := \backslash e1, \\ &\qquad\qquad x1 := e2, a := e1.a, Current := e1]; \\ &\quad e2 = \backslash e2 \end{aligned}$$

Local variables can be introduced to deal with more complicated, and potentially multiple, arguments.

Such statements, e.g., assignment statements and command calls, can be introduced during refinement, and thus it is possible to check that routine implementations are consistent with the pre- and postconditions specified in class interfaces.

2.1 Expanded types

We have not yet discussed the effect of assigning an expanded object to a reference, and vice versa. The interplay between expanded (composite) types and reference types is somewhat complicated by the fact that Eiffel allows user-defined expanded types, and thus the semantics of the assignment statement needs to be generalised in order to allow interactions between the two kinds of variables. The table in Fig. 2 suggests

$e1$ expanded and $e2$ reference $e1 := e2$ equivalent to $e1.copy(e2)$	$e1$ reference and $e2$ expanded $e1 := e2$ equivalent to $e1 := clone(e2)$
modifies $t, e1$ **require** $e2 \neq Void$ **ensure** $equal(e1, e2)$	**modifies** $t, e1$ **require** $true$ **ensure** $\underline{e1} = \{e1\} \wedge equal(e1, e2)$

Fig. 2. Hybrid assignments

how to extend our approach to handle them, leaving a full treatment (e.g., expanded parameters) for later work.

Thus, full formalisation of expanded types will require formalisation of Eiffel's $copy$ and $clone$ statements. This can be carried out using the notion of entity groups described previously.

3 PVS Formulation

The axiomatisation of Eiffel reference types presented in the previous section is sufficient and useful for manual reasoning; examples in [PO03] demonstrate its efficacy. We are currently formalising the axioms and definitions in the PVS specification language, so that we can use particularly the PVS prover to reason about references automatically. In this section, we briefly outline progress that has been made on the formalisation.

The PVS theory includes four main sections:

- declarations of basic types for entities, entity groups, and primitive types;
- declaration of primitive functions for entity comparison, identifying different kinds of entities (i.e., basic types versus reference types), as well as conversions of Eiffel basic types into PVS types;
- the axioms from Section 2;
- Definitions of **create**, reference assignment, feature calls, and hybrid assignments involving expanded and reference types.

To use the theory, programmers `import` it in a theory that defines PVS translations of programmer-defined classes. The PVS translations define new types, functions, and axioms that constrain the programmer classes. This is discussed further in the sequel.

3.1 Declarations of basic types and functions

Fundamental types must be declared in PVS for Eiffel entities (i.e., variables), entity groups, and entity groups associated with routines.

```
ENTITY: TYPE+
ENTITY_GROUP: TYPE+ = set[ENTITY]
SET_GROUP: TYPE+ = set[ENTITY_GROUP]
ROUTINE: TYPE+
QUERY, COMMAND: TYPE+ FROM ROUTINE
```

Each new class introduced in an Eiffel program (not including basic/primitive types) will introduce a new PVS type holding attributes. These PVS types all are subtypes of pre-declared PVS type `OBJECT`. This type is declared primarily for generality: to be able to define functions that apply to all classes. Note that basic/primitive Eiffel types are subtypes of `OBJECT`.

```
OBJECT: TYPE+
EIFFEL_TYPE: TYPE+ FROM OBJECT
EIF_INT, EIF_REAL, EIF_CHAR: TYPE+ FROM EIFFEL_TYPE
```

Useful functions can now be declared. In particular, we will need to know whether an entity is attached to an object, whether two entities have the same declared/static type, whether two entities are reference (or object) equal, etc. Finally, we need to define functions to implement the entity groups from the previous section, i.e., $r.\rho$ and $r.\pi$.

```
isvoid: [ ENTITY -> bool ]
deref: [ ENTITY -> OBJECT ]
sametype: [ ENTITY, ENTITY -> bool ]
ref_equal: [ ENTITY, ENTITY -> bool ]
obj_equal: [ ENTITY, ENTITY -> bool ]
routine_entities: [ ROUTINE -> set[ENTITY] ]
pi: [ ROUTINE -> SET_GROUP ]
ep_from_entity: [ ENTITY -> ENTITY_GROUP ]
pre: [ ROUTINE, ENTITY, ENTITY -> bool ]
post: [ ROUTINE, ENTITY, ENTITY, ENTITY -> bool ]
frame: [ ROUTINE -> bool ]
```

We will also need to provide conversions from Eiffel built-in types, such as integers and characters, to their PVS equivalents. In this sense, we need to implement a PVS embedding of Eiffel primitives. Since PVS provides a conversion facility, this is relatively straightforward.

```
int_c: [ EIF_INT -> int ]
real_c : [ EIF_REAL -> real ]
char_c : [ EIF_CHAR -> char ]

CONVERSION int_c, real_c, char_c
```

3.2 PVS specification of entity group axioms

The axioms from Section 2 can now be expressed in PVS, based on the functions and basic types previously declared. We present several axioms to illustrate the process. First, the axiom that states that reference equality is stronger than object equality. This is a direct transliteration of the ERC axiom.

```
ax2: AXIOM
(FORALL (em, en: ENTITY):
  ref_equal(em,en) IMPLIES obj_equal(em,en))
```

More complex is the axiom that states that entity groups associated with routines are either equal or they do not intersect.

```
ax3: AXIOM
(FORALL (r:ROUTINE):
  (FORALL (ei,ej:ENTITY_GROUP):
    member(ei, pi(r)) AND member(ej, pi(r)) IMPLIES
      ((ei=ej) OR (empty?(intersection(ei,ej))) ) ))
```

Two final examples demonstrate PVS specifications of: the axiom that states that entities in the same entity group refer to the same entity; and, an entity in an entity group must be non-$Void$.

```
ax5: AXIOM
(FORALL (r:ROUTINE): (FORALL (ei:ENTITY_GROUP):
  (FORALL (em,en: ENTITY):
  (member(ei,pi(r)) AND member(em,ei) AND member(en,ei)) IMPLIES
  ep_from_entity(em)=ep_from_entity(en) )))

ax6: AXIOM
(FORALL (r:ROUTINE): (FORALL (em:ENTITY):
  (member(ei,pi(r)) AND member(em,ei)) IMPLIES NOT isvoid(em) )))
```

Based on these axioms, and the functions declared in the previous subsection, reference and object equality can be defined as well.

3.3 PVS definitions of operations

We can now specify some of the fundamental Eiffel instructions in PVS, particularly those that manipulate reference types. As two examples, we show how to specify Eiffel's **create** statement, and reference assignment.

Eiffel's creation and initialisation statement has the form **create** e, where e is an entity that may or may not be attached to an object. It is formulated in PVS as follows.

```
create: [ ENTITY -> bool ] =
(LAMBDA (em: ENTITY): ep_from_entity(em)= singleton(em) AND default(em))
```

`default` is a function that, given an entity, returns $true$ iff the entity has been initialised to its default value. The default value depends on the type of the object attached to the entity (but typically it is made up of default values for the attributes of the object - e.g., 0 for integers, $true$ for booleans, $Void$ for references).

Reference assignment of the form $e := e1$ is formalised as follows. `ref_assign` returns $true$ iff the two argument entities are reference equal. In other words, $e := e1$ is represented in PVS as the function call $ref\_equal(em, en)$ - we are representing the *effect* of the assignment in PVS.

```
ref_assign: [ENTITY, ENTITY -> bool ] = (LAMBDA (em,en:ENTITY): ref_equal(em,en))
```

3.4 Feature calls

The fundamental construct in any object-oriented program is the targetted feature call, which has the form $o.f(a)$, where o is an entity/variable – the target – that is attached to an object, f is a function or procedure, and a is a set of arguments. Targetted procedure calls are statements, and can thus be sequentially composed with other typically programming constructs, such as assignments, loops, and selections. Targetted function calls return values, and as such can be used as r-values in assignment statements, in guards, as arguments, or in pre- and postconditions. We show how to formalise Eiffel targetted function and procedure calls in PVS, starting with the function call $e2.q(e3)$. This call might appear in an assertion, and as such it is important to formalise it separately from any statement in which it can be used. Its PVS formulation is as follows. To start with, we declare a PVS function `eval` which returns an entity when applied to a function, target, and set of arguments; the return type should be constrained to be that of the Eiffel function. This PVS function is then *implicitly* defined in the axiom `eval_ax`, which defines what it means to call q with target $e2$ and argument $e3$. The axiom assumes that functions to obtain the precondition and postcondition of a routine are available, respectively, as `pre` and `post`.

```
eval:[ ROUTINE, ENTITY, ENTITY -> ENTITY ]

eval_ax: AXIOM
(FORALL (em,en:ENTITY): FORALL (r:ROUTINE):
  (pre(r,em,en) AND NOT isvoid(em) IMPLIES post(r,em,en,eval(em,en,r))))
```

To use the above axiom, one makes use of the function `eval`. For example, to define the meaning of the assigned query call $e1 := e2.q(e3)$, one would make use of the following PVS function.

```
acq:[QUERY, ENTITY, ENTITY, ENTITY -> bool ] =
(LAMBDA (q:QUERY,e1:ENTITY,e2:ENTITY,e3:ENTITY):
  (NOT isvoid(e2) and reference(e1) AND
   reference(e2) and compatible(e1,result_type(q))) IMPLIES e1 = eval(e2,e3,q))
```

Finally, we formalise the targetted procedure call $e1.c(e2)$ in PVS. We first assume that the frame of each procedure can and has been specified by the programmer, and is available via the function `frame`. The call can then be formalised as follows.

```
tcq:[ COMMAND, ENTITY, ENTITY, ENTITY, ENTITY -> bool ] =
(LAMBDA (c:COMMAND,e1:ENTITY, e2:ENTITY,
         old_e1:ENTITY, old_e2:ENTITY):
  (NOT isvoid(e1) AND pre(c,old_e1,e2)) IMPLIES
  (post(c,e1,e2,old_e1) AND e2=old_e2 AND frame(c))))
```

Notice that the call introduces both old and new variables, i.e., pre- and poststate.

3.5 Expanded types

Based on the axioms and functions defined previously, it is now possible to formalise hybrid assignment statements, involving both expanded and reference types. For example, consider the assignment statement $e1 := e2$, where $e1$ is an expanded type and $e2$ is a reference type. In Eiffel, the meaning of this statement is identical to the instruction $e1.copy(e2)$. In other words, an attribute-by-attribute copy is made of the object attached to $e2$, stored in $e1$. We can thus formalise this instruction by partly formalising the Eiffel feature $copy$, as follows[1].

```
copy: [ ENTITY, ENTITY -> bool ] =
(LAMBDA (e1, e2: ENTITY):
  (reference(e2) AND NOT reference(e1) AND
   NOT isvoid(e2)) IMPLIES (obj_equal(e1,deref(e2)))
```

3.6 Using the theory

The above suite of functions, declarations, definitions, and axioms is part of the PVS theory of Eiffel reference types. It is intended to be used in PVS translations of Eiffel specifications[2]. Thus, an Eiffel program will be translated into a set of PVS theories – one per class – where each theory imports the above PVS theory of reference types. Each Eiffel class is translated into a PVS record, declaring the class's attributes and types. For example, a class C with attributes $x : INTEGER$ and $c : C$ (i.e., one expanded attribute and one reference) will be mapped to the PVS type

```
C: TYPE+ FROM OBJECT =
  [# x: EIF_INT, c: ENTITY #]

C_ax: AXIOM
(FORALL (varc:C): EXISTS (oc:C):
  NOT isvoid(c(varc)) IMPLIES deref(c(varc)) = oc )
```

The axiom states that whenever c is attached to an object, it must be attached to an object of type C.

Translations of routines (as PVS functions) and their contracts can be carried out in much the same way, though details remain to be worked out. We have concentrated so far on fixing the details of the theory of reference types in PVS, since the formalisation of classes and contracts depends on it.

A challenge with using the theory will be in proving that sequential compositions of routine calls that appear in method bodies satisfy a contract. This is challenging because intermediate state needs to be introduced. However, the formalisation of sequential composition in [Heh03] deals with this problem by treating sequential composition as relational composition, thus hiding the intermediate state using an existential quantifier. We expect that we can use this approach in PVS.

[1] $copy$ is not fully formalised on its domain; it is defined only on domains where the l-value is expanded and the r-value reference.

[2] An Eiffel specification includes contracts, but not implementations, of classes – i.e., no routine bodies.

4 Discussion and Conclusions

The PVS theory defined above is accepted by the PVS system, and typechecks. Work is continuing on using the theory to carry out examples of reasoning, particularly for refinement. Current work is focusing on completing the aliasing example in [PO03] in full detail in PVS. So far, use of PVS has helped us detect omissions – particularly in terms of the types used in expressions – in our semi-formal axiomatisation of entity groups: PVS forces us to be explicit about types, and this helped to reveal errors.

Additional work is considering alternative PVS formalisations of the refinement calculus (particularly, using an explicit model of Eiffel memory, with entity groups being a derived notion). We are also automating the generation of PVS theories from Eiffel. A basic PVS theory, capturing entity groups, will be imported by any generated theory; the generated theory will define records to capture the attributes and pre- and postconditions associated with routines of classes.

References

[BJ01] J. van den Berg and B. Jacobs.: The LOOP compiler for Java and JML. In *Proc. TACAS 2001*, Lecture Notes in Computer Science 2031, Springer-Verlag, 2001.

[CN00] A. Cavalcanti and D. Naumann.: A weakest-precondition semantics for refinement object-oriented programs. *IEEE Trans. Software Engineering* **26**(8), 2000.

[CO95] J. Crow, S. Owre, J. Rushby, N. Shankar, and M. Srivas.: A Tutorial Introduction to PVS, in *Proc. WIFT '95*, Springer-Verlag, 1995.

[DL98] D.L. Detlefs, K.R.M. Leino, G. Nelson, and J.B. Saxe.: Extended Static Checking. SRC Research Report 159, December 1998.

[DL01] K. Dhara and G. Leavens.: Mutation, Aliasing, Viewpoints, Modular Reasoning, and Weak Behavioral Subtyping. Technical Report #01-02, Department of Computer Science, Iowa State University, March 2001.

[Es00] Escher Technologies, Inc.: Getting Started With Perfect, available from `www.eschertech.com`, 2000.

[Heh03] E.C.R. Hehner.: *A Practical Theory of Programming (Second Edition)*, Springer-Verlag, 2003.

[LN00] K.R.M. Leino, G. Nelson, and J.B. Saxe.: ESC/Java User's Manual. Technical Note 2000-002, Compaq Systems Research Center, October 2000.

[Mey92] B. Meyer.: *Eiffel: the Language*, Prentice-Hall, 1992.

[Mey97] B. Meyer.: *Object-Oriented Software Construction* (Second Edition), Prentice-Hall, 1997.

[Mey00] B. Meyer.: Agents, iterators, and introspection. ISE Inc. Technical Paper, 2000.

[Mey03] B. Meyer.: Proving program pointer properties, draft last revised February 2003.

[PO03] R. Paige and J. Ostroff.: *ERC: an Object-Oriented Refinement Calculus for Eiffel*, under review, 2003. Draft available at www.cs.yorku.ca/techreports/2001.