A Seamless Eiffel-Based Refinement Calculus for Object-Oriented Systems (Extended Abstract)

Richard Paige[1] and Jonathan Ostroff[2]

[1] Department of Computer Science, University of York,
Heslington, York YO10 5DD, United Kingdom. paige@cs.york.ac.uk
[2] Department of Computer Science, York University,
4700 Keele Street, Toronto, Ontario M3J 1P3, Canada. jonathan@cs.yorku.ca

Abstract. The Eiffel language [4] can be used throughout the object-oriented software development process, for both specification and implementation. We outline work-in-progress on developing a refinement calculus for producing provably correct Eiffel programs from Eiffel specifications. We propose extensions to the calculus by integrating abstract data types as a front-end language that can be used for avoiding implementation bias.

1 Introduction and Motivation

A critical element object-oriented (OO) technology is that it can be applied throughout much of a typical software development process. Classes and objects, the key concepts in OO development, can be applied in modelling late requirements (where customer goals, beliefs, and desires have been clearly determined), software architectures, in detailed specification and design, in implementation, and for software testing. The generality of OO techniques and their application throughout development can reduce the chance of introducing impedance mismatches, can simplify tracing of errors, and can improve maintainability.

When constructing OO software with correctness and robustness requirements, it is often desirable to develop in a *seamless* [4, 8] fashion. With seamless development, a core set of modelling constructs are applied throughout all stages of development, and thus a model used for requirements specification can be directly mapped to a model used for design or implementation. In the process, the model will likely have to be *refined* to provide details that are missing.

In this extended abstract, we outline work-in-progress on developing a refinement calculus for OO systems, based on the Eiffel language [4]. The calculus aims at producing more reliable – that is, correct and robust – software systems that are immediately executable using existing tools. Eiffel can be used seamlessly for both specification (based on its support for *design-by-contract* [4]) and programming. We thus sketch a set of refinement rules and background theories so that provably correct Eiffel programs can be constructed from Eiffel specifications. We also propose how abstract data types - which are a more abstract approach than classes for specifying types - might be integrated into such a refinement process via translation.

2 Overview of Eiffel

Eiffel is an object-oriented programming language and method [4]; it supports classes, objects, inheritance and client-supplier relationships, generic types, dynamic binding, and automatic memory management. The language also includes the notion of a *contract*. Since contracts can be used to specify software, Eiffel can also be used as a notation for specification and design. The fundamental specification construct in Eiffel is the class. A class is both a module and a type. Fig. 1 contains a short example of an interface of class $CITIZEN$. Each feature section, introduced with the keyword **feature**, is followed by a selective export clause that specifies a list of accessor classes. Each feature is an attribute (state) a routine (computations). Routine implementations are omitted in Fig. 1.

Routines are either functions (ideally without side-effects) or procedures (which result in state changes). Routines may optionally have contracts, written in the Eiffel assertion language, as preconditions (**require** clauses), postconditions (**ensure** clauses) and class invariants. In postconditions, the keyword **old** can be used to refer to the value of an expression when the feature was called. The **modifies** clause of a routine is a frame indicating those entities[1] that may

[1] An entity is an identifier denoting a run-time value. It is an attribute, a local, or a formal argument to a routine.

```
class CITIZEN
feature {ANY}
    name, sex : STRING
    spouse : CITIZEN
    children, parents : SET[CITIZEN]
    single : BOOLEAN
        ensure Result = (spouse = Void)
    divorce
        modifies single, spouse
        require ¬ single
        ensure single ∧ (old spouse).single

invariant
    single_or_married: single ∨ spouse.spouse = Current;
    number_of_parents: parents.count ≤ 2;
    symmetry: ∀ c ∈ children • ∃ p ∈ c.parents • p = Current
end
```

Fig. 1. Class *CITIZEN*

be changed by the routine; it is adopted from [5]. The assertion language is enhanced by the fact that assertions may refer to any query (e.g., the postcondition of *divorce* refers to the queries *single* and *spouse*).

A class invariant is an assertion (conjoined terms are separated by semicolons) that must be *true* whenever an instance of the class is used by another object, i.e., whenever a client can call an accessible feature.

Routines may optionally have bodies, i.e., implementations. These are written using a standard set of imperative language constructs – e.g., assignments, sequential compositions, if-then-elses, loops, et cetera – and OO constructs such as object creation statements (**create** statements) and feature calls of the form $o.f$. The complete syntax is summarized in [3]. It is these language constructs that will be produced by the refinement calculus.

3 Fundamentals of the Refinement Calculus

In order to be able to refine Eiffel specifications (consisting of classes with preconditions, postconditions, and invariants) into programs, we need a theory of Eiffel programming. We call our theory the Eiffel Refinement Calculus (ERC). ERC is based on the predicative calculus of Hehner [2]. To Hehner's calculus, we add new machinery for introducing Eiffel's OO constructs, such as object creation and feature calls.

Program constructs in Eiffel always appear in the bodies of routines. A specification of a program construct (e.g., an assignment statement or feature call) should identify the set of computations that the construct can execute. Thus, suppose the routine r has a precondition $r.pre$ (with free variables in the set σ) identifying the prestates, and a postcondition $r.post$ (with free variables in sets $\textbf{old}\,\sigma$ and σ) indicating the computed poststates, then the specification of the routine $r.spec$ is defined as [2]

$$r.spec \;\widehat{=}\; \textbf{old}\,r.pre \rightarrow r.post \wedge t \geq \textbf{old}\,t \wedge t \neq \infty \qquad (1)$$

where t is a conceptual global clock representing time, and $t \in \sigma$. The specification $r.spec$ is true iff (a) the precondition is not satisfied (in which case any behaviour is permitted); and (b) if the precondition is satisfied then the routine's execution must terminate in finite time with the postcondition true. A program is a specification that has been implemented. We describe imperative programs in ERC as follows (OO constructs will be specified separately).

$$\begin{array}{rcl}
\textbf{skip} & \widehat{=} & e1 = \textbf{old}\,e1 \wedge e2 = \textbf{old}\,e2 \wedge \ldots \wedge en = \textbf{old}\,en \wedge time \qquad (2)\\
e1{:=}exp & \widehat{=} & \textbf{old}\,defined(exp) \rightarrow e1 = \textbf{old}\,exp \wedge e2 = \textbf{old}\,e2 \wedge \ldots \wedge time\\
\textbf{if}\ b\ \textbf{then}\ P\ \textbf{else}\ Q & \widehat{=} & (\textbf{old}\,b \rightarrow P) \wedge (\neg\,\textbf{old}\,b \rightarrow Q) \wedge time\\
P;\ Q & \widehat{=} & \exists\,\sigma' \bullet P[\sigma := \sigma'] \wedge Q[\textbf{old}\,\sigma := \sigma']\\
(\textbf{local}\ e : T;\ P) & \widehat{=} & (\exists\,e, \textbf{old}\,e : T \bullet\ P)
\end{array}$$

P and Q can themselves be specifications or programs; we can mix programs and specifications because they are both described by predicates.

It is tedious to write out the entities that do not change in a specification, so we will use the specification statement [5] for writing specifications, and will take advantage of its notation for frames. However, we will define the meaning of specification statements using timed predicates (rather than weakest preconditions). The notation

$$e : \llparenthesis\, S, D\, \rrparenthesis \;\; \widehat{=} \;\; \mathbf{old}\ S \;\rightarrow\; D \wedge \mathit{time} \wedge \mathit{same}(\sigma - e)$$

will be used (where same is a predicate indicating a set of variables that do not change value). As well, we sometimes find it useful to apply the Larch/JML [1] paragraph style of specification (illustrated in the sequel).

3.1 Refinement

A specification spec is refined by an implementation impl if all the observations represented by impl are also observations of spec. We write this as $\mathit{spec} \sqsubseteq \mathit{impl}$. Our treatment of specifications as predicates leads to a very simple definition of refinement:

$$\mathit{spec} \sqsubseteq \mathit{impl} \quad \widehat{=} \quad (\forall\, \mathbf{old}\, \sigma, \sigma \bullet \mathit{impl} \rightarrow \mathit{spec}) \tag{3}$$

3.2 Reference types

A critical element of any OO programming language is its support for reference types (pointers). Eiffel is no exception in supporting these types. ERC provides a simple background theory of reference types called *entity partitions*. For each entity e declared in a class or routine, a set $\underline{e}$ is defined which contains all entities that refer to the object that e initially points at. This set is changed by assignments and routine calls. A full axiomatization of entity partitions is provided in [7]. The axiomatization supports aliasing, reference and value equality, and allows **create** statements to be applied to an entity at any time.

3.3 Rules and feature calls

A full set of refinement rules for ERC is specified in [7]. This set includes rules for introducing loops and other program constructs, and for simplifying sequences of specifications. Rules can also be inferred from the predicate semantics of program constructs in (3) and the definition of refinement. In order to fully support OO programming, rules must also be provided for introducing *feature calls*, which are the basis of OO computing. As well, a rule must be provided for allocating new objects and assigning them to entities (i.e., a semantics must be provided for the **create** statement of Eiffel). We desire a set of rules that allow a specification to be refined by a targeted procedure call of the form $o.r(c)$, an assigned targeted function call $e1 := e2.q(c)$, or to create a new object and attach it to an entity, via **create** e. We provide two examples: a rule for introducing a **create** statement, and a rule for assigned targeted function calls. The rules are formulated by defining the meaning of such statements and calls as specification statements.

Definition 1 (Entity creation semantics). The instruction **create** e is defined as

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

where $\mathit{remove}(e) \;\; \widehat{=} \;\; \forall\, ei \in \mathbf{old}\ \underline{e} \mid ei \neq e \bullet \underline{ei} = \mathbf{old}\ \underline{ei} - \{e\}$
and where $\mathit{nochange}(e) \;\; \widehat{=} \;\; \forall\, ei \mid ei \neq e \wedge ei \notin \mathbf{old}\, \underline{e} \bullet \underline{ei} = \mathbf{old}\, \underline{ei}$. The $\mathit{default}(e)$ clause asserts that each $e.a$ is set to its default value on creation as described in [4] (e.g., if a is a $\mathit{BOOLEAN}$ it is set to false, if it's a reference it is set to Void etc.).

Definition 2 (Assigned Function Call). The function call

$e1 := e2.q(e3)$

for function q means

$$\begin{array}{l} \textbf{modifies}\ \underline{e1}, t \\ \textbf{require}\ e2 \neq \mathit{Void}; \\ \qquad q.\mathit{pre}[a := e2.a, \mathit{Current} := e2, x2 := e3] \\ \textbf{ensure}\ q.\mathit{post}[a := \textbf{old}\ e2.a, \mathit{Current} := \textbf{old}\ e2, x := e3, \mathit{Result} := e1]; \\ \qquad e1 = \textbf{old}\ e2.q(e3) \end{array}$$

With the rule for introducing an assigned function call, we are attempting to avoid replacing the targeted call $e2.q(e3)$ with a call to a function $q(e2, e3)$, since this requires us to work with constructs that are not OO (i.e., mathematical functions as opposed to feature calls on objects). Further rules (including ones for treating arbitrary arguments, and for using targeted function calls in assertions) appear in [7].

3.4 The development process

The refinement calculus outlined above can be used to refine a specification to a program as follows. Initially, a set of Eiffel classes is written, where at least one class has at least one routine possessing a contract, but no implementation; all such routines must be refined using the rules suggested above. Refinement starts at the designated *root* class of the system; this is a class from which execution of the eventual implementation must begin. All routines of the root are refined to implementations. In doing so, the refiner may make use of all other features in the system that are accessible to the root class, e.g., public features that are accessible via attributes or inheritance. After refining the root, all classes that are related to the root, i.e., via inheritance or attributes, are themselves refined. This process continues recursively until all classes are implemented in Eiffel.

The refinement process commences with an Eiffel specification, consisting of routines with contracts but no implementations. Sometimes, this style of specification forces design decisions – especially regarding data representation – to be made prematurely. Consider a STACK class. To specify the stack in Eiffel, while avoiding underspecification of critical stack properties, a data implementation, such as a $\mathit{SEQUENCE}$, must be selected. It may be undesirable to make such a decision early in the specification and refinement process. Thus, it is useful to be able to use an abstract notation, like abstract data types (ADTs) for specification, and to integrate them with the refinement process.

We are currently experimenting with different ways to use ADTs as a front-end specification language for ERC. One proposal for extending use of the calculus that appears promising is as follows.

1. Write ADTs or Eiffel specifications as needed for the desired classes in the system.
2. Transform the ADTs into *extended* Eiffel specifications that use **describe** clauses (discussed further in the sequel).
3. Refine the Eiffel specifications using ERC.

To illustrate, here is a partial ADT for a stack. The function signatures applicable to the stack are listed (we give two examples only), then axioms and preconditions, which are omitted,

$$\begin{array}{l} \textbf{TYPES}\ \mathit{STACK}[G] \\ \textbf{FUNCTIONS} \\ \quad \mathit{push} : \mathit{STACK}[G] \times G \to \mathit{STACK}[G] \\ \quad \mathit{new} : \mathit{STACK}[G] \\ \quad \ldots \\ \textbf{AXIOMS} \\ \forall x : G, s : \mathit{STACK}[G] \bullet \\ \quad \mathit{top}(\mathit{push}(s, x)) = x; \quad \mathit{pop}(\mathit{push}(s, x)) = s; \\ \quad \mathit{empty}(\mathit{new}); \quad \textbf{not}\ \mathit{empty}(\mathit{push}(s, x)) \\ \textbf{PRECONDITIONS}\ \ldots \end{array}$$

To partially transform this to an Eiffel specification, we use the following rules:

– each constructor (i.e., new) is mapped to an Eiffel creation procedure.
– each function (e.g., $empty$, top) that returns an argument different from $STACK$ is mapped to an Eiffel function.
– each function that takes a $STACK$ as an argument and returns a new $STACK$ is mapped to an Eiffel procedure.
– preconditions are mapped to routine preconditions in Eiffel.

What remains is translating ADT axioms to Eiffel. In general, automatic translation of the axioms to Eiffel postconditions and class invariants is not possible. Even with user intervention, Eiffel is not expressive enough to fully capture the axioms. For example, the two axioms defining $push$, pop, and top (i.e., the LIFO property of stacks) cannot be translated directly. The postcondition of pop, for example, cannot include sequential compositions of procedure calls, which will be necessary to directly express the ADT axioms. To specify the postcondition of top will require the addition of a data representation to the class.

We are investigating the addition of a new clause, **describe**, to Eiffel specifications. With this, we can potentially avoid adding a data representation and can make it easier to translate ADTs to classes. **describe** is like an invariant in that it applies to all instances of a class. However, **old** may be used in **describe** clauses, unlike invariants. The **describe** clause defines properties that must always be $true$, except possibly during the execution of a routine body. Thus, each constraint in the clause is to be checked before and after each routine call (except before the call of any creation procedures). Invariants must be true only before and after client calls.

Here is an example of part of a **describe** clause for $STACK$. Each axiom of the ADT is mapped to a constraint in the **describe** clause. The first example is for the axiom stating that pushing then popping an element from a stack s produces s again. The second states that $push$ followed by top gives the pushed element. Note that reference equality is used in the constraints.

$$\forall\, x : G \cdot (Current \neq Void \wedge (push(x);\ pop) \rightarrow (Current = \mathbf{old}\ Current)$$
$$\forall\, x : G \cdot (Current \neq Void \rightarrow (Current.push(x).top = x)$$

Work is ongoing to integrate **describe** clauses into the refinement process, and to implement tool support for checking **describe** clauses at run-time.

4 Ongoing Work and Conclusions

We have outlined a refinement calculus for producing immediately executable Eiffel programs from Eiffel specifications. In this manner, refinement is used as a theoretical basis for the seamless development of OO programs.

We are currently pursuing three avenues of research on refinement.

1. Extending the calculus to support further elements of Eiffel, particularly expanded (value) types, exception handlers, and agents (function objects). The extended calculus will be applied in practical refinement examples, particularly, a verification of the EiffelBase standard library [4], which poses particular challenges since it provides incomplete specifications of many classes.
2. Tool support for the calculus. We are building theories for the PVS theorem prover that will be used to help discharge the proof obligations that arise during refinement. The theories constructed in [6] for metamodelling with PVS are proving invaluable in this work.
3. Addition of **describe** clauses to the calculus.

Work on (1) and (2) is well underway; much remains to be done on the effect of using **describe** clauses in refinement.

References

1. J. Guttag and J. Horning. *Larch: Languages and Tools for Formal Specification*, Springer-Verlag, 1993.
2. E.C.R. Hehner. *A Practical Theory of Programming*, Springer-Verlag, 1993.
3. B. Meyer. *Eiffel: the Language*, Prentice-Hall, 1992.
4. B. Meyer. *Object-Oriented Software Construction*, Second Edition, Prentice-Hall, 1997.
5. C.C. Morgan. *Programming from Specifications*, Second Edition, Prentice-Hall, 1994.
6. R.F. Paige and J.S. Ostroff. Metamodelling and Conformance Checking with PVS. In *Proc. Fundamental Aspects of Software Engineering*, LNCS 2029, Springer-Verlag, April 2001.
7. R.F. Paige and J.S. Ostroff. ERC: an Object-Oriented Refinement Calculus for Eiffel. Technical Report CS-TR-2001-5, Department of Computer Science, York University, October 2001.
8. K. Walden and J.-M. Nerson. *Seamless Object-Oriented Software Architecture*, Prentice-Hall, 1995.