DOI 10.1007/s00165-003-0024-3
BCS © 2004
Formal Aspects of Computing (2004) 16: 51–79

Formal Aspects of Computing

ERC – An object-oriented refinement calculus for Eiffel

Richard F. Paige and Jonathan S. Ostroff

Department of Computer Science, York University, Toronto, Ontario M3J 1P3, Canada
Email: {paige,jonathan}@cs.yorku.ca [1]

Abstract. We present a refinement calculus for transforming object-oriented (OO) specifications (or 'contracts') of classes into executable Eiffel programs. The calculus includes the usual collection of algorithmic refinement rules for assignments, if-statements, and loops. However, the calculus also deals with some of the specific challenges of OO, namely rules for introducing *feature calls* and reference types (involving aliasing). The refinement process is compositional in the sense that a class specification is refined to code based only on the specifications (not the implementations) of the classes that the specification depends upon. We discuss how automated support for such a process can be developed based on existing tools. This work is done in the context of a larger project involving methods for the seamless design of OO software in the graphical design notation BON (akin to UML). The goal is to maintain model and source code integrity, i.e., the software developer can work on either the model or the code, where (ideally) changes in one view are reflected instantaneously and automatically in all views.

Keywords: Refinement calculi, Algorithm refinement, Object orientation, Eiffel, Modular reasoning

1. Introduction

It has always been desired that formal methods be applicable to specifying, designing, and verifying large software systems. While important theoretical gains and some practical benefits have been achieved, the application of formal methods to industrial-scale software development has been for the most part limited to critical components of fairly small subsystems or to safety-critical domains.

Object-oriented (OO) software development has been suggested as an important technique for building large, reliable, and maintainable software systems [Lan95, Mey97]. However, the most popular formal notations and methods such as Z [Spi92], VDM [Jon90], CSP [Hoa85], and B [Abr96] do not apply directly to OO software development, since they lack fundamental features like classes, inheritance, and feature redefinition. Object-oriented extensions of these languages, such as Object-Z [Smi00] and VDM++ [Lan95], while removing many of these limitations, do not have realistic target implementation languages, and thus further translation is necessary to produce executable code from refined specifications. Yet many software developers either already are or plan

[1] The authors thank the National Sciences and Engineering Research Council of Canada for their support.
Correspondence and offprint requests to: Richard F. Paige, Department of Computer Science, University of York, Heslington, York, YO10 5DD, United Kingdom. Email: paige@cs.york.ac.uk

on using OO programming languages for their projects. What guidance can formal methodologists offer these developers?

There is a method already available that casts many of the benefits of conventional formal methods – and refinement in particular – into the OO realm. This method is applicable to specification and to the development of immediately executable code. The method is Eiffel [Mey92]. Formal methodologists have paid little attention to Eiffel despite the fact that it appears to be a viable platform for making formal methods directly usable in large-scale software development.

A key element of Eiffel is *design-by-contract* (DbC) [Mey97]. The premise of DbC, in an OO setting, is that routines (e.g., functions or procedures) of a class are given *contracts*. Contracts (a) describe the benefits offered by the class to its clients without describing how these benefits are delivered; (b) define the obligations of the author or supplier of the class to the clients, and the obligations of the clients when using the class; (c) allow for better testing via assertion checking at runtime; (d) define precisely what an exception is (behaviour that does not satisfy the contract); (e) allow for subcontracting so that the meaning of a redefined routine remains consistent with inherited behaviour; and (f) provide documentation to both clients and suppliers of classes.

What is missing from Eiffel is the notion of refining an abstract specification of a class or set of classes with contracts to an immediately executable Eiffel program along with a proof that the program satisfies the specification.

What is missing in conventional formal languages and methods such as Z, B, or tabular specifications [Par92] are the techniques and benefits provided by OO that promote reusability and maintainability, namely, the structuring of large systems via classes, associations, inheritance, and polymorphism. Object-oriented extensions of formal methods such as Object-Z, VDM++, and Larch/C++ [Lea97] do not yet have comprehensive rules for refinement to industrial strength OO languages, but some work is in progress [Smi02].

The purpose of this paper is to present a refinement calculus for generating Eiffel programs. The calculus benefits from use of Eiffel's OO features for structuring specifications and programs. The calculus also targets an immediately executable, industrial-strength programming language with compiler and tool support. The refinement process is *modular*: systems are refined class by class. The refinement of a class proceeds in an environment where only the specifications (and not implementations) of dependent classes, defined in what follows, need be used. A class itself is refined routine by routine.

Informally, suppose that we have an OO system constructed from a universe of classes. One of these classes is the *ROOT_CLASS* [Mey92]; all classes on which this class depends must be in the system. The *ROOT_CLASS* provides a routine from which execution of the OO system will commence. Any class C in this universe can be refined using only its contracts and the contracts of the classes that C depends upon via a restricted set of directed relationships.

Because of modularity, we need only the contracts, and not the implementations, of a few classes to refine the specification of C to an executable program. This is the OO version of the modularity principle of conventional program development: the correctness of a system can be determined from the correctness of its parts without the need to know the internal structure of its parts. In the case of an OO system constructed from the aforementioned universe of classes, it is sufficient to refine the *ROOT_CLASS* of the system. Doing this will recursively trigger a process wherein all other classes in the system are eventually refined. At each step of the process, modularity applies, and we can refine a class by using only the contracts of related classes.

1.1. Organization of the paper

In Section 2 we provide an overview of a significant subset of Eiffel's syntax and semantics, concentrating on those elements used for the specification of systems, as well as those that will be generated as output of the refinement calculus. We also describe key elements of BON [WN95], a graphical modelling language that can be used to visualize structural and behavioural aspects of Eiffel programs, and also to assist in making Eiffel easier to use in the large.

Sections 3 and 4 contain the main contributions of the paper. Section 3 provides a fundamental set of refinement rules for Eiffel, focusing on procedural language constructs, e.g., loops and sequencing. An important contribution of Section 3 is a theorem that allows us to reuse Z and Morgan refinement rules, tranformed into the predicative calculus of Hehner. Section 4 extends the rule set to include ones for introducing feature calls. As is explained in Section 2, Eiffel possesses both *reference types* and *expanded types* (sometimes referred to as 'sub-object types'). We formulate a theory of reference types in Section 4, and by default assume that all types – except primitives such as integers and booleans – are references. We also suggest how our approach can be extended

to handle full expanded types as defined in Eiffel. In Section 5 we explain the modular nature of refinement in Eiffel, and provide a process for refining a specification into executable Eiffel code; the process is demonstrated by examples in the appendices. In Section 6 we discuss automation, and our ongoing work on supporting refinement and verification with Eiffel using PVS. Finally, in Section 7, we discuss related work.

2. Eiffel and BON

Eiffel is OO programming language and method [Mey92, Mey97]; it provides constructs typical of OO paradigm, including classes, objects, inheritance and client–supplier relationships, generic types, polymorphism and dynamic binding, and automatic memory management. However, Eiffel is not just a programming language – the notation also includes the notion of a *contract*. Since contracts can be used to specify software, Eiffel can also be used as a notation for analysis and design. The basic unit of modularity in Eiffel is the class, whose features can be specified via contracts. The notation is seamless in the sense that a single type of abstraction - the class - can be used throughout development, and the contracts of classes can be refined and extended to implementation within the same semantic framework. The basic concepts needed to model objects representing such external concepts as hospitals and nuclear reactors are not essentially different from what is needed for objects representing floating point numbers, stacks, and queues.

The BON modelling language [WN95] developed the ideas of Eiffel in the area of analysis and design. The result is a method that contains a set of concepts and corresponding graphical notations to support OO modelling centred around the three principles of seamlessness, reversibility (the ability to produce BON diagrams automatically from Eiffel programs), and contracting. BON can be used independently of Eiffel. It has been used successfully over the years at Enea in Sweden in industrial projects with such diverse languages as C++, SmallTalk, and Object Pascal. Although BON is a language-independent method, its basic concepts are similar enough to Eiffel that its graphical notation may be viewed as a graphical dialect of Eiffel for the purpose of this article. While systems can be developed entirely using Eiffel only, the graphical facilities of BON can help document and manage large-scale systems, can enable the use of tools, and are attractive to developers. We thus view BON as a means of enabling the use of Eiffel – and refinement in an Eiffel context – in the large.

We start with a brief overview of Eiffel/BON, focusing on the programming constructs that can be introduced during refinement. In order to understand these constructs and how to refine specifications, we need to understand the effect that program constructs have when they execute. Thus, we provide a brief description of Eiffel's syntax and semantics.

2.1. Runtime structure of Eiffel programs

At runtime, Eiffel programs create *values* – which are either *objects* or *references* to objects – in the memory of a machine. Objects can be *basic* (e.g., booleans, characters, integers, and reals) or *complex*, in which case they have zero or more *fields*. In turn, a field also consists of a value. Every object is an instance of a *type*, e.g., the values '1', '2', etc., are all instances of type *INTEGER*.

A reference is a value which is either *Void* or *attached* to an object. If a reference is *Void*, then no further information is available about it. If it is attached, then the reference gives access to the object. A reference is thus attached to zero or one objects. An object may in turn be attached to zero, one, or more objects because its fields may be references.

Computation proceeds by the creation of values, the attachment (or reattachment) of references to objects, accessing objects or their fields, and routine computation, which might involve changing object fields. Hence, at any instance during its execution, a machine executing an Eiffel program will have created a runtime structure consisting of a number of references and objects, and a computation step (via creation, attachments, and feature calls) will take us from one such runtime structure to a new one.

Procedural programming languages have the notion of a *variable*. An *entity* is OO generalization of the notion of a variable. An entity is a name in a software text, meant to be associated at runtime with one or more successive values, under the control of attachment and reattachment operations such as creation, assignment, and argument passing. Every entity is declared to be a particular type, and thus every object (accessed via entities) is a direct instance of some type.

An entity is either an attribute, the argument of a feature call, or a local entity of a routine (including the local entity *Result* of a function routine, as will be discussed in what follows). Entities must be declared in the

program text before they are used, e.g., $e : BOOLEAN$. Eiffel also provides support for expressions involving prefix and infix operators, but these are just syntactic sugar for query calls. An entity can be declared *expanded* (using the notation $e1$: **expanded** C) or *reference* (using the notation $e2 : C$), where C is a type (i.e., a class). Entity $e1$ denotes a reference which may become attached to an instance of C, whereas the expanded entity $e1$ directly denotes an object which is an instance of C.

Two consequences follow for the expanded entity $e1$: (a) the expression $e1 = Void$ always yields the value *false*, and (b) if $e1$ is associated with an instance of C called *obj*, then *obj* cannot be shared, i.e., no other entities may be attached to *obj*.

As mentioned earlier, an entity e is associated with a value, and a value is either a reference to an object or the object itself. If entity e is associated with a basic (respectively complex) object we write *basic*(e) (respectively *complex*(e)). If the e is expanded we write *expanded*(e), and if e is a reference to an object, we write *reference*(e). We use $e1 \stackrel{r}{=} e2$ for *reference equality* and *equal*($e1, e2$) for the weaker notion of *object equality* (field-by-field equality) – these notions are defined more precisely in what follows. We use the notation *same_type*($e1, e2$) to assert that $e1$ and $e2$ have precisely the same type (and hence the same fields).

It is normally clear from the context what we mean when using the unadorned equality symbol, i.e., $e1 = e2$. We mean $e1 \stackrel{r}{=} e2$ if $e1$ and $e2$ are both references. In all other cases we mean *equal*($e1, e2$).

2.2. Specification constructs in Eiffel

The fundamental specification construct in Eiffel is the class. A class is both a module and a type.[2] A class has a name, an optional class invariant, and a collection of features that must preserve the invariant (as is described in the following).

A *system* results from the assembly of one or more classes to produce an executable unit. A *cluster* is a set of related classes. A *universe* is a set of clusters, out of which developers will pick classes to build systems. Of these, only the *class* corresponds directly to a construct of the language. Clusters and universes are not language constructs, but mechanisms for grouping and storing classes using facilities provided by the underlying operating system such as files and directories.

Viewed as a type, a class describes the properties of a set of possible data structures (objects) which are instances of the class. Viewed as a module, the class has a set of *features*. Some features, called *attributes*, represent fields of the class's direct instances; others, called *routines*, represent computations applicable to these instances.

Features can also be categorized as either *queries* or *commands*. A query is a side-effect-free function[3] that returns a value, but does not change the runtime structure. A command may change the runtime structure but returns nothing. A query is either a function (i.e., it returns a computed value) or an attribute.

Figure 1 contains a short example of an interface of the class *CITIZEN*. Each feature section, introduced with the key word **feature**, is followed by a selective export clause that specifies a list of accessor classes. The feature *salary*, for example, can only be accessed by the client classes *EMPLOYER* and *GOVERNMENT*, or by clients that are descendants of them.

Queries and commands 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. Query routines always have a local entity *Result* of the same type as the return value of the query — the result returned by a call to the query is the final value of *Result*. *Result* is the only local entity that can appear in routine postconditions.

The **modifies** clause of a routine is a frame indicating those attributes that may be changed by the routine.[4] Preconditions and class invariants are called *single-state* assertions as they have no occurrences of **old**, whereas the postcondition is a *double-state* assertion as it refers to the old state as well as the new state. The assertion language is enhanced by the fact that assertions may refer to any query (e.g., the postcondition of *divorce* refers to the query *single*).

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). Private

[2] This definition of a class has received criticism; however, it makes the theory and programming language simple and practical.

[3] Technically, Eiffel functions can also change the values of objects and references, but for the purposes of this paper we disallow such changes and require that a query be a pure function. Functions in any case may change local entities including *Result* (defined in what follows).

[4] Eiffel supports **require** and **ensure** assertions, but **modifies** is our addition. The **modifies** clause can be enforced by a suitable postcondition. It is adopted from [Mor94] and [LB00], among others.

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

feature {EMPLOYER, GOVERNMENT}
        salary : REAL

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

Fig. 1. The class *CITIZEN*

features local to a class may temporarily invalidate the class invariant. In the invariant the symbol *Current* refers to the current object; it corresponds to `this` in C++ and Java. Clauses in the invariant may be given text labels (see Fig. 1).

The basic mechanism of OO computation is the feature call $target.f(x)$ where *target* is an expression and f is a feature. The feature may have zero or more arguments. *Current* is always attached to the current object. *Current* thus means 'the target of the current call'. Thus, for the duration of the call $target.f(x)$, *Current* denotes the object attached to *target*.

The Eiffel assertion language allows quantified expressions such as $\forall e : T \mid R \bullet P$ where variable e of type T is the bound variable, R is the domain restriction, and P is the predicate part. The 'it holds' operator $\bullet$ is right associative.[5]

In Fig. 1 the class *CITIZEN* has eight queries and one command. The attributes *age*, *salary*, and *single* are basic, and hence by definition expanded; the remaining attributes are references (a non-basic attribute is a reference unless declared **expanded**). The query *single* returns a value of type *BOOLEAN* (but does not change any attributes), while *divorce* is a parameterless command that changes the state of a citizen object (i.e., changes the attributes). The class $SET[G]$ is a generic predefined class with generic parameter G and the usual operators (e.g., $\in$, *add*). The class $SET[CITIZEN]$ thus denotes a set of objects each of type *CITIZEN*.

Short forms of assertions are permitted. For example, consider a query $children : SET[CITIZEN]$. Then $\forall c \in children \bullet P$ is an abbreviation of $\forall c : SET[CITIZEN] \mid c \in children \bullet P$. The last invariant clause of *CITIZEN* (Fig. 1) thus asserts that each child of a citizen has the citizen as one of its parents. The first invariant asserts that if you are a citizen, then you are either single or married to somebody who is married to you. The second invariant asserts that a citizen has no more than two parents.[6]

Eiffel syntactic constructs may be divided into *expressions* (denoting values) and *instructions* (performing computations). In contrast to instructions, expressions denote values (references or objects) determined at runtime, but their evaluations do not change the runtime structure; in other words, expressions, including query calls, do not have side - effects.

As mentioned earlier, an entity is either an attribute, the argument of a feature call, or a local entity of a routine (including the local entity *Result* of a function routine). More complicated expressions are built from entities using queries, e.g., $e1.q(e2)$ or $e1.q1(e2.q2(e3))$. Entities must be declared in the program text before they

[5] Our assertion language uses the BON notation for quantifiers; quantifiers over finite domains can be implemented in the current version of the Eiffel compiler using tuples and agents [Mey00].

[6] If we declared $parents.count = 2$, then it would make implementation of the specification difficult, as every parent would (recursively) have to be created with references to their parents, leading to a possibly infinite data structure. With the current definition, a parent field can be *Void* indicating that we have not yet specified who the parent is.

Table 1. Naming conventions for entities and features

e	entities (includes attributes a, arguments of routines x, local variables, *Current*, *Result*)
$e.type$	the static type of the entity e
$e.\mu$	the syntactically legal multi-dots associated with e (see sequel)
c	commands
q	queries
a	attributes
f	features, i.e., queries, attributes, and commands
r	routines (computation)
$r.\rho$	bunch of reference entities associated with r (see sequel)
$r.\pi$	bunch of entity groups associated with r (see sequel)
S	single-state formulae
D	double-state formulae
P, Q	predicates including single-and double-state formulae
$\backslash e$, $\backslash P$	abbreviation for **old** e and **old** P
$\Delta(e1, e2, \ldots)$	abbreviations for *modifies*$(e1, e2, \ldots)$
$\Xi(e1, e2, \ldots)$	abbreviation for *same*$(e1, e2, \ldots)$

Table 2. Precedences from highest (level 0) to lowest (level 9)

0	$+, -, \neg$, **pre**, **old** (unary prefix operators), . (feature dot notation)
1	$*, /$
2	$+, -, \cap, \cup$, etc.
3	relations: $=, \overset{r}{=}, \neq, <, \leqslant$, etc., as well as $\in$ and and bunch union ('.')
4	$\wedge, \vee$ (logical operators)
5	$\rightarrow, \leftarrow, \sqsubseteq$
6	$\equiv, \not\equiv$
7	$\bullet, \mid$
8	$:=$ (assignment), $\forall, \exists$
9	$\widehat{=}$ (definition); (assertion clause conjunction or sequential composition)

are used. Table 1 defines the notation that we use throughout the rest of the paper for describing entities and features. Table 2 defines the precedence of operators in Eiffel.

By default, $e : C$ is a reference declaration of entity e unless C is a basic type *INTEGER, REAL, STRING, BOOLEAN, CHARACTER*, in which case it is expanded by default.

BON provides graphical syntax for representing expanded and reference types. In BON, expanded types are called *aggregations*. For example, a class *VEHICLE* might have an attribute *propulsion* defined as *propulsion* : **expanded** *MOTOR*. An engine belongs only to the specified vehicle. An expanded entity faithfully models the fact that a motor is an integral and internal part of a particular vehicle and is not shared with any other vehicle. The aggregation relation between *VEHICLE* and *MOTOR* is shown in Fig. 2.

The BON notation for representing reference types is called an *association*. In the figure the class *VEHICLE* has a reference attribute *owner* with type *ASSET_HOLDER*.

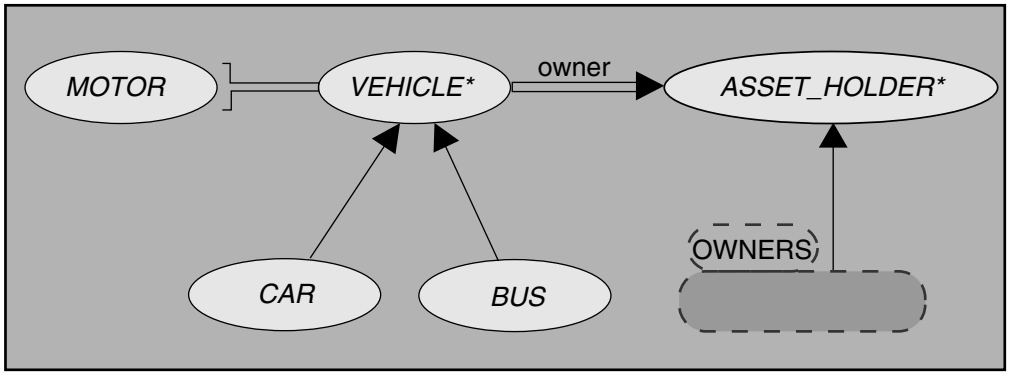

Fig. 2. BON diagram for aggregations, associations, inheritance, and clusters

```
routine      ::=  routine_name(expression : TYPE){: TYPE} is
                   { modifies entity_list }
                   { require single_state_assertion }
                    deferred | body
                   { ensure double_state_assertion }
                    end
body         ::=  {local entity_list} do instruction
instruction  ::=  skip | create e | e.c(expression) | e := expression |
                  instruction; instruction | selection | loop
selection    ::=  if bool then instruction
                  { elseif bool then instruction .} { else instruction } end
loop         ::=  modifies entity_list
                  from instruction
                  invariant double_state_assertion
                  variant integer_expression
                  until bool
                  loop instruction end
```

Fig. 3. Syntax of a routine

A child class can inherit properties from one or more parent classes, thus defining a behavioural subtyping relationship between child and parents. Eiffel supports only strong behavioural subtyping [DL01]. Child classes are always subtypes of parent classes, class invariants may be strengthened by child classes, preconditions of routines may be weakened, and postconditions may be strengthened. The BON notation for representing inheritance is drawn with a single-line arrow. Thus *CAR* and *BUS* inherit from *VEHICLE*. The compressed cluster *OWNERS* contains a number of classes (e.g., *PERSON*, *COMPANY*, etc.) that all inherit from *ASSET_OWNER*. All arrows, whether aggregation, association, or inheritance, point in such a way as to show dependencies. Thus *CAR* depends on *VEHICLE*, but not vice versa.

2.2.1. Routines

All features of a class other than attributes are *routines*. The syntax of a routine of a class is shown in Fig. 3. Constructs in curly parentheses are optional, and `bool` denotes an expression of type *BOOLEAN*. The Eiffel assertion language is used to express preconditions, postconditions, and class and loop invariants of routines. A routine has zero or more parameters (though for conciseness, the grammar in Fig. 3 shows only one parameter).

The behaviour of **create**, the assignment, and procedure call instructions are described more precisely in later sections. A loop in Eiffel is executed as follows: the initialization (the **from** statement) is executed; then the condition *bool* is evaluated, and the loop terminates if it is true; if it is false, the body of the loop is executed, and then the condition is re-evaluated. The invariant, a double-state assertion[7], must be established by the initialization and must be true when the loop body finishes its execution. The variant must be decreased by each execution of the loop body. A selection statement is executed in the usual way.

All computation is performed either by (a) object creation, (b) attachment and detachment (e.g., via an assignment statement), or (c) by feature calls. A feature call $target.f(x1, x2)$, where *target* is an expression, f is a feature name of the appropriate class, and the arguments $x1$, $x2$ are expressions, means apply feature f to the object represented by *target* using arguments $x1$ and $x2$. For simplicity, we can assume that *target* is either an entity or a single-dot call. By not considering multi-dot expressions we are simply assuming that $e1.e2.f(x1, x2)$

[7] In the implemented Eiffel language, invariants are single-state assertions. The use of a double-state assertion allows us to verify more properties without having to use auxiliary variables.

is equivalent to the compound code (**local** $e3$; $e3 := e1.e2$; $e3.f(x1, x2)$). If the target is a complex expression, then we can replace $target.f(x1, x2)$ by

(**local** $e : T$; $e := target$; $e.f(x1, x2)$)

where an appropriate type T has been chosen for entity e.

In what follows, we link Eiffel *expressions* to values (objects and references), and Eiffel *instructions* to computation via object creation, attachment, and feature call. The informal semantics provided in this section are used to motivate the axiomatic refinement calculus of Eiffel programs and specifications in the next section.

3. Fundamentals of the Eiffel 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 [Heh93]. To Hehner's calculus, we provide a refinement rule for introducing loops. We also add new machinery for introducing Eiffel's OO constructs, such as object creation and feature calls.

3.1. Specifications and programs

In ERC a *specification* of a feature is expressed as a double-state predicate. The quantities of interest in specifying the behaviour of an Eiffel construct are the poststate σ after the construct's computation terminates (the output), as well as the prestate **old** σ (the input). Given a routine r of a class C, we let $r.\sigma$ denote the state space of the routine, which includes the attributes of the class containing r, the routine arguments, the local entities of the routine, and a conceptual global time variable t (note that Eiffel itself does not provide any notion of a global variable, unlike, e.g., C++). Thus, σ is a *bunch*[8] of entities $\sigma \mathrel{\widehat{=}} e1, e2, \ldots, en, t$. Correspondingly, **old** $\sigma =$ **old** $e1$, **old** $e2, \ldots,$ **old** en, **old** t. If $\bar{\sigma}$ is a sub-bunch of σ, then we define $same(\bar{\sigma})$ by $same(\bar{\sigma}) \mathrel{\widehat{=}} (\forall \hat{e} \in \bar{\sigma} \bullet \hat{e} =$ **old** $\hat{e})$. Provided $t \notin \bar{\sigma}$, then we define $modifies(\bar{\sigma}) \mathrel{\widehat{=}} (\forall \hat{e} \in \sigma - (\bar{\sigma}, t) \bullet \hat{e} =$ **old** $\hat{e})$. We use the following abbreviations: $\backslash e$ for **old** e, $\Xi(e)$ for $same(e)$, and $\Delta(e)$ for $modifies(e)$. We note that

$$\Delta(\bar{\sigma}) \rightarrow \Xi(\sigma - \bar{\sigma}, t) \tag{1}$$

Hence, $\Delta(e1, e2)$ asserts that the entities $e1$, $e2$, and t may all change, while all other entities in σ remain the same.

Note that if entity $e1$ appears in a frame, and $e1$ is a reference variable, then the reference may change, but attributes of the attached object cannot change. Thus it would be illegal to have $e1$ in the frame of a specification with a postcondition that changes $e1.a$, for example. If a specifier explicitly wants to change $e1.a$ in a postcondition then $e1.a$ must be stated in the **modifies** clause. Similarly, supposing that A is an array of integers, A appearing in a frame indicates that the reference of A may change, not the elements.[9] Finally, *Result* appears in the **modifies** clause of all queries, since it can be used like any other local entity (and may be assigned to repeatedly in the course of execution of the body of the query).

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. A computation is described by a given prestate and a computed poststate that makes the specification true. Thus, suppose the routine r has a precondition $r.pre$ (a single-state predicate with free variables in σ) identifying the prestates, and a postcondition $r.post$ (a double-state predicate with free variables in $\backslash\sigma$ and σ) indicating the computed poststates, then we define the specification $r.spec$ of the routine as follows:

$$r.spec \mathrel{\widehat{=}} \backslash(r.pre) \rightarrow r.post \wedge time \tag{2}$$

$$time \mathrel{\widehat{=}} t \geqslant \backslash t \wedge (\backslash t < \infty \rightarrow t < \infty) \tag{3}$$

[8] The notion of a bunch is taken from [Heh93]. A set s, e.g., $s = \{e1, e2, e2\}$ is a collection of entities in a package. A bunch is the contents of a set without the package. Thus the bunch b corresponding to the set s is $e1, e2, e3$. The symbol $\sim$ is used to obtain the contents of a set; thus $\sim s = b = e1, e2, e3$. The standard operations of set theory are used for bunches, e.g., given a bunch $b2 = e4, e5$, then $b \cup b2 = e1, e2, e3, e4, e5$ – we also write this union as '$b, b2$' — and $e5 \in b2$ expresses the fact that bunch $b2$ contains $e5$.

[9] Array elements in Eiffel are accessed via the call to the function $item(i)$, and thus cannot appear in frames since the elements are not attributes.

where t is a conceptual global clock representing time, and $t \in \sigma$; by default, t appears in the frame of every specification. Operator precedences were given in Table 2. Various ontologies of time can be used (e.g., recursive time, real time [Heh93]) but for our purposes we simply require that each program construct terminates in finite time (provided it starts at a non-infinite time).

Specification *r.spec* states that if the routine precondition is satisfied in the prestate of a computation, then the routine's execution terminates in finite time with the postcondition true in the poststate. If the precondition is not satisfied in the prestate, then any behaviour is permitted. A subset of programs in Eiffel can be described in ERC as follows (we omit the OO constructs, which are covered in Section 4):

$$\begin{aligned}
\textbf{skip} &\mathrel{\widehat{=}} e1 = \backslash e1 \wedge e2 = \backslash e2 \wedge \ldots \wedge en = \backslash en \wedge time \\
e1{:=}exp &\mathrel{\widehat{=}} defined(\backslash exp) \;\rightarrow\; e1 = \backslash exp \wedge \Delta(e1) \wedge time \\
\textbf{if } b \textbf{ then } P \textbf{ else } Q \textbf{ end} &\mathrel{\widehat{=}} defined(\backslash b) \;\rightarrow\; (\backslash b \rightarrow P) \wedge (\neg \backslash b \rightarrow Q) \\
P;\; Q &\mathrel{\widehat{=}} \exists \hat{\sigma} \bullet P[\sigma := \hat{\sigma}] \wedge Q[\backslash\sigma := \hat{\sigma}] \\
(\textbf{local } e : T;\; P) &\mathrel{\widehat{=}} (\exists e, \backslash e : T \bullet P) \\
\textbf{check } S \textbf{ end} &\mathrel{\widehat{=}} S \rightarrow \textbf{skip}
\end{aligned} \tag{4}$$

where P and Q can themselves be specifications or programs; we can mix programs and specifications because they are both described by predicates. The predicate $defined(exp)$ describes the conditions under which the expression exp can be evaluated (so, for example, $defined(e1 \div e2) \mathrel{\widehat{=}} e2 \neq 0$). The notation $P[e := exp]$ means safely replace every free occurrence of variable e in predicate P by the expression exp. A full range of program constructs is defined in [Heh93]. The semantics of assignment defined in (4) only covers the case where both the entity $e1$ and the expression exp are **expanded**. In the following, we develop special machinery for Eiffel reference assignments. Note that for the local variable declaration, it is necessary to quantify over both pre- and poststate because specification P may refer to both.

It is tedious to always write out the entities that do not change in a specification, so we use the Morgan specification statement [Mor94] for writing specifications, and take advantage of its notation for writing frames. However, we define the meaning of specification statements using timed predicates (rather than weakest preconditions). The notation

$$e : \llparenthesis\, S, D \,\rrparenthesis \;\mathrel{\widehat{=}}\; \backslash S \;\rightarrow\; D \wedge time \wedge \Delta(e)$$

denotes a specification[10] for which e is the frame (the bunch of entities that may change between the prestate and poststate), S is the precondition, and D is the postcondition. An equivalent (Larch-like) syntax for writing specifications which cannot be written conveniently on one line is:

modifies e
require S
ensure D

Note that the variable e in the frame means that it *may* change, not that it must change. Using the specification syntax, we can describe the program constructs in (4) more concisely. For example, the expanded assignment statement $e1 := exp$ can be written as $e1, t : \llparenthesis\, defined(exp),\; e1 = \backslash exp \,\rrparenthesis$. Many useful laws of programming can be derived from the definitions of programs in (4). For example, it is straightforward to provide a proof using the relevant predicative definitions for the following:

$$defined(\backslash b) \rightarrow (\textbf{if } b \textbf{ then } P \textbf{ else } P \textbf{ end} \;\equiv\; P) \tag{5}$$

$$[D_1 \rightarrow S_2] \rightarrow ((D_1 \wedge time \;;\; \backslash S_2 \rightarrow D_2 \wedge time) \equiv (D_1 \;;\; D_2) \wedge time) \tag{6}$$

$$D_1 \wedge time;\; D_2 \wedge time \;\equiv\; (D_1;\; D_2) \wedge time \tag{7}$$

$$[D_1 \rightarrow D] \;\rightarrow\; (D_1;\; D_2) \rightarrow (D;\; D_2) \tag{8}$$

where D_1, D_2 are double-state assertions, S_2 is a single-state assertion in σ, and

$$[D(\backslash\sigma, \sigma)] \mathrel{\widehat{=}} \forall \backslash\sigma, \sigma \bullet D(\backslash\sigma, \sigma)$$

The *time sequencing* rules ((6) and (7)) assert that we can ignore the embedded *time* components (under the stated assumption, where necessary), provided that the execution of each construct in the sequence itself terminates in finite time.

[10] Morgan's syntax is actually $e : [S, D]$, but we use the square brackets for substitution and double-state quantification.

The *simple substitution rule* is derived from the assignment definition and sequencing (4) and is similar to a corresponding rule in [Heh93]. It is particularly useful in simplifying long sequences of assignments.

Rule 3.1 (Simple Substitution). For any expanded entity $e1$ not in bunch e, and expression exp whose type conforms to $e1$,

$$e1 := exp;\ e : (\!|\ S, D\ |\!) \quad \equiv \quad e, e1 : (\!|\ S[e1 := exp], (D \wedge e1 = \backslash e1)[\backslash e1 := \backslash exp\ ]\ |\!)$$

An ERC specification *spec* can, in general, be any predicate with free variables in $\backslash\sigma$ and σ. The specification *false* describes no computations, while *true* describes arbitrary behaviour, including non-terminating computations. In order to know if it is feasible to refine a specification to an executable program, we need to know when a specification is implementable.

$$spec \text{ is implementable} \quad \widehat{=} \quad \forall \backslash\sigma \bullet (\exists \sigma \bullet spec \wedge t \geqslant \backslash t) \tag{9}$$

The definition asserts that (a) every prestate of an implementable specification must have at least one corresponding poststate, and (b) time cannot decrease between the input and the output. For example, the specification $Q \mathrel{\widehat{=}} x = 2 \wedge t < \infty$ is unimplementable, because $Q \wedge t \geqslant \backslash t$ is unsatisfiable for an initial value of $\backslash t = \infty$.[11] By contrast, given a prestate in which $\backslash t = \infty$, the *skip*-like program $e = \backslash e \wedge time$ does have a corresponding poststate (namely $t = \infty$).

3.2. 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 $spec \sqsubseteq impl$. Our treatment of specifications as predicates leads to a very simple definition of refinement:

$$spec \sqsubseteq impl \quad \widehat{=} \quad (\forall \backslash\sigma, \sigma \bullet impl \rightarrow spec) \tag{10}$$

3.3. Reuse of rules and loops in ERC

Given a specification $e : (\!|\ S,\ D\ |\!)$, we would like to know under what condition we can write $e : (\!|\ S,\ D\ |\!) \sqsubseteq Loop$, where *Loop* is an Eiffel loop. Although there is a loop structure in [Heh93], the semantics is given in terms of least fixed points. An alternative rule in [Heh93] unfolds loops using recursive calls (and this is actually the recommended technique used for producing looping computations). We desire a refinement rule stated in terms of a variant and invariant of the loop, which is the way it appears in Eiffel. Morgan [Mor94] provides a loop refinement rule in terms of a single-state invariant whereas we want the convenience of being able to use a double-state invariant (since it allows us to include frame conditions in invariants, and it makes it easier to show that a loop establishes a double-state postcondition). Z has such a loop refinement rule [Wor94], and we thus reuse the Z rule in this work. However, we must first justify the fact that we can use Z refinement rules in ERC.

In the discussion below, we restrict our attention to feasible specifications and refinement rules. ERC can express *abort* (the set of all non-terminating computions in ERC is described by $t = \infty$ and its refinements), and a *miracle* (the set of no computations) is described in ERC by *false*. The miraculous Morgan specification statement $e : (\!|\ true,\ false\ |\!)$ cannot be expressed in Z; nor can the set of all non-terminating computations. Thus, when converting between ERC, Z, and Morgan, we ban infeasible specifications (i.e., *abort* and *miracle*).

As mentioned earlier, a specification statement can be translated into a timed predicate as follows (assuming that semantically equivalent expressions and operators can be translated appropriately):

$$e1, t : (\!|\ S,\ D\ |\!) \quad \widehat{=} \quad \backslash S \rightarrow D \wedge \Delta(e1) \wedge time \tag{11}$$

Consider a Z schema specification, with predicate P (this predicate is just the conjunction of the type declarations, precondition, and postcondition of the Z schema). The precondition $P.pre$ of a Z predicate P is defined as $P.pre \mathrel{\widehat{=}} (\exists \sigma' \bullet P)$, where σ' denotes the poststate. Thus, the Z schema can also be thought of as specified by a pair $(P.pre, P)$. We can translate any Z predicate P into ERC notation by prefixing the unprimed entities by **old** or $\backslash$, and by removing the primes from all primed entities.

[11] Infinite starting times are possible, e.g., in the pathological program 'infiniteloop; Q', the program Q would start at time $t = \infty$.

Given two Z schemas with predicates P_1 and P_2, respectively, then refinement, under the above mentioned syntax translation, is [Wor94]

$$P_1 \sqsubseteq_z P_2 \;\widehat{=}\; (\forall \backslash\sigma, \sigma \bullet P_1.pre \rightarrow P_2.pre \wedge (P_2 \rightarrow P_1)) \tag{12}$$

whereas the ERC version of refinement (10) was expressed as

$$P_1 \sqsubseteq P_2 \;\widehat{=}\; (\forall \backslash\sigma, \sigma \bullet (P_2.pre \rightarrow P_2) \rightarrow (P_1.pre \rightarrow P_1)) \tag{13}$$

Since (12) entails (13), we have the following theorem:

Theorem 1 (Refinement Rule Reuse). Given two Z specifications expressed (under syntax translation) as specification statements $e : \llparenthesis S_1,\ D_1 \rrparenthesis$ and $e : \llparenthesis S_2,\ D_2 \rrparenthesis$, it follows that

$$e : \llparenthesis S_1,\ D_1 \rrparenthesis \sqsubseteq_z e : \llparenthesis S_2,\ D_2 \rrparenthesis \quad \rightarrow \quad e : \llparenthesis S_1,\ D_1 \rrparenthesis \sqsubseteq e : \llparenthesis S_2,\ D_2 \rrparenthesis$$

Since refinement in Morgan's calculus is equivalent to (13) [LM99], it follows that any refinement rule for feasible specifications from [Mor94] and [Wor94] will also work in ERC. For example, we can reuse the frame change and local variable introduction rules (Rule 3.2) from [Mor94].

Rule 3.2 (Morgan Refinement Rules).

(a) Frame Change Rule: Let $e1$ and $e2$ be disjoint bunches of variables. Then

$$e1 : \llparenthesis S,\ D \rrparenthesis \;\equiv\; e1, e2 : \llparenthesis S,\ D \wedge e2 = \backslash e2 \rrparenthesis$$

(b) Local variable introduction: Let x be a fresh variable. Then

$$e : \llparenthesis S,\ D \rrparenthesis \;\sqsubseteq\; \textbf{local}\ x : T;\ e, x : \llparenthesis S,\ D \rrparenthesis$$

We now consider loops. Before we present the refinement rule, we introduce some notation, allowing us to talk about the intermediate states that arise during a loop computation. We annotate specifications with primes (e.g., Q') to indicate systematic addition of primes to *free variable names* used within the specification. A prime applied to an **old** expression (or backprimed expression) removes the **old** keyword (or backprimes). Here is an example.

$$(x = \backslash y \wedge y = \backslash(x + y))' = (x' = y) \wedge (y' = (x + y))$$

We can now state the refinement rule for loops; it is shown in Rule 3.3. The rule is a reformulation of the corresponding Z rule [Wor94, p. 218], and relies on Theorem 1.

The invariant I in Rule 3.3 is a relation between two points in the execution of the loop — the first point is before the loop execution begins (at which point the state is denoted by $\backslash\sigma$) and the second point is when the 'until' exit test is made (and the state is σ). The invariant must always hold true when the 'until' test is made at the exit point.

The loop first executes the initialization instruction *Init*. The first 'provided that' clause is a safety condition that says that any circumstances acceptable to the specification (precondition S) must also be acceptable to the initialization (precondition S_{init}).

Rule 3.3 (Initialized Loop Rule). Suppose we have an Eiffel loop as follows:

Loop ::= **modifies** e
from *Init*
invariant I
variant v
until b
loop
Body
end

where *Init* is $e, t : (\!| S_{init},\ D_{init} |\!)$, *Body* is $e, t : (\!| S_{body},\ D_{body} |\!)$, b is a boolean expression, I is a double-state loop invariant, and integer expression v is the loop variant. Let $\bar{\sigma} = \sigma - e, t$. Then, given a specification $e, t : (\!| S,\ D |\!)$, it follows that

$$e, t : (\!| S,\ D |\!) \sqsubseteq \mathit{Loop}$$

provided that

$$\begin{aligned}
S &\rightarrow S_{init} \\
\backslash S \wedge D_{init} \wedge same(\bar{\sigma}) &\rightarrow I \\
\backslash S \wedge I \wedge b &\rightarrow D \\
\backslash S \wedge I \wedge same(\bar{\sigma}) &\rightarrow defined(b) \\
\backslash S \wedge I \wedge \neg b \wedge same(\bar{\sigma}) &\rightarrow S_{body} \\
\backslash S \wedge I \wedge \neg b \wedge (D_{body} \wedge same(\bar{\sigma}))' &\rightarrow I[_ := _'] \\
\backslash S \wedge I \wedge \neg b \wedge same(\bar{\sigma}) &\rightarrow v \geqslant 0 \\
\backslash S \wedge I \wedge \neg b \wedge (D_{body} \wedge same(\bar{\sigma}))' &\rightarrow v' < v
\end{aligned}$$

The notation $I[\_ := \_']$ means 'textually substitute primed versions of free variables for unprimed versions (and do not change the **old** variables)'.

The second clause in the loop rule says that the initialization must establish the invariant. The third states that the specification precondition S and the invariant must define b.

The fourth clause says that on exit (i.e., b true) the invariant must establish the required specification post-condition D.

The fifth clause asserts that the body is only executed when it is safe to do so (i.e., when its precondition S_{body} holds). The sixth clause is a three-state formula. We assume that the invariant holds between the starting state $\backslash\sigma$ and the exit test state σ. If under these conditions the body is executed taking us to a new state σ', then the invariant must continue to hold between $\backslash\sigma$ and the new state σ'.

The final two clauses assert that the variant must never be less than zero, and that every execution of the body must decrease the variant. If the last two clauses hold, then the loop terminates in finite time and hence the *time* condition (3) of the loop is satisfied.

The infrastructure provided with Hehner's calculus is sufficient for proving termination of recursive calls. In the case of cyclical calls (e.g., where routine a in class A calls routine b in class B, which then calls routine a in class A) it may not be possible to determine statically whether a chain of calls terminates, but the refinement laws of Hehner's calculus – particularly those for refinement in place – will be of use here.

For loop termination we can use the loop rule. For other constructs, we can ignore *time* in our derivations involving sequential computations by appealing to (7), provided we know that that cyclic calls are not involved. Where we do not know that this is the case, the full timed version of sequential calls must be invoked. The examples in what follows do not involve the cyclic case; hence, except for the loop construct we can usually ignore the time component.

4. Refinement rules for feature calls

The preceding sections laid the groundwork for the Eiffel refinement calculus including refinement rules for introducing typical imperative constructs, such as assignments and loops. These rules, while adequate for the refinement of specifications to imperative programs, are inadequate in an OO setting. In OO programs there are two fundamental instructions – command and query calls – which are introduced to effect changes in or test the state of objects. Techniques are therefore needed to introduce calls. In this section we provide refinement rules for feature calls. To accomplish this, we first define a theory of Eiffel reference types, and build the rules on this theory.

It is possible to define refinement rules for introducing command and query calls by reverting to an imperative setting, and using functions. A call $e.r$ would be transformed to a call $r(e)$, and refinement could proceed using rules and techniques that appear in imperative refinement calculi. We desire to carry out refinement in a purely OO style, so that no further translation is needed to produce OO program, and so as to maintain the level of abstraction provided by OO technology.

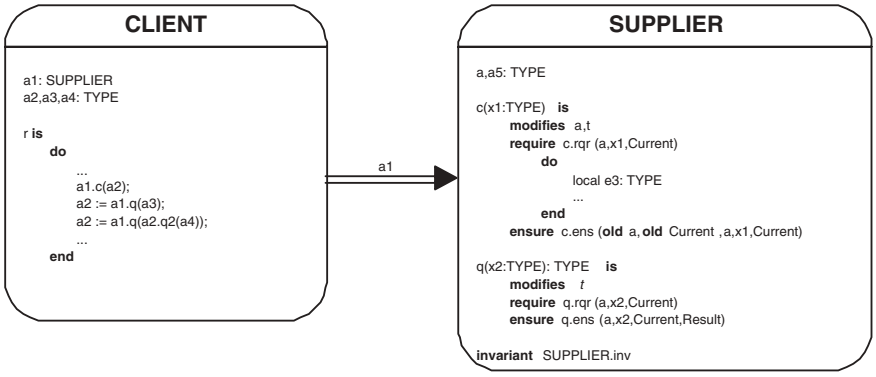

Fig. 4. BON diagram with feature notation

We commence the presentation of the rules with some notation and syntax. The rules and our theory of reference types are given in terms of the BON class diagrams shown in Fig. 4. *TYPE* in Fig. 4 represents an arbitrary type: the type of argument $x1$ may differ from $x2$ (and these in turn may be different from attributes a and $a5$).

Let $C.inv$ denote the invariant of class C, and given routine r, let $r.modifies$ denote the bunch of variables in the **modifies** clause of the routine, and let $r.rqr$ and $r.ens$ denote the **require** and **ensure** clauses, respectively. Then the full pre- and postconditions for each routine[12] are defined as follows:

$$r.pre \mathrel{\widehat{=}} r.rqr \wedge C.inv \tag{14}$$

$$r.post \mathrel{\widehat{=}} r.ens \wedge C.inv \wedge r.modifies \tag{15}$$

where $r.modifies$ is the **modifies** clause of routine r. The routine specification $r.spec$ defined in (2) uses the above definitions.

The state space σ in the context of routine r of the class *CLIENT* will include time t, attributes $a1$, $a2$, $a3$, $a4$ of the class *CLIENT* once they are created, local entities of the routine (including *Result* if the routine is a query), and any further dotted or multi-dot entities (such as $a1.a$ and $a1.a5$) already in existence or brought into existence by feature calls within the body of r. If the class *TYPE* has attributes $a6$ and $a7$, then we can have multi-dot accesses, such as $a1.a.a6$ and $a1.a.a7$, appearing in contracts; this set of multi-dot feature accesses is of arbitrary size. We let $a1.\mu$ denote the bunch of legal multi-dot suffixes m so that $a1.m$ is legal, e.g., $a1.\mu = a, a5, a.a6, a.a7, \ldots$, etc. In general, given an entity $e1$, we let $e1.\mu$ denote the bunch of syntactically legal multi-dot suffixes [Mey92] associated with entity $e1$.

The declaration $e : C$ merely declares the static type of entity e, but does not cause an object to be created at runtime [Mey97, p. 235]. The **create** e instruction in a routine creates a new instance of C, initializes each field to a default value (e.g., *BOOLEAN* to *false*, a reference to *Void*), attaches e to the instance, and calls the creation feature if there is one for that class. The creation feature must ensure that the instance satisfies the class invariant if the default values do not suffice.

4.1. Entity groups for reference types

Reference types are critical for any theory of OO programming. Our theory of reference types, based on so-called *entity groups*, is used by assignment statements, by **create** statements, and also by feature calls.

To model reference types, we could declare a memory store relation $memory : entity \leftrightarrow object$, to keep track of entities and their attachments. Commands like *store* and *select* could be used to update and access memory. Approximately this approach is taken in the LOOP project [BJ01] for reasoning about Java programs in PVS and Isabelle. However, we prefer not to model memory explicitly, as it can lead to lengthy expressions (involving storage arrays and sequences of element overridings) in refinement, although an automated tool could help hide

[12] Except for *creation* routines, which are called when an object is created and attached to an entity. For such routines r, the invariant $C.inv$ need not be true to start with, and as such $r.pre \mathrel{\widehat{=}} r.rqr \wedge default(e)$.

```
class C1 feature {ANY}
        a, b : INTEGER
        c(x : INTEGER) is
                modifies a
                ensure a = x
end

class C feature {ANY}
        e1, e2, e3 : C1
        r is
                modifies e1, e2, e3
                do
                        create e1;
                        create e3;
                        e2 := e1;
                        e1.c(1);
                        e1 := e3;
                        e3.c(3);
                ensure e1.a = 3 = e3.a ∧ e2.a = 1
                end
end
```

Fig. 5. Aliasing and feature calls for references

some of the complexity. Consequently, we introduce below an approach based on entity groups, which does not explicitly model main memory, and which, in our experience, leads to simpler expressions when used in refinement.

Consider a reference declaration $e : C$. On declaration, e refers to no object, i.e., $e = Void$. Objects may be attached to e in one of three ways:

1. Via the instruction **create** e, which generates a new object of type C and attaches it to entity e.
2. Via a type-valid assignment statement $e := exp$, where exp is an expression (e.g., an entity or a query) that evaluates to an object reference, so that the type of the object conforms to C, or $exp = Void$. Both e and exp refer to values that are references (what follows deals with the case in which e is reference and exp is expanded).
3. Via the binding of formal arguments to actual parameters in feature calls (similar to the case of assignment).

create can be applied to an entity at any time during the execution of a program; this adds complexity to the theory of reference types that we provide.

At the heart of our theory of references is the notion of entity groups. 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. The bunch of reference entities $r.\rho$ for routine r in Fig. 5 is given by $r.\rho = e1, e2, e3$.

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 is executed (or, more generally, before the execution of the first **create** statement on an entity in a program), the group is an empty set. After a **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 the bunch of all entity groups of routine r, and we require that these groups be disjoint (this equivalence relation will be formalized in what follows). As an example, in Fig. 5, $r.\pi = \underline{e1}, \underline{e2}, \underline{e3}$. After the assignment $e2 := e1$ in the body of r, we have that $\underline{e1} = \{e1, e2\}$, $\underline{e2} = \underline{e1}$, and $\underline{e3} = \{e3\}$.

For the sake of convenience, we introduce a shorthand for referring to groups. Suppose $\underline{e} = \{e, e2, e3\}$, i.e., the three entities e, $e2$, and $e3$ all refer to the same object. Then

$$\underline{e}[e2, e3] \quad \widehat{=} \quad \underline{e} = \{e, e2, e3\} \tag{16}$$

and to avoid repetition $\underline{e}[]$ means $\underline{e} = \{e\}$.

The notation in (16) is just syntactic sugar for an ERC assertion. For example, suppose $\sigma = t, e, e2, e3, e4$. Then $\underline{e}[e2, e3]$ can be expressed in ERC as $(e = e2 = e3) \wedge (e4 \neq e)$, where the equality symbol is interpreted in the Eiffel assertion language as reference equality ($\stackrel{r}{=}$), as the elements of $\underline{e}$ are all reference entities. When we reason using reference types, we carry around set expressions of the form (16), which describe the attachments of objects to entities. We can now define reference equality using our notion of groups:

$$(e1 \stackrel{r}{=} e2) \;\widehat{=}\; (\underline{e1} = \underline{e2}) \tag{17}$$

where $e1$ and $e2$ are both references.

As mentioned earlier, 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, as follows:

$$\begin{aligned}
equal(e1, e2) &\;\widehat{=}\; \left(\begin{array}{l} (void_case \rightarrow e1 = e2 = Void) \\ \wedge\, (\neg\, void_case \rightarrow same_type(e1, e2) \wedge \alpha \end{array}\right) \\
void_case &\;\widehat{=}\; e1 = Void \vee e2 = Void \\
\alpha &\;\widehat{=}\; \left(\begin{array}{l} (basic(e1) \rightarrow e1 = e2) \\ \wedge\, (complex(e1) \rightarrow reference_case \wedge expanded_case) \end{array}\right) \\
reference_case &\;\widehat{=}\; \forall a \mid attribute(a, e1) \wedge reference(e1.a) \bullet e.a \stackrel{r}{=} e2.a \\
expanded_case &\;\widehat{=}\; \forall a \mid attribute(a, e1) \wedge expanded(e1.a) \bullet equal(e1.a, e2.a)
\end{aligned} \tag{18}$$

where $attribute(a, e1)$ means that a is an attribute of the object associated with $e1$. In the above definition, where $same\_type(e1, e2)$ holds, $e1.a$ is complex (respectively basic) if and only if $e2.a$ is complex (respectively basic). The above definition thus captures the meaning of the standard equality operator of Eiffel (as will appear in a revision to [Mey92]) as follows:

1. If both $e1$ and $e2$ are *Void*, then the result of $equal(e1, e2)$ is true. If one is *Void* and the other is not, then the result is false. If the types are not identical, then the result is also false (this means that $e1$ and $e2$ will also have identical attributes).
2. If $e1$ and $e2$ are basic types (e.g., *INTEGER*, *BOOLEAN*, etc.), then the result is true if and only if both have the same value.[13]
3. If $e1$ and $e2$ are complex objects, then the fields must be identical, i.e., every corresponding reference field of $e1$ and $e2$ must point to the same object, and every subobject field of $e1$ is recursively equal to the corresponding field of $e2$, i.e.,

   $$equal(e1, e2) \rightarrow (\forall m \in e1.\,\mu \bullet e1.m = e2.m) \tag{19}$$

   where $e1.\,\mu$ is, as before, the bunch of syntactically legal multi-dots associated with entity $e1$.

Finally, we need an axiom that asserts that reference equality is at least as strong as object equality:

$$e1 \stackrel{r}{=} e2 \rightarrow equal(e1, e2) \tag{20}$$

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

$$\forall \underline{e1}, \underline{e2} \in r.\pi \;\bullet\; \underline{e1} = \underline{e2} \;\vee\; \underline{e1} \cap \underline{e2} = \varnothing \tag{21}$$

Axiom (21) states that two entity groups are either disjoint or identical. Entity groups are just the equivalence classes induced by the reference equality operator. We should really call a group an 'equivalence class' but refrain from doing thus, so as not to confuse a group with the notion of an Eiffel class. 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. In the example of Fig. 5, after the assignment $e2 := e1$ in the body of r, $\underline{e1}[e2]$ and $\underline{e3}$ are clearly disjoint.

$$\forall \underline{e} \in r.\pi \bullet (\forall e1, e2 \in \underline{e} \bullet e1 \stackrel{r}{=} e2) \tag{22}$$

[13] After possible coercion of the heavier type, an issue which we have neglected for simplicity. In Eiffel there are actually two kinds of objects: *standard* and *special*. Special objects are bit sequences, strings, and arrays. Bit sequence equality is determined by bit-by-bit equality. Strings and array equality holds if they have the same length and each field is (recursively) equal.

Axiom (22) states that if two entities are in the same entity group, they refer to the same object. In the example of Fig. 5, if at a certain execution point $\underline{e1}[e2]$ holds, then $e1 = e2$, $e1.a = e2.a$, and $e1.b = e2.b$.

$$\forall \underline{e} \in r.\pi \bullet \forall e1, e2 \in \underline{e} \bullet \underline{e1} = \underline{e2} \tag{23}$$

Theorem (23) (it follows directly from (22) and (17)) asserts that if two entities are in the same group, then the group can be referred to in expressions by either name.

$$\forall \underline{e} \in r.\pi \bullet \forall e \in \underline{e} \bullet e \neq \mathit{Void} \tag{24}$$

$$\forall e \in r.\rho \bullet (\forall \underline{e}' \in r.\pi \bullet e \notin \underline{e}') \equiv e = \mathit{Void} \tag{25}$$

$$\forall e \in r.\rho \bullet (e = \mathit{Void}) \equiv (\underline{e} = \varnothing) \tag{26}$$

Axiom (24) states that any entity in a group is attached to an object, and thus cannot be *Void*. Axiom (25) 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 (26) equates a *Void* reference with an empty entity group.

The notion of entity groups can be used to define the semantics of instructions that use reference types in ERC such as the **create** statement, assignment, and feature call. The **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. The semantics of **create** e is given in Definition 4.1.

Definition 4.1 (Entity Creation Semantics). Given a reference entity e, the instruction

create e

is defined as

modifies e
ensure $\underline{e} = \{e\} \wedge \mathit{default}(e)$

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 *BOOLEAN* it is set to false, if it is 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., **create** e; $e.r$.

The following theorems follow from the group axioms, and are useful when refining **create** statements and other pieces of code. We assume that e and $e1$ are distinct names; the theorems apply if e or $e1$ is replaced by a dot expression, e.g., $e.a$.

$$\Delta(e) \wedge e \notin \underline{e1} \wedge \backslash e \in \backslash \underline{e1} \rightarrow \underline{e1} = \backslash \underline{e1} - \{\backslash e\} \tag{27}$$

$$\Delta(e) \wedge e \notin \underline{e1} \wedge \backslash e \notin \backslash \underline{e1} \rightarrow \underline{e1} = \backslash \underline{e1} \tag{28}$$

$$\Delta(e) \wedge \underline{e}[] \rightarrow \underline{e1} = \backslash \underline{e1} - \{\backslash e\} \wedge e1 = \backslash e1 \tag{29}$$

We can treat $\underline{e1}$ and $\backslash \underline{e1}$ as auxiliary group-variables provided we are ready to reduce such use to the more basic definitions. For example, let $r.\sigma = \{e1, e2, e3, t\}$ and $\underline{e1} = \{e1, e2\}$. Then the definition of $\underline{e1} = \backslash \underline{e1}$ is as follows:

$$\underline{e1} = \backslash \underline{e1} \quad \widehat{=} \quad e1 = e2 = \backslash e1 = \backslash e2 \wedge e1 \neq e3 \tag{30}$$

As an example of the creation definition, consider

{precondition : $\underline{e2}[e1] \wedge \underline{e3}[]$}
create $e1$
{postcondition : $\underline{e1}[] \wedge \underline{e2}[] \wedge \underline{e3}[]$}

The correctness of the above can be checked directly as follows. Assume $\backslash \underline{e2} = \{\backslash e1, \backslash e2\} \wedge \backslash \underline{e3} = \{\backslash e3\}$ as required by the precondition. Then

create $e1$
$=$ ⟨ entity creation Definition 4.1 ⟩
$\underline{e1} = \{e1\} \wedge \Delta(e1)$
$=$ ⟨ $\underline{e1} = \{e1\} \rightarrow e1 \notin \underline{e2}$, assumption $\backslash \underline{e2} = \{\backslash e1, \backslash e2\}$ and (27) ⟩
$\underline{e1} = \{e1\} \wedge \Delta(e1) \wedge \underline{e2} = \{\backslash e2\}$

$$
\begin{array}{ll}
= & \quad \langle \ \Delta(e1) \rightarrow e2 = \backslash e2 \ \rangle \\
& \underline{e1} = \{e1\} \wedge \Delta(e1) \wedge \underline{e2} = \{e2\} \\
\Rightarrow & \quad \langle \ \text{assumption } \backslash\underline{e3} = \{\backslash e3\} \text{ and (28)} \ ; \ \Delta(e1) \rightarrow e3 = \backslash e3 \ \rangle \\
& \underline{e1} = \{e1\} \wedge \underline{e2} = \{e2\} \wedge \underline{e3} = \{e3\} \\
= & \quad \langle \ \text{definition} \ \rangle \\
& \{\underline{e1}[] \wedge \underline{e2}[] \wedge \underline{e3}[]\}
\end{array}
$$

Each of the above proof steps is trivial, so that in the following we feel free to collapse the above proof into one step with justification of Definition 4.1.

The type-compatible reference assignment statement $e1 := e2$ changes entity $e1$ to refer to the same object as entity $e2$. Definition 4.2 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 4.2 (Reference Assignment Semantics). Suppose $e1$ and $e2$ are references and their declared types are compatible according to Meyer [Mey97], so that $e2$ can be assigned to $e1$. Then

$e1 := e2$

is defined as

modifies $e1$

ensure $e1 =$ **old** $e2$

The following theorems can be inferred from the group axioms and can be used to simplify calculations involving reference assignments; note that the occurrences of $e1 = e2$ in the theorems are reference equalities, and not assignment statements. Once again, we assume that $e1$ and $e2$ are distinct names, and one or both may be replaced by dot expressions.

$$e1 = e2 \wedge \Delta(e1) \wedge (\backslash e1, \backslash e2 \in \backslash\underline{e3}) \ \rightarrow \ \underline{e3} = \backslash\underline{e3} \tag{31}$$

$$e1 = e2 \wedge \Delta(e1) \wedge \backslash\underline{e3} = \{\backslash e1, \backslash e3\} \ \rightarrow \ \underline{e3} = \{e3\} \tag{32}$$

$$e1 = e2 \wedge \Delta(e1) \wedge \backslash e2 = \mathit{Void} \ \rightarrow \ e1 = e2 = \mathit{Void} \tag{33}$$

$$e1 = e2 \wedge \Delta(e1) \wedge \backslash\underline{e2} = \{\backslash e2, \backslash e3\} \ \rightarrow \ \underline{e2} = \{e1, e2, e3\} \tag{34}$$

We now present the meaning of query and command calls, focusing on how these calls effect changes in the state of objects and in entity groups. Definition 4.3 provides the semantics of query calls where the result of a query is assigned to an entity.

Definition 4.3 (Assigned Query Call). The query call

$e1 := e2.q(e3)$

for query q of *SUPPLIER* (in Fig. 4) means

modifies $e1$

require $e2 \neq \mathit{Void}$

ensure $e1 = \backslash(e2.q(e3))$

Since it is possible for $e1$ to be $e2$ itself, we must use $\backslash$ in the postcondition when referring to $e2$. Note that the precondition of query q is inferred through use of (36) discussed below. We can recursively treat queries of arbitrary complexity using (35)

$$e1 := e2.q(\mathit{exp}) \ \ \widehat{=} \ \ (\exists\, e \bullet \ e := \mathit{exp};\ e1 := e2.q(e)) \tag{35}$$

where e is a fresh entity of the appropriate type and *exp* is itself a query.

Referring to Fig. 4, a query call $e2.q(e3)$ appearing in a contract can be evaluated using the following:

$$q.\mathit{pre}[\alpha] \wedge e2 \neq \mathit{Void} \rightarrow q.\mathit{post}[\alpha] \tag{36}$$

$$\alpha \ \ \widehat{=} \ \ a := e2.a,\ \mathit{Current} := e2,\ x2 := e3,\ \mathit{Result} := e2.q(e3) \tag{37}$$

```
class C feature
        a : INTEGER
        q : INTEGER
                modifies Result
                ensure Result = a + 2
        c is
                modifies a
                require q > a
                ensure a = old a + 10
end
```

Fig. 6. Class C

where α is a substitution. In other words, we may use the targeted contract of q to evaluate $e2.q(e3)$, and thereafter we can use (36) as an assumption in reasoning; such an assumption can be introduced in a proof step, or as an antecedent in an implication. Since queries are required to be side-effect free, we also have

$$\backslash q.pre[\alpha] \wedge \backslash e2 \neq Void \rightarrow \backslash q.post[\alpha] \tag{38}$$

Because we are treating (36) as an assumption, and we are not replacing occurrences of $e2.q(e3)$ with (36), we can treat arbitrary query specifications, i.e., we do not have to require that queries are functional.

Definition 4.4 provides the meaning of a targeted command call $e1.c(e2)$, where $e1$ and $e2$ are reference entities and c is a command of the class *SUPPLIER* (see Fig. 4). The meaning of this call is supplied by the precondition and postcondition of c, targeted to the entities $e1$ and $e2$.

Definition 4.4 (Targeted Command Call). The call

$e1.c(e2)$

for command $c(x1 : TYPE)$ in *SUPPLIER* having attribute a (in Fig. 4) means

modifies $c.modifies[a := e1.a, x1.a := e2.a]$
require $e1 \neq Void$;
$\quad c.pre[x1 := e2, a := e1.a, Current := e1]$
ensure $c.post[\backslash a := \backslash e1.a, \backslash Current := \backslash e1,$
$\quad x1 := e2, a := e1.a, Current := e1]$

(35) can be used to treat complex commands of the form $e1.c(exp)$.

The **modifies** clause states that the command can change the attribute $e1.a$. The **require** clause establishes that any call must be to a non-*Void* entity. The clause also includes the precondition of c, targeted to entity $e1$. The **ensure** clause is the postcondition of c, targeted to $e1$ and with $e2$ as the argument.

As a simple example illustrating the use of feature calls, consider the class C in Fig. 6, and assume that we would like to show that

$$\{e \neq Void\}\ y := e.q;\ e.c;\ z := e.q\ \{z = y + 10\}$$

Assume $\backslash e \neq Void$. Then

$y := e.q;\ e.c;\ z := e.q$
$=$ $\langle$ Definition 4.3 $\rangle$
$\backslash e \neq Void \rightarrow y = \backslash e.q \wedge \Delta(y)\ ;\ e.c;\ z := e.q$
$\Rightarrow$ $\langle$ (38) applied to $e.q$: $\backslash e \neq Void \rightarrow \backslash e.q = \backslash e.a + 2$ $\rangle$
$\backslash e \neq Void \rightarrow y = \backslash e.a + 2 \wedge \Delta(y);\ e.c;\ z := e.q$
$\Rightarrow$ $\langle$ assumption; $\Delta(y) \rightarrow e.a = \backslash e.a \wedge e = \backslash e$ $\rangle$
$y = e.a + 2 \wedge e \neq Void;\ e.c;\ z := e.q$
$\Rightarrow$ $\langle$ Definition. 4.4 $\rangle$
$y = e.a + 2 \wedge e \neq Void;\ \backslash e \neq Void \wedge \backslash e.q > \backslash e.a \rightarrow e.a = \backslash e.a + 10 \wedge \Delta(e.a)\ ;\ z := e.q$
$\Rightarrow$ $\langle$ (38) applied to $e.q$: $\backslash e \neq Void \rightarrow \backslash e.q = \backslash e.a + 2$ $\rangle$
$y = e.a + 2 \wedge e \neq Void;\ \backslash e \neq Void \wedge \backslash e.a + 2 > \backslash e.a \rightarrow e.a = \backslash e.a + 10 \wedge \Delta(e.a);\ z := e.q$
$=$ $\langle$ $\backslash e.a + 2 > \backslash e.a \equiv true$ $\rangle$
$y = e.a + 2 \wedge e \neq Void;\ \backslash e \neq Void \rightarrow e.a = \backslash e.a + 10 \wedge \Delta(e.a);\ z := e.q$

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

Fig. 7. Hybrid assignments

$\Rightarrow$ ⟨ definition of sequential composition (4) and one point rule; $\Delta(e.a) \rightarrow y = \backslash y$ ⟩
$e \neq Void \wedge e.a = y + 8;\ z := e.q$
$\Rightarrow$ ⟨ same reasoning pattern as before using (38) ⟩
$z = y + 10$

The appendix contains a simple verification example using groups and reference types, as well as an example involving a loop.

4.2. Expanded types

If entities $e1$ and $e2$ are both expanded (i.e., they are either a basic value such as integer or boolean, or refer to a subobject), then the standard rule for assignment in (4) suffices. If both are references, then the theory developed in Section 4.1 can be used. However, we have not yet discussed the effect of assigning an expanded object to a reference, and vice versa. The table in Fig. 7 suggests how to extend our approach to handle them, leaving a full treatment (e.g., expanded parameters) for later work.

5. Class compositionality

Refinement can be applied in a modular way to large software systems composed from classes by inheritance and client–supplier relationships. A specification (i.e., contract) of a class A, that depends upon various other classes, can be refined to code just by appealing to the specifications of the other classes, and without the need to know the class implementations. To illustrate this process, we examine, without loss of generality, OO system shown in Fig. 8.

We have used a small BON diagram to depict this system. While we could, without difficulty, use Eiffel text for the same purpose, we use BON here to illustrate how one might desire to apply the calculus in practice to large systems. Frequently in large-scale OO development, a class diagram of some kind will be drawn, from which code will be produced. It is necessary for our refinement techniques to be linked to such diagrams, so that they can be used together in the large.

Suppose that we want to refine class A to code, where A depends directly on class B and class C, and indirectly on class D (via association with class C). In order to refine features of A to code, we need to know which features of classes that A is related to can be used in the refinement process. A includes features inherited from B and all of B's ancestors. A may make direct use of features of C that are exported to it. It may also make *indirect* use of features of D through C, as follows. Suppose that A has a feature f. Due to the relationship between A and C, A also has a feature $c : C$; the relationship between C and D means that C has a feature $d : D$. If feature d of class C is exported to A, and feature foo of class D is also exported to A, then the following is an acceptable precondition for f:

$$c.d.foo \geqslant 0 \tag{39}$$

Thus, A can make use of features of D through C's subobject, even though there is no direct relationship drawn in Fig. 8 between A and D. It is also possible that $c.d.foo$ will enter the refinement even less directly. For example, the contract of c may contain a reference to d, and the contract of d might in turn contain a reference to $d.foo$. So we need to define a transitive closure of the inheritance, association, and aggregation relationships, which takes into account information hiding, in order to determine the features needed to carry out refinement correctly.

The precise definition of the *short-flat* form of a class is given in [Mey92]. Briefly, the *short* form is the interface of the class consisting of all exported features and their contracts (but not their implementations). The flat form of

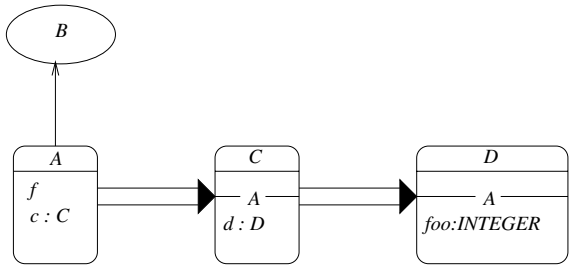

Fig. 8. System structure for demonstrating the modular refinement process

the class includes all the features obtained from proper ancestors, putting them at the same level as the immediate features of the class (taking into account both renaming and redefinition). If C is a class, then we let $\bar{C}$ denote the short-flat form of C. We define **depend** A recursively as follows:[14]

- $\bar{A} \in$ **depend** A.
- If $\bar{X} \in$ **depend** A and the arguments, contracts, or return values of a feature of $\bar{X}$ has an association or aggregation with class Y, then $\bar{Y} \in$ **depend** A.
- Nothing else is in **depend** A.

In the case of Fig. 8, **depend** $A = \{\bar{A}, \bar{C}, \bar{D}\}$.

To refine A to an implementation, we must carry out the following steps, in order:

1. Determine **depend** A.
2. Determine **spec** A, the set of contracts for all routines in $\bar{A}$ that need refining. Only contracts that are newly declared in A, as well as contracts that redefine ones inherited from ancestors, are included in **spec** A.
3. Show that each element of **spec** A is implementable. Alternatively, refining each contract in **spec** A to an Eiffel program itself demonstrates implementability.
4. To refine class A to a program, refine each specification s of **spec** A by refinement steps to **code** s, an Eiffel implementation. In other words, it must be shown that

   $\forall s \in$ **spec** $A \bullet s \sqsubseteq$ **code** s

   In the refinement, we need only use the contracts (not the implementations) of classes in **depend** A.

The important thing to note in this process is that to refine A to code, we *only* need **spec** A and the contracts of features belonging to **depend** A. No implementations of the features of **depend** A are needed; in this sense, refinement is compositional. In most cases the entire system need not be considered when refining a class, since **depend** A will only involve the contracts of a subset of all classes in the system. Thus, refinement can be done class by class, and thereafter feature by feature.

To refine the complete system, consisting of a number of classes, to a program, we must start from the root class. The root is refined to code, using the contracts of the classes in **depend** $ROOT$. Then each class that the root depends upon (i.e., all elements of **depend** $ROOT$) must in turn be refined to code, using only the contracts of the classes it depends upon. This process recursively continues until all classes in the system have been refined to programs. In this process there is no system-level validity check that has to be discharged to show that the entire system is correct. Once all classes have been refined, then the system is implemented and a proof has been discharged to show its correctness.

The efficiency and validity of this process hinges on using *directed* OO relationships, and information hiding. In particular, information hiding as defined in Eiffel requires that only the procedures of a class can effect changes in the state of objects of that class; clients of a class cannot change state directly via assignment to attributes of an object. In other words, the attributes of a class are read-only to clients. If this level of information hiding [Par72b] was not used in Eiffel, then clients could change attributes of a class, and hence **depend** A would possibly need to contain all classes in the system. Similarly, if undirected relationships, as present in modelling languages

[14] This is similar to the notion of 'cone of influence' in compiler technologies.

like UML [RJ99] were used, then it would not, in general, be possible to refine a class to code because no class would have been given responsibility for the relationship – that is, neither class would be responsible for providing an attribute or a routine to represent the relationship. Thus, refinement would have to be carried out after all undirected relationships in OO specification had been replaced by directed ones.

This style of modular reasoning is also inherent in Object-Z [Smi00]; in fact, the semantics of Object-Z has been defined precisely for this kind of so-called modular reasoning. We discuss this more in Section 7.

6. Automation

Discharge of the proof obligations that arise during a refinement can benefit from automated support. We are currently experimenting with using PVS to reason about Eiffel specifications. Our eventual goal is to use PVS to discharge the (automatically generated) proof obligations that arise in a refinement of a specification to an Eiffel program.

Currently, we have developed a mapping of Eiffel specifications into PVS theories, based on a representation of a class as a datatype, a representation of routines as PVS functions, and a representation of class invariants as either subtype constraints or axioms (the latter being used whenever invariant clauses on self-referential classes occur). This representation can be extended to support client–supplier and inheritance relationships, and has allowed non-trivial theorems to be discharged. A particular benefit of translating Eiffel to PVS is that it lets us use the tool to deal with reference types without difficulty. The theory of reference types has been coded in PVS; the theory was presented in [PO03]. A new tool is being designed and implemented to automate the translation process. Thereafter, a tool will be developed to assist in the refinement process. The tool will pass proof obligations generated in refinement to PVS. This is discussed more elsewhere [PO99].

For additional examples of using PVS to reason about OO models, we refer the reader to [PO01], wherein the metamodel of BON is expressed in the PVS language, and it is shown how to use the PVS system to prove that BON models conform to (or fail to conform to) the BON metamodel.

7. Related work and conclusions

In this paper we have provided rules for refinement of OO specifications in Eiffel into immediately executable Eiffel programs. The rules include ones for introducing feature calls and object creation statements during refinement. The refinement rules that we have presented are modular, and can be applied partwise over OO models consisting of a number of classes, where each class contains a number of features with contracts. Thus, the refinement process, combined with OO structuring mechanisms – i.e., client–supplier relationships and inheritance relationships – is applicable to large-scale systems.

Refinement to OO industrial-strength languages has not been treated much in the literature, although several approaches, such as JML, Larch, and Object-Z provide suitable foundations for such refinement calculi. There is, however, a rich collection of work on specification, verification, and static checking of OO software. We now discuss this work.

Object-Z is a specification language containing no immediately executable industrial-strength programming language. The semantics of Object-Z has evolved over a number of years, and has seen many distinct revisions. A value semantics was initially supported, and this was later extended to a reference semantics [Smi95]; the current semantics in [Smi00] supports references and modular reasoning. The semantics of Object-Z in [Gri97], based on labelled transition systems, also supported *strict modular* reasoning: the meaning of an operation in an Object-Z specification is a transition on the local state of an object, together with an external message. Modular reasoning is thus supported by the semantics, which provides object identities and mechanisms for achieving independence of behaviour of operations. This kind of semantics is useful for reasoning about the properties of an OO system as a whole [Smi95], but may not be as convenient for algorithm refinement, and in particular producing executable code from specifications.

Object-Z reference types differ from Eiffel — aliasing is supported, but you cannot equate an entity to *Void*. So while aliasing, referencing, and dereferencing of objects is permitted in Object-Z, it is still not entirely clear how to transform these specifications into programs in typical OO languages such as Eiffel. Refinement of value semantics to reference semantics is treated in [Smi02]. Abstract specifications are written in a value semantics in order to facilitate reasoning, and then are refined to a concrete specification in reference semantics in order to facilitate transformation to code.

Müller's recent thesis [Mul01] presents a modular verification approach for OO systems, based on a Hoare logic. The approach has many notational similarities to our own work. The specification language used has similarities to JML, and makes use of constrained invariants and read-only references for helping to solve the frame problem and aliasing problems. A key difference between his work and ours – and work on the Extended Static Checker, discussed in the next paragraph – is that we do not deal with information hiding in ERC, whereas Müller's work does. In ERC, contracts can make use of any attributes and routines, including private ones, thus exposing potential implementation details to clients. There are restrictions in Eiffel on the appearance of private features in **require** clauses (though there are no restrictions on what appears in an **ensure** clause). These restrictions are not as of yet captured in ERC, though it should not be difficult to extend the calculus to do so.

The work on the Extended Static Checker for Java [LNS00] has resulted in the development of a tool for the automatic verification of Java programs. This tool works by taking annotated Java programs, with specifications very similar to the contracts used in this paper, and checking that the programs satisfy the annotations. This approach focuses on automatic verification of programs, rather than refinement of specifications to programs. A great advantage of this work is that it is already implemented as a tool and (with annotation) can work on regular Java programs. Although it works above the 'decidability ceiling' (and can thus catch errors that regular typecheckers cannot), it focuses on catching the kinds of errors in programs that can be automatically detected, e.g., *void* reference accesses. Hence it does not catch all possible errors. It handles reference types as well as primitive types. The Perfect Developer tool [Esc00] is also used for static checking and automatic verification. With this tool, a new language, Perfect, must be used for writing specifications, but target code in a number of different programming languages (e.g., C++) can be automatically generated.

We view such static checking techniques as complementary to the refinement calculus in this paper: in some cases we may want to refine contracts or classes to programs; in other cases we may prefer to use static checking to help find errors in programs. The latter approach is more appropriate to use when verifying library classes or pre-existing applications.

Work has been carried out on validating UML OO models against constraints, via simulation. The work of Richters and Gogolla on the USE tool [RG00] is a particular example of this. In the USE tool, a UML model is imported, a snapshot (an instance) of the model is taken, and the snapshot is checked against OCL constraints that are evaluated. A subset of UML and OCL is supported by the USE tool. Validation of OO models against metamodel constraints via automated theorem proving is demonstrated by Paige and Ostroff using PVS [PO01].

Cavalcanti and Naumann present a weakest-precondition-based refinement calculus for OO language, ROOL, which has similarities to Java [CN00]. The calculus supports mutually recursive classes and dynamic binding; its semantics is based on a typing system. Work is continuing on extending the calculus to reference types.

VDM++ is OO dialect of VDM. It is based on a three-valued logic. Lano [Lan95] presents data and algorithm refinement rules for VDM++, but these rules focus on refinement of the imperative and concurrent constructs, and do not present mechanisms for introducing command or query calls. Furthermore, the results of the refinement require further translation to produce executable code in C++, Java, Eiffel, etc. Lano [Lan95] also presents informal procedures for carrying out these translations.

An approach to OO development similar to Eiffel is Larch/C++ [Lea97], which aims at supporting formal specification, as well as reducing the gap between specification and working code. A key distinction between Larch/C++ and Eiffel is that with Larch, a two-tiered approach is used. Specifications of mathematical toolkit features (e.g., library modules such as arrays, lists, and function types) are provided algebraically using abstract data types. These specifications can then be used in Larch/C++ behavioural interface specifications, wherein the abstract data type functions can appear in preconditions, postconditions, and invariant clauses. By keeping the abstract data type specifications separate from behavioural interface specifications, formal reasoning on the shared language specifications can be carried out, and the formal specifications can be reused specifying for different behavioural interface languages. Eiffel uses only OO techniques: in place of the Larch Shared Language specifications, only classes with contracts are used instead, including for the specification of mathematical toolkit features. By using only classes, software development can proceed seamlessly (and if necessary reversibly) within the same semantic framework. Larch/C++ does not support reversibility. Further, Larch/C++ does not have rules for refinement, though they could in principle be developed. We would expect these rules to be more complex than those for Eiffel because C++ is a hybrid language having both OO and conventional constructs. Also, implementing a Larch/C++ specification will require the Larch Shared Language specifications to be implemented, perhaps using built-in libraries; an impedence mismatch between abstract data types and C++ classes arises here. Larch/C++ does possess mature tool support for formal manipulation and reasoning. Reasoning will typically be done within the algebraic framework, using functions of the algebraic specifications. With Eiffel, reasoning is done using first-order logic.

JML [LB00] is a modelling language for Java, with many similarities to Larch/C++. It supports the specification of contracts in a pre- and postcondition style; class invariants can also be specified. JML has been designed to work seamlessly with Java. Unlike BON, it is a text-based modelling language and is syntactically similar to Java. JML is a richer language than BON, in that it supports history constraints and modelling exceptions. It has no refinement rules defined, although such rules are feasible to produce. Work is underway on integrating JML with the Extended Static Checker for Java [LNS00]. Work is also ongoing on providing theorem proving support for JML and Java, in order to verify Java programs. The LOOP project [BJ01] is developing theories for PVS and Isabelle that will allow automated reasoning about JML specifications and Java programs. Tools are also being produced as a part of this project to generate PVS specifications automatically from Java and JML.

Meyer [Mey03] introduces a technique for proving the correctness of classes based on abstract data types. In his approach, model fields can be introduced and an underlying state representation based on functions is provided. A semantics of program constructs is then provided. The formalization appears to be compatible with supporting tools such as the B-Toolkit. By contrast, ERC aims at supporting refinement (instead of verification) and does not require the introduction of model fields. Moreover, the state representation is based on entity groups.

This paper did not deal completely with expanded types nor did it show how to automate the refinement procedures. The calculus also does not consider information hiding issues. Automation and the extension to expanded types are currently under investigation; Section 7 reported on the current status of some of this work. We also intend to expand the framework to concurrent and real-time software. The use of the predicative calculus of Hehner [Heh93], which supports concurrency and communication, as the underlying specification formalism should make the extension to concurrency feasible; the calculus already supports real-time specification and refinement, but additional examples are needed in order to demonstrate the effectiveness of the calculus in this domain.

Acknowledgements

Thanks to the referees for their extremely detailed, thorough comments, patience, and excellent ideas. They surpassed the call of duty with their reviews.

Appendix

A. An example using references and entity groups

Figure 9 contains a short example demonstrating how to reason with the simple theory of reference types, presented in Section 4. We assume that we have a class *CELL* and a class *C* with interfaces and implementations as shown. We want to prove that the body of feature *r* of class *C* establishes its postcondition $e3 = 6$. We do this by replacing each statement in the implementation of *r* with its meaning in terms of specification statements, and then simplify. Each proof step involves many smaller steps. For example, in the first step, we have

$$
\begin{array}{ll}
 & \textbf{create}\ e1;\ \textbf{create}\ e2 \\
\Rightarrow & \quad \langle\ \text{entity creation Definition 4.1 and (29)}\ \rangle \\
 & \underline{e1} = \{e1\};\ \underline{e2} = \{e2\} \wedge \underline{e1} = \backslash\underline{e1} - \{\backslash e2\} \wedge e1 = \backslash e1 \\
= & \quad \langle\ \text{sequential composition (4) and treating } \underline{e1} \text{ and } \underline{e2} \text{ as group variables}\ \rangle \\
 & \exists\, \hat{\underline{e1}}, \hat{e1}, \hat{e2} \bullet \hat{\underline{e1}} = \{\hat{e1}\} \wedge \underline{e2} = \{e2\} \wedge \underline{e1} = \hat{\underline{e1}} - \{\hat{e2}\} \wedge e1 = \hat{e1} \\
\Rightarrow & \quad \langle\ \text{Leibniz axiom using } \hat{\underline{e1}} = \{\hat{e1}\}\ \rangle \\
 & \exists\, \hat{\underline{e1}}, \hat{e1}, \hat{e2} \bullet \underline{e2} = \{e2\} \wedge \underline{e1} = \{\hat{e1}\} \wedge e1 = \hat{e1} \\
= & \quad \langle\ \text{one-point rule using } e1 = \hat{e1}\ \rangle \\
 & \underline{e1}[] \wedge \underline{e2}[]
\end{array}
$$

where we ignored *time*, using (7) as our justification. The manipulation using group-variables can be verified by reducing the expressions involving the group-variables to their basic definitions as in (30). Using Definition 4.4, the targeted call $e1.m(5)$ reduces to $e1 \neq \mathit{Void} \rightarrow e1.a = 5 \wedge \mathit{same}(\underline{e1}, \underline{e2}) \wedge \mathit{time}$, so that

$$
\begin{array}{ll}
 & \underline{e1}[] \wedge \underline{e2}[];\ e1.m(5) \\
\Rightarrow & \quad \langle\ \text{Definition 4.4; } \underline{e1}[] \rightarrow e1 \neq \mathit{Void} \text{ using (19); sequential composition (3)}\ \rangle \\
 & \underline{e1}[] \wedge \underline{e2}[] \wedge e1.a = 5
\end{array}
$$

The rest of the proof in Fig. 9 continues along the same lines.

```
class C feature
    e1, e2 : CELL; e3 : INTEGER
    r is do
        create e1;
        create e2;
        e1.m(5);
        e2.m(6);
        e1.n(e2);
        e3 := e1.a.max(e1.b.a);
    ensure e3 = 6
    end
end
```

```
class CELL feature
    a : INTEGER; b : CELL

    n(x : CELL) is
    modifies b
    require x ≠ Void
    ensure b = x

    m(x : INTEGER)
    modifies a
    ensure a = x
end
```

Body of routine r in class C

$\rightarrow$ ⟨**create** Definition 4.1; composition (4); Definition 4.4; (24); (6) and logic⟩
$e1.a = 5 \wedge e2.a = 6 \wedge \underline{e1}[] \wedge \underline{e2}[];$
$e1.n(e2);$
$e3 := e1.a.max(e1.b.a)$

$=$ ⟨ *semantics of* $e1.n(e2)$ using Definition 4.4⟩
$e1.a = 5 \wedge e2.a = 6 \wedge \underline{e1}[] \wedge \underline{e2}[];$
$e1.b, t : (\!| e1 \neq Void, e1.b = e2 |\!);$
$e3 := e1.a.max(e1.b.a)$

$\rightarrow$ ⟨ $\underline{e1}[] \rightarrow e1 \neq Void$ and (6) ⟩
$\underline{e1}[] \wedge \underline{e2}[] \wedge e1.a = 5 \wedge e1.b = e2 \wedge e2.a = 6;$
$e3 := e1.a.max(e1.b.a)$

$\rightarrow$ ⟨assignment (4) , entities are expanded⟩
$\underline{e1}[] \wedge \underline{e2}[] \wedge e1.a = 5 \wedge e1.b = e2 \wedge e2.a = 6;$
$e3 =$ **old** $(e1.a.max(e1.b.a)) \wedge same(e1.a, e2.a, e1.b.a)$
⟨ composition (4) ; one-point; (19): $e1.b = e2 \rightarrow e1.b.a = e2.a$⟩
$\underline{e1}[] \wedge \underline{e2}[] \wedge e1.a = 5 \wedge e1.b = e2 \wedge e2.a = 6 \wedge$
$e3 = 5.max(6)$

$\rightarrow$ ⟨(24) and postcondition of *max* in Fig. 9⟩
ensure clause of r

Fig. 9. Example of reasoning using reference types

B. A refinement example involving a loop

This section illustrates a simple example of refinement in Eiffel. The problem we solve is a simple one, taken from [Wor94], to find the maximum of a non-empty array s of integers. Though simple, this problem in fact illustrates the main feature call interactions that arise in OO refinement, including the subtle case in which the target of an assignment invokes a query call on the target itself. We suppose that we have a class, *FOO*, that includes a feature that will be used to determine the maximum of the array. We provide the Eiffel specification for the class *FOO* in Fig. 10.

We use $\uparrow$ in the specification to represent the *mathematical* operator that gives the maximum of two or more integers. We use the generalized quantifier notation of Gries and Schneider [GS93] to take the maximum of a set of integers; the postcondition of the *max_array* routine demonstrates this syntax. We use several logical laws for reasoning taken from [GS93] as well.

```
class FOO
feature
        s : ARRAY[INTEGER]

        max_array : INTEGER
                -- calculate maximum of array s
                -- s.good  ≙  s ≠ Void ∧ ¬ s.empty
        modifies Result
        require s.good
        ensure Result = (↑ j : INTEGER | s.valid_index(j) • s.item(j))
end
```

Fig. 10. The class *FOO* for refinement example

```
class ARRAY[G] feature
  lower, upper : INTEGER
  count, capacity : INTEGER

  valid_index(x : INTEGER) : BOOLEAN
   ensure Result = (lower ⩽ x ⩽ upper)

  item(x : INTEGER) : G
   require valid_index(x)

  empty : BOOLEAN
   ensure Result = (count = 0)

   invariant lower ⩽ upper ∧ count ⩾ 0
end
```

```
class INTEGER feature
  max(x : INTEGER) : INTEGER
   ensure Result = Current ↑ x
end
```

Fig. 11. Excerpt from interfaces of the *ARRAY* and *INTEGER* classes

The class *FOO* has one association with the class *ARRAY* and an aggregation with the class *INTEGER*, via the return value of query *max_array*, which calculates the maximum of the array *s*. A local variable *Result*, automatically declared for the query, will hold the result of the computation. We make use of the following features of the classes *ARRAY* and *INTEGER*, shown in Fig. 11; for the sake of completeness, we present fragments of the specification of each class.

The notation *s.item*(*j*) is the Eiffel syntax for the array index operation. *s.lower* and *s.upper* are the lower and upper bounds of the array *s*, respectively. Note that *item*, a query of *ARRAY*, has no postcondition; in this sense, we can view *item* as an atomic specification unit, one whose meaning is not denoted by any other, perhaps more concrete, representation.

We can now refine the specification of *FOO* to code. In the process we use the contracts, but not the implementations, of the classes on which *FOO* depends. The process of Section 5 starts by calculating the classes on which *FOO* depends. Thus

depend $FOO = \{FOO, INTEGER, ARRAY[INTEGER], BOOLEAN\}$

Then we calculate **spec** *FOO* which consists of the contract for *max_array*. Now, we must refine each element of **spec** *FOO*, i.e., **spec** *max_array*, using the contracts of features in **depend** *FOO* that are accessible to *FOO* (in this example we have shown only public features of *INTEGER* and $ARRAY[G]$ that will be needed in the refinement; in general, all public features can be used in the refinement of a client or descendant class).

Figure 12 provides a refinement from the specification of *max_array* to executable Eiffel code. As argued at the end of Section 3.3 and in (6), we ignore *time*. In Fig. 12 we define several terms: P (defined in step (b)) is the specification for the initialization of the loop; I and V are a loop invariant and variant, respectively; and W is the specification for the body of the loop.

Step (a) in Fig. 12 uses the local variable introduction rule to introduce a loop index. For step (b) we must discharge all the provisos in the loop rule. Some of the provisos are simple, e.g., the loop proviso $S \rightarrow S_{init}$ reduces to $s \neq Void \rightarrow s \neq Void$ which is trivially true. The other provisos are relatively straightforward. As an example, the sixth proviso can be discharged as follows:

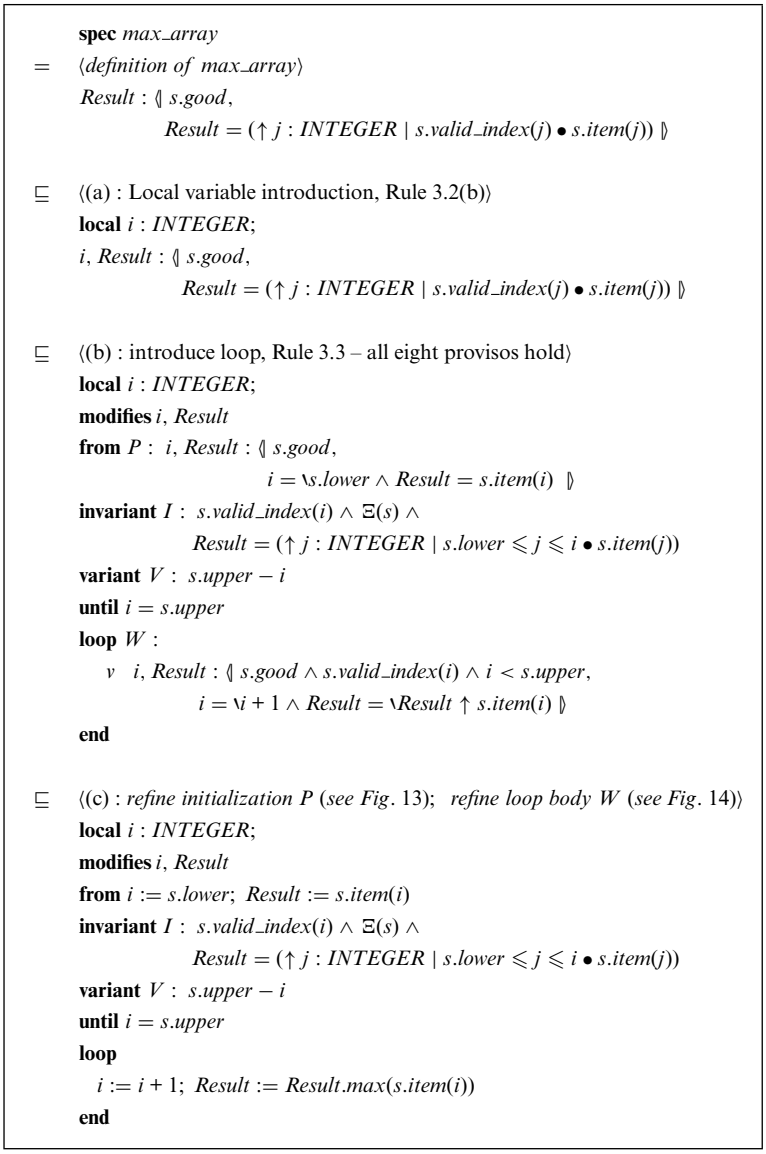

Fig. 12. Refinement tree for *max_array* (the last step produces executable Eiffel code)

Assume $I \wedge \neg\, b \wedge \backslash S$. Then:

$(D_{body} \wedge same(\bar{\sigma}))'$

$=$ ⟨ definition ⟩

$i' = i + 1 \wedge s' = s \wedge (Result' = Result \uparrow s'.item(i'))$

$=$ ⟨ Leibniz and $i' = i + 1 \wedge s' = s$ ⟩

$i' = i + 1 \wedge s' = s \wedge (Result' = Result \uparrow s.item(i + 1))$

$=$ ⟨ from the invariant, $Result = (\uparrow j : INTEGER \mid s.lower \leqslant j \leqslant i \bullet s.item(j))$; split off term (8.23) of [GS93] ⟩

$i' = i + 1 \wedge s' = s \wedge Result' = (\uparrow j : INTEGER \mid s.lower \leqslant j \leqslant i + 1 \bullet s.item(j))$

$=$ ⟨ Leibniz and $i' = i + 1 \wedge s' = s$ ⟩

$i' = i + 1 \wedge s' = s \wedge Result' = (\uparrow j : INTEGER \mid s'.lower \leqslant j \leqslant i' \bullet s'.item(j))$

$=$ ⟨ from invariant $\Xi(s)$ ⟩

$i' = i + 1 \wedge s' = s \wedge s' = \backslash s \wedge Result' = (\uparrow j : INTEGER \mid s'.lower \leqslant j \leqslant i' \bullet s'.item(j))$

$$
\begin{array}{ll}
 & P \\
= & \quad \langle \text{ definition } \rangle \\
 & i, Result : (\!| s.good,\ i = \backslash s.lower \wedge Result = s.item(i) |\!) \\
= & \quad \langle \text{ textual substitution [GS93] } \rangle \\
 & i, Result : (\!| s.good[i := s.lower],\ (i = \backslash i \wedge Result = s.item(i))[\backslash i := \backslash s.lower] |\!) \\
= & \quad \langle \text{ simple substitution, Rule 3.1 } \rangle \\
 & i := s.lower;\ Result : (\!| s.good,\ Result = s.item(i) |\!) \\
\sqsubseteq & \quad \langle \text{ (36): } s.good \rightarrow s \neq Void \wedge s.valid\_index(s.lower); \text{ pre. weakening } \rangle \\
 & i := s.lower;\ Result : (\!| s \neq Void \wedge s.valid\_index(s.lower),\ Result = s.item(i) |\!) \\
\sqsubseteq & \quad \langle \text{ assigned query call Definition 4.3 } \rangle \\
 & i := s.lower;\ Result = s.item(i)
\end{array}
$$

Fig. 13. Proof of refinement for loop initialization

$$
\begin{array}{ll}
 & W \\
\sqsubseteq & \langle \text{textual substitution [GS93]} \rangle \\
 & i, Result : (\!| (s.good \wedge s.valid\_index(i-1) \wedge i-1 < s.upper)[i := i+1], \\
 & \qquad (i = \backslash i \wedge Result = \backslash Result \uparrow s.item(i))[\backslash i := \backslash i + 1] |\!) \\
\sqsubseteq & \langle \text{simple substitution, Rule 3.1} \rangle \\
 & i := i + 1; \\
 & Result : (\!| s.good \wedge s.valid\_index(i-1) \wedge i - 1 < s.upper, \\
 & \qquad i = \backslash i \wedge Result = \backslash Result \uparrow s.item(i) |\!) \\
\sqsubseteq & \langle s.good \wedge s.valid\_index(i-1) \wedge i - 1 < s.upper \rightarrow s \neq Void \wedge s.valid\_index(i); \\
 & \quad \langle \text{precondition weakening (Morgan Rule 1.2 [Mor94])} \rangle \\
 & Result : (\!| s \neq Void \wedge s.valid\_index(i), \\
 & \qquad Result = \backslash Result \uparrow s.item(i) |\!) \\
\sqsubseteq & \langle \text{proof described in Fig. 15} \rangle \\
 & Result := Result.max(s.item(i))
\end{array}
$$

Fig. 14. Refinement tree for the loop body

$$
\begin{array}{ll}
\Rightarrow & \quad \langle \text{ From invariant, } s.valid\_index(i);\ \neg\, b \ \widehat{=}\ i \neq s.upper;\ i' = i + 1 \wedge s' = s \ \rangle \\
 & s'.valid\_index(i') \wedge s' = \backslash s \wedge Result' = (\uparrow j : INTEGER \mid s'.lower \leqslant j \leqslant i' \bullet s'.item(j)) \\
= & \quad \langle \text{ definition } \rangle \\
 & I[\_ := \_']
\end{array}
$$

We next present the refinement steps and corresponding proofs that are required in implementing the loop initialization and loop body. The first step is the refinement of the loop initialization, P, by the sequential composition

$$i := s.lower;\ Result := s.item(s.lower)$$

and is shown in Fig. 13.

The final step in the refinement example of Section 6 is to implement the loop body specification, W, by an assignment statement (which is a simple increment) and a command call. Figure 14 shows the tree for the refinement of W.

Most of the proof obligations that arise from this refinement tree are straightforward to discharge. In the penultimate step of Fig. 14, the justification is proved as follows:

$$
\begin{array}{ll}
 & s.good \wedge s.valid\_index(i-1) \wedge i - 1 < s.upper \\
\Rightarrow & \quad \langle \text{ (36) } \rangle \\
 & s.good \wedge s.lower \leqslant i - 1 \leqslant s.upper \wedge i - 1 < s.upper \\
\Rightarrow & \quad \langle \ i - 1 < s.upper \rightarrow i \leqslant upper; s.lower \leqslant i - 1 \rightarrow s.lower \leqslant i \ \rangle \\
 & s.good \wedge s.lower \leqslant i \leqslant s.upper
\end{array}
$$

$$
\begin{array}{ll}
 & Result := Result.max(s.item(i)) \\
= & \langle \text{meaning of a nested query call (35) in Definition 4.3} \rangle \\
 & \textbf{local } v : INTEGER;\ v := s.item(i);\ Result := Result.max(v) \\
= & \langle \text{semantics of query assignments Definition 4.3} \rangle \\
 & \textbf{local } v : INTEGER; \\
 & \quad v : (\!| s \neq Void \wedge s.valid\_index(i), v = \backslash s.item(i) |\!); \\
 & \quad Result : (\!| true, Result = \backslash Result \uparrow v |\!) \\
\rightarrow & \langle \text{sequential composition (4)} \rangle \\
 & Result : (\!| s \neq Void \wedge s.valid\_index(i), Result = \backslash Result \uparrow s.item(i) |\!)
\end{array}
$$

Fig. 15. Proof of final refinement step

$$
\begin{array}{ll}
\Rightarrow & \langle \text{ propositional logic } \rangle \\
 & s.valid\_index(i) = s.lower \leqslant i \leqslant s.upper \rightarrow s.good \wedge s.valid\_index(i) \\
\Rightarrow & \langle \text{ (36) } \rangle \\
 & s.good \wedge s.valid\_index(i) \\
\Rightarrow & \langle \text{ by definition of } s.good \rangle \\
 & s \neq Void \wedge s.valid\_index(i)
\end{array}
$$

A step unique to OO refinement occurs in refining the loop body to a sequence of assignments. The second assignment is

$$Result := Result.max(s.item(i))$$

which is a nested query call. The proof is shown in Fig. 15.

Many of the proof obligations and refinement steps shown in Fig. 12 are very similar to refinement steps in imperative program design calculi. For example, the steps for introducing an initialized loop, or a simple assignment statement, pattern those seen in [Heh93] and [Mor94]. The exact proof obligations required to discharge steps (a)–(c) can easily be mechanically produced by applying the quoted refinement rules.

However, we point out that in all of these proof steps, including those for the query calls, no complicated mathematics is required: the proofs primarily use substitutions, and there is only one quantifier – that to introduce the local variable. A theorem prover like PVS would discharge most of the provisos for this proof automatically.

One might expect that the conjunct containing the *maximum* quantifier in the loop invariant would have to be a comment in Eiffel. However, the latest implementations of Eiffel support the notion of an agent [Mey00] which can be used to equip assertions with executable predicates with quantifiers.

To complete the process, the classes *INTEGER* and *ARRAY* (and all their dependent classes) should now be refined. However, these classes belong to a standard library, and so we can assume that they have been implemented and their correctness ensured.

References

[AC96] Abadi M, Cardelli L (1996) A theory of objects. Springer, Berlin Heidelberg New York

[Abr96] Abrial J-R (1996) The B-book Cambridge. University Press, Cambridge

[BH97] Bancroft PG, Hayes IJ (1997) Type extension and refinement. In: Proceedings of formal methods pacific (FMP'97). Springer, Berlin Heidelberg New York

[BJ01] van den Berg J, Jacobs B (2001) The LOOP compiler for Java and JML. In: Proceedings of TACAS 2001. LNCS 2031. Springer, Berlin Heidelberg New York

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

[CS99] Cavalcanti A, Sampaio A, Woodcock J (1999) An inconsistency in procedures, parameters, and substitution in the refinement calculus. Sci Comput Programming 33(87–96)

[CO95] Crow J, Owre S, Rushby J, Shankar N, Srivas M (1995) A tutorial introduction to PVS. In: Proceedings of WIFT'95, Springer, Berlin Heidelberg New York

[DL98] Detlefs DL, Leino KRM, Nelson G, Saxe JB (1998) Extended static checking. SRC Research Report 159

[DL01] Dhara K, Leavens G (2001) Mutation, aliasing, viewpoints, modular reasoning, and weak behavioral subtyping. Technical Report #01-02, Department of Computer Science, Iowa State University

[Esc00] Escher Technologies, Inc (2000) Getting started with perfect. Available from www.eschertech.com
[GS93] Gries D, Schneider F (1993) A logical approach to discrete mathematics. Springer, Berlin Heidelberg New York
[Gri97] Griffiths A (1997) Modular reasoning in object-Z. Technical Report 97-28, Software Verification Research Center, University of Queensland
[Heh93] Hehner ECR (1993) A practical theory of programming. Springer, Berlin Heidelberg New York
[Hoa85] Hoare CAR (1985) Communicating sequential processes. Prentice-Hall, Englewood Cliffs
[JM97] Jezequel J-M, Meyer B (1997) Design-by-Contract: the lessons of the Ariane 5. IEEE Comput 30(2)
[Jon90] Jones CB (1990) Systematic software development using VDM, 2nd edn. Prentice-Hall, Englewood Cliffs
[KM95] Kent S, Maung I (1995) Quantified assertions in Eiffel. In: Proceedings of TOOLS Pacific 1995, Prentice-Hall, Englewood Cliffs
[Lan95] Lano K (1995) Formal object-oriented development. Springer, Berlin Heidelberg New York
[Lea97] Leavens G (1997) Larch/C++ Reference Manual Version 5.14. Available at www.cs.iastate.edu/∼leavens/larchc++.html
[LB00] Leavens G, Baker A, Ruby C (2000) Preliminary design of JML: a behavioural interface language for Java. Technical Report #98-06j, Department of Computer Science, Iowa State University
[Lei95] Leino KRM (1995) Toward reliable modular programs. PhD Thesis, Department of Computer Science, California Institute of Technology
[Lei97] Leino KRM (1997) Ecstatic: an object-oriented programming language with an axiomatic semantics. In: Proceedings of the fourth international workshop on foundations of object-oriented languages
[LM99] Leino KRM, Manohar R (1999) Joining specification statements. Theor Comput Sci 216(1–2):375–394
[LNS00] Leino KRM, Nelson G, Saxe JB (2000) ESC/Java user's manual. Technical Note 2000-002, Compaq Systems Research Center
[LW94] Liskov B, Wing J (1994) A behavioural notion of subtyping. ACM Trans Programming Language Syst 16(6)
[Mey92] Meyer B (1992) Eiffel: the language. Prentice-Hall, Englewood Cliffs
[Mey97] Meyer B (1997) Object-oriented software construction, 2nd edn. Prentice-Hall, Englewood Cliffs
[Mey00] Meyer B (2000) Agents, iterators, introspection. ISE Inc. Technical Paper
[Mey03] Meyer B (2003) Towards practical proofs of class correctness. In: Proceedings of ZB-2003. LNCS 2651. Springer, Berlin Heidelberg New York
[Mor94] Morgan CC (1994) Programming from specifications, 2nd edn. Prentice-Hall, Englewood Cliffs
[Mul01] Müller P (2001) Modular specification and verification of object-oriented programs. PhD Dissertation, Fern-Universitat Hägen
[PO99] Paige RF, Ostroff JS (1999) Developing BON as an industrial-strength formal method. In: Proceedings the of world congress on formal methods (FM'99). Vol I. LNCS 1708. Springer, Berlin Heidelberg New York
[PO01] Paige RF, Ostroff JS (2001) Metamodelling and conformance checking with PVS. In: Proceedings of the fundamental aspects of software engineering (FASE'01). LNCS 2029. Springer, Berlin Heidelberg New York
[PO03] Paige RF, Ostroff JS, Brooke PJ (2003) Formalizing Eiffel reference and expanded types in PVS. In: Proceedings of the international workshop on aliasing, confinement, ownership (IWACO'03), co-located with ECOOP'03, Utrecht University Technical Report UU-CS-2003-030
[Par72a] Parnas DL (1972) A technique for software module specification with examples. Commun ACM 15(5)
[Par72b] Parnas D (1972) On the criteria to be used in decomposing systems into modules. Commun ACM 15(12)
[Par92] Parnas D (1992) Tabular representation of relations. CRL Report 260, Communications Research Laboratory, McMaster University
[RG00] Richters M, Gogolla M (2000) Validating UML models and OCL constraints. In: Proceedings of unified modeling language 2000. LNCS 1939. Springer, Berlin Heidelberg New York
[RE98] de Roever W-P, Englehardt K (1998) Data refinement: model-oriented proof methods and their comparison. Cambridge University Press, Cambridge
[RJ99] Rumbaugh J, Jacobson I, Booch G (1999) The unified modeling language reference manual. Addison-Wesley, Reading
[Smi95] Smith G (1995) Reasoning about object-Z specifications. In: Proceedings of the asia-pacific software engineering conference 1995. IEEE Press, Piscatawasy
[Smi00] Smith G (2000) The object-Z specification language. Kluwer, Dordrecht
[Smi02] Smith G (2002) Introducing reference semantics via refinement. In: Proceedings of the international conference on formal engineering methods 2002. LNCS 2495. Springer, Berlin Heidelberg New York
[Spi92] Spivey JM (1992) The Z reference manual, 2nd edn. Prentice-Hall, Englewood Cliffs
[WN95] Walden K, Nerson J-M (1995) Seamless object-oriented software architecture. Prentice-Hall, Englewood Cliffs
[Wor94] Wordsworth J (1994) Software development with Z. Addison-Wesley, Reading

Received January 2000
Accepted in revised form August 2003 by B.C. Pierce