ERC - an Object-oriented Refinement Calculus for Eiffel

Richard Paige and Jonathan S. Ostroff

Technical Report CS-2001-05

August 1, 2001

Department of Computer Science

4700 Keele Street North York, Ontario M3J 1P3 Canada

Submitted August (2001) 3: 1–000

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. {paige,jonathan}@cs.yorku.ca[1]

Keywords: refinement calculi; algorithm refinement; object orientation; Eiffel; modular reasoning

7 November 2001

Abstract. We present a refinement calculus for transforming object-oriented specifications into immediately executable, correct programs in Eiffel. The calculus includes a collection of algorithm refinement rules, in particular rules for introducing *feature calls*, which can be used to refine a specification to a program in a subset of Eiffel, thus assisting in the seamless development of programs from specifications. We provide a modular process for partwise refinement of object-oriented specifications into programs, in the sense that specifications can be transformed to code class-by-class and feature-by-feature. And we discuss how automated support for such a process can be developed based on existing tools.

1. Introduction

It has always been desired for formal methods to 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

[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. paige@cs.york.ac.uk

[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. OO extensions of these languages, such as Object-Z [DR94] 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 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 an 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, viz., the structuring of large systems via classes and the client-supplier and inheritance relationships between classes (described in more detail in the sequel), and use of dynamic dispatch and polymorphism. Object-oriented extensions of formal methods such as Object-Z, VDM++, and Larch/C++ [Lea97] do not have comprehensive refinement rules that can be used to transform specifications into implemented code in an OO programming language that has seen wide industrial use.

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 the sequel, 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* [Mey92]; all classes on which the root 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.

Sections 3 and 4 contain the main contributions of the paper. Section 3 provides a fundamental set of refinement rules for Eiffel, focussing on procedural language constructs, e.g., loops and sequencing. An important contribution of Section 3 is a theorem that allows us to re-use 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 will be explained in Section 2, Eiffel possesses both *reference types* and *expanded types* (sometimes referred to as "subobject types"). We will formulate a theory of reference types in Section 4, and will by default assume that all types – except primitives such as integers and booleans – are references. We will 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. We illustrate the refinement process with a short example, in Section 6. In Section 7, we discuss automation, and our goal of supporting refinement and verification with Eiffel using PVS. Finally, in Section 8, we discuss related work.

2. Eiffel and BON

Eiffel is an object-oriented programming language and method [Mey92, Mey97]; it provides constructs typical of the object-oriented 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 software engineering ideas of Eiffel to their logical conclusion in the area of analysis and design. The result is a method which contains a set of concepts and corresponding graphical notations to support object-oriented modeling centered around the three principles of seamlessness, reversibility (the ability to automatically produce BON diagrams 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 close enough to Eiffel that its graphical notation may be viewed as a graphical dialect of Eiffel for the purpose of this article.

We start with a brief overview of Eiffel/BON, focussing 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 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 re-attachment) 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 OO 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 the object-oriented generalization of the notion of a variable. An entity is a name in a software text, meant to be associated at run-time 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 the sequel). 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* ($e2 : C$), 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 references may be attached to obj.

An entity is associated with a value, and a value is either a reference or an object (basic or complex). In the sequel, we use the notation $refs(e1, e2)$ if both entities are references, $objs(e1, e2)$ if both entities are simple objects, and $objc(e1, e2)$ if both entities are associated with complex objects. If $refs(e1, e2)$ holds and $e1$ and $e2$ are

attached to the same object, or are both void, then we write $e1 \stackrel{r}{=} e2$ using the equality symbol shown for *reference equality*. The weaker notion of *object equality* (field-by-field equality) is denoted by $equal(e1, e2)$ — this notion will be defined in the sequel.

It is normally clear from the context what we mean when use the unadorned equality symbol, i.e. $e1 = e2$. We mean $e1 \stackrel{r}{=} e2$ if $refs(e1, e2)$. 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 will be described in the sequel).

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.

Fig. 1 contains a short example of an interface of 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 client classes $EMPLOYER$ and $GOVERNMENT$, or by clients that are descendants of them.

Routines may optionally have contracts, written in the Eiffel assertion language, as preconditions (**require** clauses), postconditions (**ensure** clauses) and class invariants. In postconditions, the keyword **old** can be used to refer to the value of an expression when the feature was called. 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 $Result$.

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

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

[3] Technically, 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 the sequel).

[4] Eiffel version 4.5 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, 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 • ∃ p ∈ c.parents • p = Current
end
```

Fig. 1. Class $CITIZEN$

enhanced by the fact that assertions may refer to any query (e.g., the postcondition of *divorce* refers to the queries $single$ and $spouse$).

A class invariant is an assertion (conjoined terms are separated by semicolons) that must be $true$ whenever an instance of the class is used by another object (i.e., whenever a client can call an accessible feature). Private 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 object-oriented computation is the feature call $target.f(x)$ where $target$ is an expression and f 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, class $CITIZEN$ has eight queries and one command. The attributes are $name$, sex, age, $spouse$, $children$, $parents$, and $salary$. The first three attributes are of expanded type, while the next three are of reference types. 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). 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 $children$:

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

$SET[CITIZEN]$. 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 $e1.q1(e2.q_2(e3))$. Entities must be declared in the program text before they are used. Eiffel also provides support for expressions involving prefix and infix operators, but these are just syntactic sugar for query calls. Table 1 defines the notation that we will use throughout the rest of the paper for describing entities and features. Table 2 defines the precedence of operators in Eiffel.

e	entities (includes attributes a, arguments of routines x, local variables, $Current$, $Result$)
$e.type$	the static type of the object attached to entity e
$e.\mu$	the syntactically legal multi-dots associated with e (see sequel)
c	commands
q	queries (including attributes)
f	features, i.e., queries (including attributes) and commands
r	routines (computation)
$r.\rho$	bunch of reference entities associated with r (see sequel)
$r.\pi$	bunch of entity partitions associated with r (see seqel)
S	single state formulae
D	double state formulae
P, Q	predicates including single and double state formulae

Table 1. Notation for entities and features

0.	$+$, $-$, $\neg$, **pre**, **old** (unary prefix operators), . (dot notation)
1.	$*$, $/$
2.	$+$, $-$, $\cap$, $\cup$ etc.
3.	relations: $=$, $\stackrel{r}{=}$, $\neq$, $<$, $\leq$ etc. and $\in$
4.	$\wedge$, $\vee$ (logical operators)
5.	$\rightarrow$, $\leftarrow$, $\sqsubseteq$
6.	$\equiv$, $\not\equiv$
7.	$\bullet$, $\mid$
8.	$:=$ (assignment), $\forall$, $\exists$
9.	$\widehat{=}$ (definition), ; (conjunction that separates assertion clauses)

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

[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 don't know yet who the parent is

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

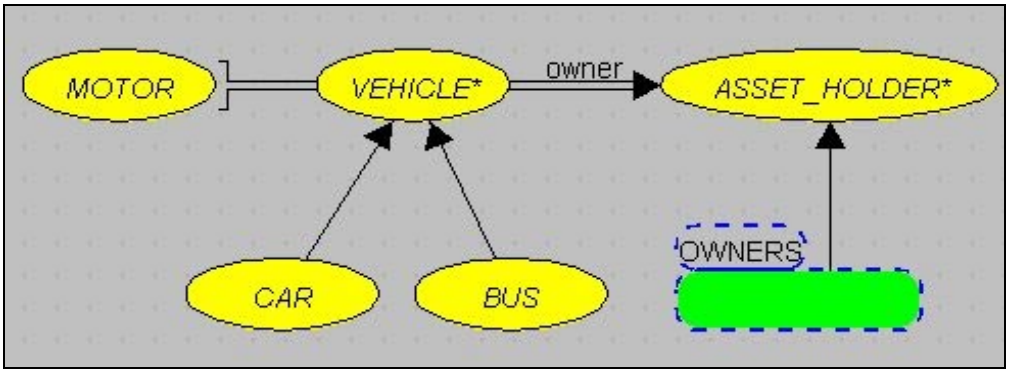

Fig. 2. BON diagram for aggregations, associations, inheritance, and clusters

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, class $VEHICLE$ has a reference attribute $owner$ with type $ASSET\_HOLDER$.

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 will be 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.

```
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

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

$$(\mathbf{local}\ e : T;\quad e := target;\ e.f(x1, x2))$$

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

In the sequel, we will 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 will be used to motivate the axiomatic refinement calculus of Eiffel programs and specifications in the next section.

[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.

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 will 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 Eiffel construct are the poststate σ after the construct's computation terminates (the output), as well as the prestate $\mathbf{old}\,\sigma$ (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 variables of the routine, and a conceptual global time variable t. Thus, σ is a *bunch*[8] of entities $\sigma \mathrel{\widehat{=}} e1, e2, \ldots, en, t$. Correspondingly, $\mathbf{old}\,\sigma = \mathbf{old}\,e1, \mathbf{old}\,e2, \ldots, \mathbf{old}\,en, \mathbf{old}\,t$ (note that $\mathbf{old}\,e$ is also an entity different from e). If $\bar{\sigma}$ is a sub-bunch of σ, then we define $same(\bar{\sigma})$ by $same(\bar{\sigma}) \mathrel{\widehat{=}} (\forall\, e' \in \bar{\sigma} \bullet e' = \mathbf{old}\,e')$.

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 must make 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 $\mathbf{old}\,\sigma$ and σ) indicating the computed poststates, then the specification of the routine $r.spec$ is defined as [Heh93]

$$r.spec \mathrel{\widehat{=}} \mathbf{old}\,r.pre \rightarrow r.post \wedge time \tag{1}$$

$$time \mathrel{\widehat{=}} t \geq \mathbf{old}\,t \wedge t \neq \infty \tag{2}$$

where t is a conceptual global clock representing time, and $t \in \sigma$. Operator precedences are 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 the execution of each program construct terminates in finite time ($t \neq \infty$).

The specification $r.spec$ is true iff (a) the precondition is not satisfied (in which case any behaviour is permitted); and (b) if the precondition is satisfied then the routine's execution must terminate in finite time with the postcondition true. A *program* is a specification that has been implemented. We describe programs in ERC as follows

$$\mathbf{skip} \mathrel{\widehat{=}} e1 = \mathbf{old}\,e1 \wedge e2 = \mathbf{old}\,e2 \wedge \ldots \wedge en = \mathbf{old}\,en \wedge time \tag{3}$$

$$e1{:=}exp \mathrel{\widehat{=}} \mathbf{old}\,defined(exp) \rightarrow e1 = \mathbf{old}\,exp \wedge e2 = \mathbf{old}\,e2 \wedge \ldots \wedge time$$

$$\mathbf{if}\ b\ \mathbf{then}\ P\ \mathbf{else}\ Q \mathrel{\widehat{=}} (\mathbf{old}\,b \rightarrow P) \wedge (\neg\,\mathbf{old}\,b \rightarrow Q) \wedge time$$

$$P;\ Q \mathrel{\widehat{=}} \exists\,\sigma' \bullet P[\sigma := \sigma'] \wedge Q[\mathbf{old}\,\sigma := \sigma']$$

$$(\mathbf{local}\,e : T;\ P) \mathrel{\widehat{=}} (\exists\, e, \mathbf{old}\,e : T \bullet P)$$

[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, e.g., $e1, e2, e3$. The standard operations of set theory will be used for bunches, e.g. given a bunch $b2 = e4, e5$, then $b \cup b1 = e1, e2, e2, e4, e5$ — we also write this union as "$b, b1$" — and $e5 \in b2$ expresses the fact that bunch $b2$ contains $e5$. The symbol $\sim$ is used to obtain the contents of a set; thus, if b is a bunch, then $\sim\{b\} = b$.

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 (3) only covers the case where both the entity $e1$ and the expression exp are **expanded**. In the sequel, we will 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 be a sequence of specifications.

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

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

will denote a specification[9] for which e is the frame (the bunch of entities that may change between the pre-state and post-state), S is the precondition, and D is the post-condition. An equivalent (Larch-like) syntax for writing specifications which cannot be written conveniently on one line is:

$\mathbf{modifies}\ e$
$\mathbf{require}\ S$
$\mathbf{ensure}\ D$

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 (3) more concisely. For example, the expanded assignment statement $e1 := exp$ can be written as $e1, t : \llparenthesis\ defined(exp),\ \ e1 = \mathbf{old}\ exp\ \rrparenthesis$. From the definitions of programs in (3), many useful laws of programming can be derived. For example, it is straightforward to provide a proof using the relevant predicative definitions for the following laws.

$$\mathbf{if}\ b\ \mathbf{then}\ P\ \mathbf{else}\ P \quad \equiv \quad P \tag{4}$$

$$\begin{array}{l}\text{If } (D_1[\sigma := \sigma'] \rightarrow S_2[\mathbf{old}\,\sigma := \sigma']) \\ \text{then } (D_1 \wedge time\ ;\ S_2 \rightarrow D_2 \wedge time) \equiv (D_1\ ;\ D_2) \wedge time\end{array} \tag{5}$$

where D_1, D_2 are double state-assertions and S_2 is a single state assertion. The *time sequencing rule* (5) asserts that we can ignore the embedded $time$ components under the stated assumption, provided that the execution of each construct in the sequence itself terminates in finite time.

The *simple substitution rule* is derived from the assignment definition and sequencing (3) and is similar to a corresponding rule in [Heh93]. It is particularly useful in simplifying long sequences of assignments.

[9] Morgan's syntax is actually $x : [S, D]$, but we use the square brackets for substitution.

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

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

An ERC specification $spec$ can, in general, be any predicate with free variables in $\mathbf{old}\,\sigma$ 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.

$$spec \text{ is implementable} \ \widehat{=}\ (\forall\, \mathbf{old}\,\sigma \bullet \exists\, \sigma \bullet spec \wedge t \geq \mathbf{old}\ t) \tag{6}$$

The definition asserts two thing. First, every prestate of an implementable specification must have at least one poststate. Second, time cannot decrease between the input and the output. This makes, for example, the specification $Q \mathrel{\widehat{=}} x = 2 \wedge t < \infty$ unimplementable, because $Q \wedge t \geq \mathbf{old}\ t$ is unsatisfiable for an initial value of $\mathbf{old}\ t = \infty$.

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:

$$spec \sqsubseteq impl \ \widehat{=}\ (\forall\, \mathbf{old}\,\sigma, \sigma \bullet impl \rightarrow spec) \tag{7}$$

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 Hehner [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 will thus reuse the Z rule in this work. However, we must first justify the fact that we can use Z refinement rules in ERC.

Assuming $\sigma = e1, e2, t$, we mentioned earlier that a specification statement can be translated into a timed predicate as follows (assuming that semantically equivalent expressions and operators can be translated appropriately):

$$e1, t : (\!|\ S,\ D\ |\!) \quad \widehat{=} \quad S \rightarrow D \wedge (e2 = \mathbf{old}\ e2) \wedge time \tag{8}$$

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 $P.pre \mathrel{\widehat{=}} (\exists\, \sigma' \bullet P)$. 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**, and by removing the primes from all primed entities.

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

$$P_1 \sqsubseteq_z P_2 \;\widehat{=}\; (\forall\, \mathbf{old}\, \sigma, \sigma \bullet P_1.pre \rightarrow P_2.pre \wedge (P_2 \rightarrow P_1)) \tag{9}$$

whereas the ERC version of refinement (7) was expressed as

$$P_1 \sqsubseteq P_2 \;\widehat{=}\; (\forall\, \mathbf{old}\, \sigma, \sigma \bullet (P_2.pre \rightarrow P_2) \rightarrow (P_1.pre \rightarrow P_1)) \tag{10}$$

Since (9) entails (10), we have the following theorem:

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

$$e{:}\langle\!|\, S_1,\ D_1 \,|\!\rangle \sqsubseteq_z e{:}\langle\!|\, S_2,\ D_2 \,|\!\rangle \quad\rightarrow\quad e{:}\langle\!|\, S_1,\ D_1 \,|\!\rangle \sqsubseteq e{:}\langle\!|\, S_2,\ D_2 \,|\!\rangle$$

Since refinement in Morgan's calculus is equivalent to Z refinement, it follows that any refinement rule from [Mor94, 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:

$$e1{:}\langle\!|\, S,\ D \,|\!\rangle \;\equiv\; e1, e2{:}\langle\!|\, S,\ D \wedge e2 = \mathbf{old}\, e2 \,|\!\rangle$$

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

$$e{:}\langle\!|\, S,\ D \,|\!\rangle \;\sqsubseteq\; \mathbf{local}\, x : T;\ e, x{:}\langle\!|\, S,\ D \,|\!\rangle$$

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 removes the **old** keyword. Here is an example.

$$\begin{array}{ll} & (x = \mathbf{old}\ y \wedge y = \mathbf{old}\ (x + y))' \\ = & (x' = y) \wedge (y' = (x + y)) \end{array}$$

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, p218], and relies on Theorem 1.

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

$$
\begin{array}{lll}
Loop & ::= & \textbf{modifies}\ e, t \\
 & & \textbf{from}\ Init \\
 & & \textbf{invariant}\ I \\
 & & \textbf{variant}\ v \\
 & & \textbf{until}\ b \\
 & & \textbf{loop} \\
 & & \quad Body \\
 & & \textbf{end}
\end{array}
$$

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:

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

provided that

$$
\begin{array}{rcl}
S & \rightarrow & S_{init} \\
\textbf{old}\ S \wedge D_{init} \wedge same(\bar{\sigma}) & \rightarrow & I \\
\textbf{old}\ S \wedge I \wedge b & \rightarrow & D \wedge same(\bar{\sigma}) \\
\textbf{old}\ S \wedge I \wedge \neg\, b & \rightarrow & S_{body} \\
\textbf{old}\ S \wedge I \wedge \neg\, b \wedge (D_{body} \wedge same(\bar{\sigma}))' & \rightarrow & I[\_ := \_'] \\
\textbf{old}\ S \wedge I \wedge \neg\, b & \rightarrow & v \geq 0 \\
\textbf{old}\ S \wedge I \wedge \neg\, b \wedge (D_{body} \wedge same(\bar{\sigma}))' & \rightarrow & v' < v
\end{array}
$$

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

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 $\textbf{old}\,\sigma$), 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}). The second clause in the loop rule says that the initialization must establish the invariant.

The third clause says that on exit (i.e., b true) the invariant must establish the required specification postcondition D. If the exit condition is false, then the body must be executed while preserving the invariant.

The fourth clause asserts that the body is only executed when it is safe to do so (i.e., when its precondition S_{body} holds). The fifth clause is a three-state formula. We assume that the invariant holds between the starting state $\textbf{old}\,\sigma$ 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 still continue to hold between $\textbf{old}\,\sigma$ 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 (2) of the loop is satisfied. The loop is the only construct where termination is a problem. Provided we have justified the clauses in the loop rule, we can henceforth ignore $time$ in our derivations, by appealing to (5).

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 object-oriented 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 atop 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 an OO program, and so as to maintain the level of abstraction provided by OO technology.

We commence presentation of the rules with some notation and syntax. The rules, and our theory of reference types, will be 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 $a5$).

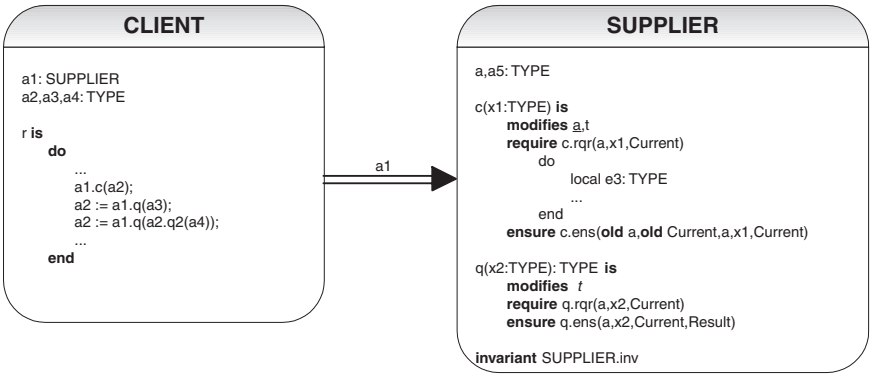

Fig. 4. BON diagram with feature notation

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[10] are defined as follows:

$$r.pre \quad \widehat{=} \quad r.rqr \wedge C.inv \tag{11}$$

[10] 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 $r.pre \mathrel{\widehat{=}} r.rqr$.

$$r.post \quad \widehat{=} \quad r.ens \wedge C.inv \wedge r.same$$
$$r.same \quad \widehat{=} \quad \forall\, e \in (\sigma - r.modifies) \bullet e = \mathbf{old}\ e$$

The routine specification $r.spec$ defined in (1) uses the above definitions.

The state space σ in the context of routine r of class $CLIENT$ will include time t, attributes $a1, a2, a3, a4$ of class $SUPPLIER$ 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 feature accesses of $a1$, i.e. $a1.\,\mu = a1, a1.a, a1.a5, a1.a.a6, a1.a.a7 \ldots$ etc. In general, given an entity $e1$, we let $e1.\,\mu$ denote the bunch of syntactically legal multi-dots [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 run time [Mey97, p235]. The $\mathbf{create}\ e$ instruction in a routine creates a new instance of C, initializes each field to a default value (e.g., $BOOLEAN$s 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 partitions for reference types

Reference types are critical for any theory of object-oriented programming. Our theory of reference types, based on so-called *entity partitions*, will be used by assignment statements, **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. This approximate 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 leads to lengthy expresions (involving storage arrays and sequences of element overridings) in refinement, although an automated tool could help hide some of the complexity. Consequently, we introduce below an approach based on entity partitions, 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 $\mathbf{create}\ e$, 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 (the sequel will deal 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).

We point out that $\mathbf{create}$ 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 partitions. 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). 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 $r.\rho = e1, e2, e3$.

```
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, t
                do
                        create e1;
                        create e3;
                        e2 := e1;
                        e1.c(1);
                        e1 := e3;
                        e3.c(3);
                ensure e1.a = 3 = e3.a  ∧  e2.a = 1  ∧  e1.b = old b
                end
end
```

Fig. 5. Aliasing and feature calls for references

Associated with each entity $e \in r.\rho$ there is a corresponding *entity partition* which we denote by $\underline{e}$. Before the first creation statement in the body of r, the partition is an empty set. After a $\mathbf{create}\ e$ instruction, $\underline{e} = \{e\}$, and this entity partition is used to keep track of all reference entities in σ that point to the same object as e. We let $r.\pi$ denote the bunch of all entity partitions of routine r, and we require that these partitions be disjoint (this will be formalized in the sequel). 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 entity partitions. Suppose $\underline{e} = \{e, e2, e3\}$, i.e. the three entities e, $e2$, and $e3$ all refer to the same object. Then

$$\underline{e}[e2, e3] \quad \widehat{=} \quad \underline{e} = \{e, e2, e3\} \tag{12}$$

We also write the entity partition $\underline{e1}[e1]$ as $\underline{e1}[]$, to avoid repetition.

The notation in (12) 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 automatically 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 will carry around set expressions of the form (12), which describe the attachments of objects to entities. We can now define reference equality using our notion of partitions:

$$(e1 \stackrel{r}{=} e2) \;\; \widehat{=} \;\; (\underline{e1} = \underline{e2}) \tag{13}$$

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 using reference equality, provided that its occurrence in a program is syntactically legal, as follows:

$$equal(e1, e2) \;\; \widehat{=} \;\; \left(\begin{array}{l} (refs(e1, e2) \rightarrow e1 \stackrel{r}{=} e2) \\ \wedge\, (objs(e1, e2) \rightarrow e1 = e2) \\ \wedge\, (objc(e1, e2) \rightarrow (\forall\, e1.a \mid attribute(a, e1) \bullet equal(e1.a, e2.a)) \end{array} \right) \tag{14}$$

where $attribute(a, e1)$ means that a is an attribute of the object associated with $e1$. The above definition captures the meaning of the standard equality operator of Eiffel [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.
2. If $e2$ is a basic type ($INTEGER, BOOLAN$ etc.), then the result is true if both objects have the same value.[11]
3. If $e2$ is associated with a complex object, then the fields must be identical, i.e. every corresponding reference field of $e1$ and $e2$ must point to the same object, and every object field of $e2$ is recursively equal to the corresponding field of $e1$, i.e.

$$equal(e1, e2) \rightarrow (\forall\, m \in e1.\, \mu \bullet e1.m = e2.m) \tag{15}$$

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:

$$e1 \stackrel{r}{=} e2 \rightarrow equal(e1, e2) \tag{16}$$

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

$$\forall\, \underline{ei}, \underline{ej} \in r.\pi \mid (\underline{ei} \neq \underline{ej}) \bullet \underline{ei} \cap \underline{ej} = \varnothing \tag{17}$$

Axiom (17) states that two different entity partitions do not intersect, because they refer to different objects; any entity can refer to at most one object at any time - this captures the precisely specified semantics of Eiffel entities described in Section 2. 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.

$$\forall\, \underline{e} \in r.\pi \bullet (\forall\, e1, e2 \in \underline{e} \bullet e1 \stackrel{r}{=} e2) \tag{18}$$

[11] 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 if they have the same length and each field is (recursively) equal.

Axiom (18) states that if two entities are in the same entity partition, 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 $e1.b = e2.b$.

$$\forall\, \underline{e} \in r.\pi \bullet \forall\, e \in \underline{e} \bullet e \neq \mathit{Void} \tag{19}$$

$$\forall\, e \in r.\rho \mid (\forall\, \underline{e}' \in r.\pi \bullet e \notin \underline{e}') \bullet e = \mathit{Void} \tag{20}$$

$$\forall\, e \in r.\rho \ \bullet\ (e = \mathit{Void}) \ \equiv\ (\underline{e} = \varnothing) \tag{21}$$

Axiom (19) states that any entity in an entity partition is attached to an object, and thus cannot be Void. Axiom (20) states that if an entity e is in no entity partition, then it is Void. This is the state an entity is in after declaration, or after an assignment $e := \mathit{Void}$. Axiom (21) equates a Void reference with an empty entity partition.

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

Axiom (22) asserts that if two entities are in the same partition, then the partition can be called by either name.

Our notion of entity partitions can be used to define the semantics of instructions that use reference types in ERC such as the create statement, assignment and feature call. Partitions may appear in frames. $\underline{e}$ in the **modifies** clause means that we may modify the partition $\underline{e}$ as well as any other partitions $\underline{ei}$ such that $ei \in \mathbf{old}\ \underline{e}$ or $ei \in \underline{e}$.

The $\mathbf{create}\ e$ instruction creates a new object and attaches it to entity e; any previous reference or attachment via entity e is lost; however, any other entities that referred to the object originally attached to e remain. The semantics of $\mathbf{create}\ e$ is given in Definition 4.1.

Definition 4.1. (Entity creation semantics) The instruction

$$\mathbf{create}\ e$$

is defined as

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

where

$$\mathit{remove}(e)\ \ \widehat{=}\ \ \forall\, ei \in \mathbf{old}\ \underline{e} \mid ei \neq e \bullet \underline{ei} = \mathbf{old}\ \underline{ei} - \{e\},$$

and where

$$\mathit{nochange}(e)\ \ \widehat{=}\ \ \forall\, ei \mid ei \neq e \wedge ei \notin \mathbf{old}\ \underline{e} \bullet \underline{ei} = \mathbf{old}\ \underline{ei},$$

The $\mathit{default}(e)$ clause asserts that each $e.a$ is set to its default value on creation as described in [Mey97] (e.g., if a is a $\mathit{BOOLEAN}$ it is set to false, if it's a reference it is set to Void etc.). The $\mathit{nochange}$ clause is not strictly necessary, as it follows from the frame; however, it is inserted for clarity to indicate that all other partitions do not change. If the class $e.\mathit{type}$ has a creation routine r (whose purpose is to establish the class invariant), then we must execute this routine after the creation statement, i.e., $\mathbf{create}\ e;\ e.r$.

The postcondition of **create** says that the entity partition $\underline{e}$ contains a single entity e (i.e., only e refers to the new object), and that its prior entity partition $\mathbf{old}\ \underline{e}$ does not contain e. We note that if $\mathbf{old}\ \underline{e} = \varnothing$ or $\mathbf{old}\ \underline{e} = \{e\}$ then $\mathit{remove}(e) \equiv \mathit{true}$.

As an example, assume that we are in a state in which $\underline{e1}[e2]$ holds and we then

execute $\mathbf{create}\ e1$. Definition 4.1 reduces to

$$\begin{array}{l}\mathbf{modifies}\ \underline{e1}, \underline{e2}, t\\ \mathbf{ensure}\ \underline{e1} = \{e1\} \wedge \underline{e2} = \{e2\}\end{array}$$

After executing **create** $e1$, $e1$ refers to a new object, while $e2$ refers to the original object (which now has the name $e2$).

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 [Mey97], so that $e2$ can be assigned to $e1$. Then

$e1 := e2$

is defined as

$$\begin{array}{l}\mathbf{modifies}\ \underline{e1}, \underline{e2}, t\\ \mathbf{ensure}\ put(e1, e2) \wedge remove(e1)\end{array}$$

where $remove(e1)$ is as in Definition 4.1, and

$$\begin{array}{lcl} put(e1, e2) & \widehat{=} & (\mathbf{if}\ \mathbf{old}\ e2 = Void\ \mathbf{then}\ e1 = e2 = Void \\ & & \quad \mathbf{else}\ \underline{e2} = \mathbf{old}\ \underline{e2} \cup \{e1\}).\end{array}$$

The definition states that if $e2$ is $Void$, then $e1$ is also set to $Void$. Otherwise, $e1$ is added to the entity partition of $e2$ and removed from its old entity partition.

We now present the meaning of query and command calls, focussing on how these calls can effect changes in the state of objects and entity partitions. 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 (in Fig. 4) means

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

Since it is possible for $e1$ to be $e2$ itself, we must use **old** in $q.post$. The target of the call to query q in $e2.q(e3)$ is $e2$, and hence $Current := e2$. We can recursively treat queries of arbitrary complexity using (23)

$$e1 := e2.q(exp) \quad \widehat{=} \quad (\exists\, e \bullet\ e := exp;\ e1 := e2.q(e)) \tag{23}$$

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

Definition 4.3 deals only with the case where a query is called and the result assigned to an entity; this is an extremely common way to effect state changes in object-oriented

programs. However, queries can also be called in pre- or postconditions associated with other features, and in class invariants. If a query call $e2.q(e3)$ appears in a contract P, then P can be evaluated by replacing it as shown in equation 24.

$$P \quad \equiv \quad (q.pre[\alpha] \rightarrow q.post[\alpha]) \rightarrow P \tag{24}$$

$$\alpha \quad \widehat{=} \quad a := e2.a, \mathit{Current} := e2, x2 := e3, \mathit{Result} := e2.q(e3) \tag{25}$$

i.e., we may use the targeted contract of q to evaluate $e2.q(e3)$ in P. We do not need "**old**" in the substitution because queries are side-effect free.

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 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 attributes a and b (in Fig. 4) means

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

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

The **modifies** clause states that the command changes only time and the attributes of entities in the entity partition of $e1$. This also allows the reference $e1$ to change to point at a new object. The **require** clause establishes that any call must be to a non-Void entity. As well, the clause 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.

A final useful axiom captures the notion that once a reference entity is created, it always satisfies its invariant. Let $e : C$ and let class C have invariant $C.\iota$. Let $e.\iota$ be the invariant targeted to e, i.e. $e.\iota = C.\iota[f := e.f, \mathit{Current} := e]$, where f is a feature (or bunch of features) of C. Then, once e is created ($e \neq \mathit{Void}$) it always satisfies its invariant, i.e

$$e \neq \mathit{Void} \;\rightarrow\; e.\iota \tag{26}$$

A simple verification example using partitions and reference types is presented in Appendix A.

4.2. Expanded types

If entities $e1$ and $e2$ are both expanded (i.e., they refer either to a basic value such as integer or boolean, or to a sub-object), then the standard rule for assignment in (3) 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. 6 suggests how to extend our approach to handle them, leaving a full treatment (e.g., expanded parameters) for later work.

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

Fig. 6. Hybrid assignments

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., contracts) 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, the OO system shown in Fig. 5.

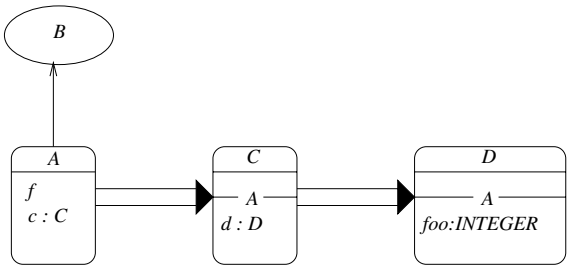

Fig. 7. System structure for demonstrating the modular refinement process

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 will 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.

$$c.d.foo \geq 0$$

Thus, A can make use of features of D through C's subobject, even though there is no direct relationship drawn in Fig. 5 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 will 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 correctly carry out refinement.

The precise definition of the *short-flat* form of a class is defined 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 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[12]:

- $\bar{A} \in$ **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. 5, **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 $\mathbf{spec}\ A$ by refinement steps to $\mathbf{code}\ s$, an Eiffel implementation. In other words, it must be shown that

   $$\forall\, s \in \mathbf{spec}\ A \bullet s \sqsubseteq \mathbf{code}\ s$$

   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 $\mathbf{spec}\ A$ and the contracts of features belonging to $\mathbf{depend}\ A$. No implementations of the features of **depend** A are needed. 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* object-oriented relationships, and information hiding. In particular, information hiding as it exists 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.

[12] This is similar to Leino's notion of dependency relation in [Lei95].

If this level of information hiding [Par72] 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 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 the OO specification had been replaced by directed ones.

This style of modular reasoning is also inherent in Object-Z [DR94]; 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 8.

6. Example

This section illustrates a simple example of refinement in Eiffel. The problem we will solve is a simple one, taken from [Wor94], to find the maximum of a non-empty array s of integers. 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 now provide the Eiffel specification for the class FOO.

class FOO
feature
$s : ARRAY[INTEGER]$

$max\_array : INTEGER$
-- calculate maximum of array s
-- $s.good \;\widehat{=}\; s \neq Void \wedge \neg\, s.empty$
require $s.good$
ensure $Result = (\uparrow j : INTEGER \mid s.valid\_index(j) \bullet s.item(j))$
end

Fig. 8. Class FOO for refinement example

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 will use several logical laws for reasoning taken from [GS93] as well.

Class FOO has one association with class $ARRAY$ and an aggregation with 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 classes $ARRAY$ and $INTEGER$; 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.

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

  $valid\_index(x : INTEGER) : BOOLEAN$
  ensure $Result = (lower \leq x \leq upper)$

  $item(x : INTEGER) : G$
  require $valid\_index(x)$

  $empty : BOOLEAN$
  ensure $Result = (count = 0)$

  invariant $lower \leq upper \wedge count \geq 0$
end

class $INTEGER$ **feature**
  $max(x : INTEGER) : INTEGER$
  ensure $Result = Current \uparrow x$
end

Fig. 9. Excerpt from interfaces of *ARRAY* and *INTEGER* classes

We can now refine the specification of FOO to code. In the process, we will 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

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

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 refinement of a client or descendent class).

Fig. 10 provides a refinement from the specification of max_array to final code. As argued at the end of Section 3.3 and in (5), we ignore $time$. In Fig. 10, 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. 10 just uses the local variable introduction rule. For step (b) we must discharge all the provisos in the loop rule. Some of the provisos are simple, e.g., the first loop proviso $S \rightarrow S_{init}$ reduces to $s \neq Void \rightarrow s \neq Void$ which is trivially true. The second proviso is proved with the use of the invariant axiom (26). The other provisos are relatively straightforward. As an example, the fifth proviso can be discharged as follows:

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

$$\begin{array}{ll} & (D_{body} \wedge same(\bar{\sigma}))' \\ = & \quad \langle \text{ definition } \rangle \\ & i' = i + 1 \wedge s' = s \wedge (Result' = Result \uparrow s'.item(i')) \\ = & \quad \langle \text{ Leibniz and } i' = i + 1 \wedge s' = s \ \rangle \\ & i' = i + 1 \wedge s' = s \wedge (Result' = Result \uparrow s.item(i+1)) \\ = & \quad \langle \text{ from the invariant, } Result = (\uparrow j : INTEGER \mid s.lower \leq j \leq i \bullet s.item(j)); \\ & \quad \text{split off term (8.23) } \rangle \\ & i' = i + 1 \wedge s' = s \wedge Result' = (\uparrow j : INTEGER \mid s.lower \leq j \leq \\ & \quad i + 1 \bullet s.item(j)) \\ = & \quad \langle \text{ Leibniz and } i' = i + 1 \wedge s' = s \ \rangle \\ & i' = i + 1 \wedge s' = s \wedge Result' = (\uparrow j : INTEGER \mid s'.lower \leq j \leq \end{array}$$

$$
\begin{array}{ll}
 & \textbf{spec}\ max\_array \\
= & < definition\ of\ max\_array > \\
 & Result : \langle\!|\ s.good, \\
 & \qquad Result = (\uparrow j : INTEGER \mid s.valid\_index(j) \bullet s.item(j))\ |\!\rangle \\
 & \\
\sqsubseteq & < (a) : \text{Local variable introduction Rule 3.2(b)} > \\
 & \textbf{local}\ i : INTEGER; \\
 & i, Result : \langle\!|\ s.good, \\
 & \qquad Result = (\uparrow j : INTEGER \mid s.valid\_index(j) \bullet s.item(j))\ |\!\rangle \\
 & \\
\sqsubseteq & < (b) : \text{introduce loop Rule 3.3 – all seven provisos hold} > \\
 & \textbf{local}\ i : INTEGER; \\
 & \textbf{modifies}\ i, Result \\
 & \textbf{from}\ P :\ i, Result : \langle\!|\ s.good, \\
 & \qquad i = \textbf{old}\ s.lower \wedge Result = s.item(i)\ \ |\!\rangle \\
 & \textbf{invariant}\ I :\ s.valid\_index(i) \wedge same(s) \wedge \\
 & \qquad Result = (\uparrow j : INTEGER \mid s.lower \leq j \leq i \bullet s.item(j)) \\
 & \textbf{variant}\ V :\ s.upper - i \\
 & \textbf{until}\ i = s.upper \\
 & \textbf{loop}\ W : \\
 & \quad i, Result : \langle\!|\ s.good \wedge s.valid\_index(i) \wedge i < s.upper, \\
 & \qquad i = \textbf{old}\ i + 1 \wedge Result = \textbf{old}\ Result \uparrow s.item(i)\ |\!\rangle \\
 & \textbf{end} \\
 & \\
\sqsubseteq & < (c) : refine\ initialization\ P\ (see\ Fig.\ 12);\ \ refine\ loop\ body\ W\ (see\ Fig.\ 13) > \\
 & \textbf{local}\ i : INTEGER; \\
 & \textbf{modifies}\ i, Result \\
 & \textbf{from}\ i := s.lower;\ Result := s.item(i) \\
 & \textbf{invariant}\ I :\ s.valid\_index(i) \wedge same(s) \wedge \\
 & \qquad Result = (\uparrow j : INTEGER \mid s.lower \leq j \leq i \bullet s.item(j)) \\
 & \textbf{variant}\ V :\ s.upper - i \\
 & \textbf{until}\ i = s.upper \\
 & \textbf{loop} \\
 & \quad i := i + 1;\ Result := Result.max(s.item(i)) \\
 & \textbf{end}
\end{array}
$$

Fig. 10. Refinement tree for max_array (the last step produces executable Eiffel code)

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

The refinement steps for implementing the loop initialization P and the loop body W are detailed in the appendix, along with proofs of the resultant obligations.

Many of the proof obligations and refinement steps shown in Fig. 10 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, Mor94]. The exact proof obligations required to discharge steps (a), (b), and (c) can easily be mechanically produced by applying the quoted refinement rules. As examples, we discharge proof obligations for the loop initialization refinement and the loop body refinement in Appendix A.

A step unique to object-oriented 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 refinement and proof obligations are explained in detail in Appendix B. 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, 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.

7. 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. 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 examples of using PVS to reason about object-oriented 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.

8. Related Work and Conclusions

In this paper, we have provided rules for refinement of object-oriented 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 object-oriented 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.

We are not aware of any other OO approaches that yet provide refinement rules (which do not require OO specifications to be transformed to procedural specifications) down to an actual industrial-strength programming language, though several approaches, such as JML and Larch, 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 solely a specification language, and contains no immediately executable programming language. Algorithm refinement rules remain to be fully worked out for Object-Z, primarily with regards to rules for introducing routine calls within a refinement. The semantics of Object-Z supports *strict modular* reasoning [Gri97]: 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. Finally, the semantics of Object-Z is based on labeled transition systems. The semantics of routines in Eiffel is based on first-order logic, making it easier to support reasoning via available tools, such as the PVS system.

The work on the Extended Static Checker [DL98] for Java 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 by 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. Hence it does not catch all possible errors. It handles reference types as well as primitive types. The Escher language and tool [Es00] is also used for static checking and

automatic verification. With Escher, 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 will be more appropriate when verifying library classes or pre-existing applications, rather than when developing new classes or applications.

Work has been carried out on validating UML object-oriented 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 object-oriented models against metamodel constraints via automated theorem proving is demonstrated by Paige and Ostroff using PVS [PO01].

VDM++ is an OO dialect of VDM. It is based on a three-valued logic. [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. [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 object-oriented 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 modeling 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 modeling language and is syntactically similar to Java. JML is a richer language than BON, in that it supports history constraints and modeling excep-

tions. It has no refinement rules defined, although such rules are feasible to produce. Work is underway on integrating JML with the Extended Static Checker [DL98].

This paper did not deal completely with expanded types nor did it show how to automate the refinement procedures. 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 [Heh93], which supports concurrency and communication, as the underlying specification formalism should make the extension to concurrent and real-time systems feasible.

A. Use of reference type theory

Fig. 11 contains a short example demonstrating how to reason with our 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 in the figure. 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

$\quad \mathbf{create}\ e1;\ \mathbf{create}\ e2$
$\Rightarrow \quad \langle$ entity creation Defn. 4.1 $\rangle$
$\quad \underline{e1}[]\ ;\ \ \underline{e2}[] \wedge \underline{e1} = \mathbf{old}\ \underline{e1}$
$= \quad \langle$ sequential composition (3) $\rangle$
$\quad \exists\, e1' \bullet \underline{e1'}[] \wedge \underline{e2}[] \wedge \underline{e1} = \underline{e1'}$
$= \quad \langle$ one-point rule using $\underline{e1} = \underline{e1'}$ $\rangle$
$\quad \underline{e1}[] \wedge \underline{e2}[]$

where we ignored $time$ using (5) as our justification. Using Defn. 4.4, the targeted call $e1.m(5)$ reduces to $e1 \neq Void \rightarrow e1.a = 5 \wedge same(\underline{e1}, \underline{e2}) \wedge time$, so that

$\quad \underline{e1}[] \wedge \underline{e2}[];\ e1.m(5)$
$\Rightarrow \quad \langle$ Defn. 4.4; $\underline{e1}[] \rightarrow e1 \neq Void$ using (19); sequential composition (3) $\rangle$
$\quad \underline{e1}[] \wedge \underline{e2}[] \wedge e1.a = 5$

The rest of the proof in Fig. 11 continues along the same lines.

B. Proof of Refinement Steps for Section 6

We present the refinement steps and corresponding proofs that are required in implementing the loop initialization and loop body. These refinement steps were omitted, for the sake of brevity, from Fig. 10. It is useful to consider the refinement steps, since they illustrate how to introduce the fundamental constructs in OO computing, namely feature calls. The first step is the refinement of the loop initialization, P, by the sequential composition

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

is shown in Fig. 12.

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. Fig. 13 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 second last step of Fig. 13, the justification is proved as follows:

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 $\underline{b}, \underline{x}, t$
require $x \neq Void$
ensure $b = x$

$m(x : INTEGER)$
modifies a, t
ensure $a = x$
end

$Body$ of routine r in class C
$\rightarrow$ $<$ **create** Defn. 4.1; composition (3); Defn. 4.4; (19); (5) 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 Defn. 4.4 $>$
$e1.a = 5 \wedge e2.a = 6 \wedge \underline{e1}[] \wedge \underline{e2}[];$
$e1.\underline{b}, \underline{e2}, t : \langle\!|\ e2 \neq Void, e1.b = e2\ |\!\rangle;$
$e3 := e1.a.max(e1.b.a)$
$\rightarrow$ $<$ $\underline{e2}[] \rightarrow e2 \neq Void$ and (5) $>$
$\underline{e1}[] \wedge \underline{e2}[] \wedge e1.a = 5 \wedge e1.b = e2 \wedge e2.a = 6;$
$e3 := e1.a.max(e1.b.a)$
$\rightarrow$ $<$ assignment (3) , 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 (3) ; one-point; (15): $e1.b = e2 \rightarrow e.1b.a = e2.a >$
$\underline{e1}[] \wedge \underline{e2}[] \wedge e1.a = 5 \wedge e1.b = e2 \wedge e2.a = 6 \wedge$
$e3 = 5.max(6)$
$\rightarrow$ $<$ (24) and postcondition of max in Fig. 9 $>$
$ensure\ clause\ of\ r$

Fig. 11. Example of reasoning using reference types

$$
\begin{array}{ll}
 & P \\
= & \quad \langle \text{ definition } \rangle \\
 & i, Result : (\!| \, s.good,\ i = \mathbf{old}\ s.lower \wedge Result = s.item(i) \, |\!) \\
= & \quad \langle \text{ textual substitution [GS93] } \rangle \\
 & i, Result : (\!| \, s.good[i := s.lower],\ (i = \mathbf{old}\ i \wedge Result = s.item(i))[\mathbf{old}\ i := \mathbf{old}\ s.lower] \, |\!) \\
= & \quad \langle \text{ simple substitution Rule 3.1 } \rangle \\
 & i := s.lower;\ Result : (\!| \, s.good,\ Result = s.item(i) \, |\!) \\
\sqsubseteq & \quad \langle \text{ (26) and (24): } s.good \rightarrow s \neq Void \wedge s.valid\_index(s.lower); \text{ pre. weakening } \rangle \\
 & i := s.lower;\ Result : (\!| \, s \neq Void \wedge s.valid\_index(s.lower),\ Result = s.item(i) \, |\!) \\
\sqsubseteq & \quad \langle \text{ assigned query call Defn. 4.3 } \rangle \\
 & i := s.lower;\ Result = s.item(i)
\end{array}
$$

Fig. 12. Proof of refinement for loop initialization

$$
\begin{array}{ll}
 & W \\
\sqsubseteq & < textual\ substitution\ [\text{GS93}] > \\
 & i, Result : (\!| \, (s.good \wedge s.valid\_index(i-1) \wedge i-1 < s.upper)[i := i+1], \\
 & \qquad (i = \mathbf{old}\ i \wedge \\
 & \qquad Result = \mathbf{old}\ Result \uparrow s.item(i))[\mathbf{old}\ i := \mathbf{old}\ i+1] \, |\!) \\
\sqsubseteq & < \text{simple substitution Rule 3.1} > \\
 & i := i+1; \\
 & Result : (\!| \, s.good \wedge s.valid\_index(i-1) \wedge i-1 < s.upper, \\
 & \qquad i = \mathbf{old}\ i \wedge Result = \mathbf{old}\ Result \uparrow s.item(i) \, |\!) \\
\sqsubseteq & < s.good \wedge s.valid\_index(i-1) \wedge i-1 < s.upper \rightarrow s \neq Void \wedge s.valid\_index(i); \\
 & \ precondition\ weakening\ (Morgan\ Rule\ 1.2\ [\text{Mor94}]) > \\
 & Result : (\!| \, s \neq Void \wedge s.valid\_index(i), \\
 & \qquad Result = \mathbf{old}\ Result \uparrow s.item(i) \, |\!) \\
\sqsubseteq & < \ proof\ described\ in\ Fig.\ 14\ > \\
 & Result := Result.max(s.item(i))
\end{array}
$$

Fig. 13. Refinement tree for the loop body

$$
\begin{array}{ll}
 & s.good \wedge s.valid\_index(i-1) \wedge i-1 < s.upper \\
\Rightarrow & \quad \langle \ (24) \ \rangle \\
 & s.good \wedge s.lower \leq i-1 \leq s.upper \wedge i-1 < s.upper \\
\Rightarrow & \quad \langle \ i-1 < s.upper \rightarrow i \leq upper;\ s.lower \leq i-1 \rightarrow s.lower \leq i \ \rangle \\
 & s.good \wedge s.lower \leq i \leq s.upper \\
\Rightarrow & \quad \langle \text{ propositional logic } \rangle \\
 & s.valid\_index(i) = s.lower \leq i \leq s.upper \rightarrow s.good \wedge s.valid\_index(i) \\
\Rightarrow & \quad \langle \ 24 \ \rangle \\
 & s.good \wedge s.valid\_index(i) \\
\Rightarrow & \quad \langle \text{ by definition of } s.good \ \rangle \\
 & s \neq Void \wedge s.valid\_index(i)
\end{array}
$$

The most interesting obligation comes in the last step of the refinement, where we want to introduce a nested query call in an assignment. The proof is shown in Fig. 14.

$$
\begin{array}{ll}
 & Result := Result.max(s.item(i)) \\
= & < \text{meaning of a nested query call (23) in Defn. 4.3} > \\
 & \textbf{local } v : INTEGER;\ v := s.item(i);\ Result := Result.max(v) \\
= & < \text{semantics of query assignments Defn. 4.3} > \\
 & \textbf{local } v : INTEGER; \\
 & \quad v : (\!| s \neq Void \wedge s.valid\_index(i), v = \textbf{old } s.item(i) |\!); \\
 & \quad Result : (\!| true, Result = \textbf{old } Result \uparrow v |\!) \\
\rightarrow & < \text{sequential composition (3)} > \\
 & Result : (\!| s \neq Void \wedge s.valid\_index(i), Result = \textbf{old } Result \uparrow s.item(i) |\!)
\end{array}
$$

Fig. 14. Proof of final refinement step

References

[AC96] M. Abadi and L. Cardelli.: *A Theory of Objects*, Springer-Verlag, 1996.

[Abr96] J.-R. Abrial.: *The B-Book*, Cambridge University Press, 1996.

[BH97] P.G. Bancroft and I.J. Hayes. Type extension and refinement. In *Proc. Formal Methods Pacific (FMP'97)*, Springer-Verlag, 1997.

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

[CS99] A. Cavalcanti, A. Sampaio, J. Woodcock.: An inconsistency in procedures, parameters, and substitution in the refinement calculus. *Science of Computer Programming*, **33** (87-96), 1999.

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

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

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

[DR94] R. Duke, G. Rose, and G. Smith.: Object-Z: a Specification Language Advocated for the Description of Standards. Technical Report 94-45, Software Verification Research Center, University of Queensland, December 1994.

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

[GS93] D. Gries and F. Schneider.: *A Logical Approach to Discrete Math*, Springer-Verlag, 1993.

[Gri97] A. Griffiths.: Modular Reasoning in Object-Z. Technical Report 97-28, Software Verification Research Center, University of Queensland, August 1997.

[Heh93] E.C.R. Hehner.: *A Practical Theory of Programming*, Springer-Verlag, 1993.

[Hoa69] C.A.R. Hoare.: An Axiomatic Basis for Computer Programming. *Comm. ACM* **12**(10), October 1969.

[Hoa85] C.A.R. Hoare.: *Communicating Sequential Processes*, Prentice-Hall, 1985.

[JM97] J.-M. Jezequel and B. Meyer.: Design-by-Contract: The Lessons of the Ariane 5. *IEEE Computer* **30**(2), January 1997.

[Jon90] C.B. Jones.: *Systematic Software Development Using VDM* (Second Edition), Prentice-Hall, 1990.

[KM95] S. Kent and I. Maung.: Quantified Assertions in Eiffel. In *Proc. TOOLS Pacific 1995*, Prentice-Hall, 1995.

[Lan95] K. Lano.: *Formal Object-Oriented Development*, Springer-Verlag, 1995.

[Lea97] G. Leavens.: Larch/C++ Reference Manual Version 5.14. Available at www.cs.iastate.edu/∼leavens/larchc++.html. October 1997.

[LB00] G. Leavens, A. Baker, and C. Ruby.: Preliminary Design of JML: a Behavioural Interface Language for Java. Technical Report #98-06j, Department of Computer Science, Iowa State University, Revised May 2000.

[Lei95] K.R.M. Leino.: Toward Reliable Modular Programs. Ph.D. Thesis, Department of Computer Science, California Institute of Technology, 1995.

[Lei97] K.R.M. Leino.: Ecstatic: an object-oriented programming language with an axiomatic semantics. In *Proc. Fourth International Workshop on Foundations of Object-Oriented Languages*, January 1997.

[LW94] B. Liskov and J. Wing.: A Behavioural Notion of Subtyping. *ACM Trans. Prog. Lang. Sys.* 16(6), November 1994.

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

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

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

[Mor94] C.C. Morgan.: *Programming from Specifications* (Second Edition), Prentice-Hall, 1994.

[PO99] R.F. Paige and J.S. Ostroff.: Developing BON as an Industrial-Strength Formal Method. In *Proc. World Congress on Formal Methods (FM'99): Volume I*, LNCS 1708, Springer-Verlag, September 1999.

[PO01] R.F. Paige and J.S. Ostroff.: Metamodelling and Conformance Checking with PVS. In *Proc. Fundamental Aspects of Software Engineering (FASE'01)*, LNCS 2029, Springer-Verlag, April 2001.

[Par72] D. Parnas.: On the Criteria to be Used in Decomposing Systems into Modules. *Comm. ACM* **15**(12), December 1972.

[Par92] D. Parnas.: Tabular Representation of Relations. CRL Report 260, Communications Research Laboratory, McMaster University, October 1992.

[RG00] M. Richters and M. Gogolla.: Validating UML Models and OCL Constraints. In *Proc. Unified Modeling Language 2000*, Lecture Notes in Computer Science 1939, Springer-Verlag, October 2000.

[RE98] W.-P. de Roever and K. Englehardt.: *Data Refinement: Model-oriented Proof Methods And Their Comparison*, Cambridge University Press, 1998.

[RJ99] J. Rumbaugh, I. Jacobson, and G. Booch.: *The Unified Modeling Language Reference Manual,* Addison-Wesley, 1999.

[Smi95] G. Smith.: Reasoning about Object-Z Specifications. In *Proc. Asia-Pacific Software Engineering Conference 1995*, IEEE Press, 1995.

[Smi99] G. Smith.: The Object-Z Reference Manual, Kluwer, 1999.

[Spi92] J.M. Spivey.: *The Z Reference Manual*, Second Edition, Prentice-Hall, 1992.

[WN95] K. Walden and J.-M. Nerson.: *Seamless Object-Oriented Software Architecture*, Prentice-Hall, 1995.

[Wor94] J. Wordsworth.: *Software Development with Z,* Addison-Wesley, 1994.