An Object-Oriented Refinement Calculus

Richard F. Paige and Jonathan S. Ostroff
Department of Computer Science, York University, Toronto, Ontario M3J 1P3,
Canada. {paige,jonathan}@cs.yorku.ca*

August 28, 2000

Abstract

Design-by-contract has been used in the BON/Eiffel setting as an industrial-strength technique for building reliable, reusable, and maintainable software systems. We show that it provides a basis for refining object-oriented specifications to programs on an industrial scale. We present a collection of algorithm refinement rules, in particular, new rules for introducing *feature calls* (which are the basis of object-oriented computing) that can be used to refine a specification in BON to an immediately executable and correct program in Eiffel. We show how refinement of large systems can be decomposed into an arbitrary number of small steps. And we describe how automated support for such a process can be developed based on existing tools.

1 Introduction

A key goal of formal methods has always been to specify, design and verify large programs. While important theoretical gains and some practical benefits have been achieved, the actual application of these methods to industrial-strength software development has been for the most part limited to critical components of fairly small subsystems.

Object-oriented (OO) software development has been suggested as a useful and important technique for building large, reliable, and maintainable software systems [Lan95, Mey97]. However, the most popular formal notations and methods such as Z [Spi92], VDM [Jon90], CSP [Hoa85] and B [Abr96] do not apply directly to OO software development, since they lack fundamental features like the ability to specify classes, inheritance, and feature redefinition. OO extensions of these languages, such as Object-Z [DR94] and VDM++ [Lan95], while eliminating many of these problems, do not have realistic associated implementation languages (and thus further translation is necessary to produce executable code), nor have they completely worked out all the details inherent in refining object-oriented specifications to programs. 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 and language already available that casts many of the benefits of conventional formal methods — and refinement in particular — into the OO realm. This method and language is applicable to specification and to the development of immediately executable code. The method is BON [WN95] and the language is Eiffel [Mey92]. Formal methodologists have paid little attention to BON/Eiffel despite the fact that it appears to be an ideal platform for making formal methods directly usable in large-scale software development.

A common element shared by the BON method and Eiffel language 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*. These contracts precisely describe the services that classes offer to clients, and the obligations clients have when using the services.

Contracts (a) describe the class interface, i.e., 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 sub-contracting so that the

*The authors thank the National Sciences and Engineering Research Council of Canada for their support.

meaning of a redefined routine under inheritance remains consistent[1]; and (f) provide documentation to both clients and suppliers of classes. By writing well-designed preconditions, postconditions and invariants, as well as carefully choosing names for classes and routines, we get the *self-documenting principle* — the documentation of a class is developed hand-in-hand with the class and is stored together with the class; documentation is automatically extracted by tools from the class text itself at various levels of abstraction.

What is missing from DbC in BON/Eiffel is the notion of refining an abstract specification of a class down to an immediately executable Eiffel program with a guarantee that the program satisfies the specification.

What is missing in conventional formal methods such as Z, B or tabular specifications [Par92] are methods for structuring and packaging large specifications, in order to promote reusability and maintainability. Conventional methods have notions of a schema or module, but not the benefits provided by object orientation, viz., the structuring of large systems via classes and the client-supplier and inheritance relationships between classes (described in more detail in the sequel). Object-oriented extensions of formal methods such as Object-Z, VDM++, and Larch/C++ do not have comprehensive refinement rules that can be used to transform specifications into implemented code in an actual OO programming language.

The purpose of this paper is to use the object-oriented features of BON/Eiffel to help structure the specification of large systems with the addition of refinement rules for developing implemented Eiffel code from BON specifications, and in particular refinement rules for feature calls, which is the basis for object-oriented computing. The development of code is *compositional* — the development 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. This compositional property means that the process can potentially be applied in the development of large industrial-scale software.

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 compositionality, we need only the contracts, and not the implementations, of a few classes to refine the class specification of C to an executable program. This is the OO version of the compositionality 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, the compositionality principle applies, and we can refine a class by using only the contracts of related classes. Instead of refining a large OO program all at once, we will have broken down the task into separate classes and features that are annotated with contracts.

The main contribution of OO refinement is its structuring of specifications and programs into parts — viz., classes — that enforce information hiding and encapsulation. The refinement method described in this paper uses the class as the atomic notion of module to which composition, and thereby refinement, is applied. This is what makes OO the ideal setting in which to carry out refinement.

As will be explained in the sequel, BON/Eiffel has *expanded* (sometimes referred to as "subobject") types as well as *reference* types. In this paper, we limit ourselves to expanded types only, so as to concentrate on getting the refinement rules right: we focus on the development of refinement rules for transforming BON specifications into Eiffel programs that are suitable for mechanization using existing tools, e.g., in an industrial-strength theorem prover like PVS [CO95]. Extension of the rules to reference types will be carried out once the rules for expanded types have been developed and mechanized. For the extension to reference types, we can make us of the work on Object-Z [DR94], JML [LB00], and the Larch formalization of the Eiffel reference type (which is identical to the BON reference type) in [KM95]. Formalizing reference types will also benefit from our work on automating the refinement calculus (see Section 7 and also [PO99]).

[1] If a client of class $RECTANGLE$ (which inherits from class $POLYGON$) calls a feature to calculate the perimeter, then we want to ensure that $RECTANGLE$ does not redefine *perimeter* to calculate the *area* instead. Redefinition could change the implementation of a feature but not its essential meaning.

1.1 Organization of the paper

We first present an overview of design-by-contract, illustrating its principles and its use in the development of reliable object-oriented software. Then we provide an overview of the Business Object Notation (BON), which supports DbC. We summarize BON's features, and provide a notion of the meaning of a contract, due to Meyer [Mey92]. We also summarize the notation that we will use throughout the remainder of the paper. Sections 4 and 5 contain the main contributions of the paper. In Section 4, we explain how to carry out algorithm refinement in BON, and provide a collection of refinement rules. These rules include novel techniques specifically for object-oriented computing, in particular, for introducing *feature calls*, the basis of object-oriented specification and programming. In Section 5, we explain the compositionality of refinement in BON, and provide a process for refining a BON specification into executable Eiffel code. We illustrate the process with a short example, in Section 6. In Section 7, we discuss automation, and our goal of supporting refinement and verification with BON/Eiffel using PVS. Finally, in Section 8, we discuss related work.

2 Design-by-Contract

An object-oriented system (by which we mean a specification or a program) consists of a number of classes connected by relationships. Each class consists of a number of *features*, which may be attributes (state) or routines (computations). The premise of design-by-contract is that each routine of a class should have a contract. The contract expresses both constraints on when a routine can be used, and the results that calling a routine will produce, e.g., a returned value, or a change to the state of an invoking object. In effect, the contract expresses *obligations* that are placed on any client — any other feature that uses the routine — as well as the *benefits* that the client can rely on by calling the routine. Obligations on the client can be expressed as a *precondition* to a routine, while benefits to the client are expressed as a *postcondition*. Dually, the author of a routine is obliged to make sure that the postcondition is satisfied by the routine implementation, and benefits by not having to deal with cases other than those covered by the precondition.

Consider the $push$ procedure of a $STACK$ class. As Fig. 1 shows, a client of $push$ is obliged to call the routine only when the stack is not full, and benefits by having the argument of the call put on top of the stack. Accordingly, the author of $STACK$ is obliged to make sure that their implementation of $push$ puts the argument to the call atop the stack, and benefits from the precondition by not having to treat the case when the stack is full.

	OBLIGATION	**GAIN**
CLIENT	call *push(x)* when the Stack isn't *full*	gets *x* added to the top of the Stack
CLASS AUTHOR	make sure that *x* is on the top of the Stack	need not treat the case when Stack is *full*

Figure 1: Benefits and obligations for a *push* feature of a stack class

A simple extension of DbC is to allow classes themselves to be annotated with assertions that apply to all features of the class. Such assertions are called *class invariants*.

Design-by-contract has been suggested as useful for a number of purposes.

- *Reliability.* Reliability is concerned with the robustness of software (i.e., reaction to abnormal conditions, such as unexpected environmental changes or hardware faults) and its correctness (i.e., does a piece of software satisfy its specification). DbC can make software more robust by explicitly delimiting what is an abnormal condition (an unsatisfied precondition or postcondition) and who is responsible for dealing with the condition. For example, if a precondition is unsatisfied, it is due to a bug in the client, and therefore the designer of the client must deal with the bug.

- *Reusability.* By precisely specifying contracts for class features, the conditions under which a feature, and thereby a class, can be reused can be explicitly codified. A discussion of some of the benefits of contracts in a reuse situation can be found in [JM97].

- *Maintainability.* By keeping the contracts in place with their implementations, a maintainable software system can be produced. Keeping contracts with implementations helps to keep specifications and programs in synch over time. Having contracts for features can help in the process of finding errors in programs, thus aiding in the process of corrective maintenance.

3 The Business Object Notation

BON is an object-oriented method possessing a recommended process as well as a graphical notation for specifying object oriented systems. The notation provides mechanisms for specifying inheritance and client-supplier relationships between classes, and has a small collection of techniques for expressing dynamic relationships. The notation also includes an *assertion language*; the method is predicated on the use of this assertion language for specifying contracts of routines and invariants of classes.

BON is designed to support three main techniques: seamlessness, reversibility, and design-by-contract. Seamlessness means that the same modeling abstractions can be used throughout the software development process, including both specification and implementation. This allows the direct mapping of abstractions that are used in specification to abstractions used when coding. Reversibility means that models can be automatically produced from programs; BON models can automatically be extracted from Eiffel programs. BON provides a small collection of powerful specification features that guarantee seamlessness and full reversibility when used with Eiffel.

The fundamental specification construct in BON is the *class*. In BON, a class is both a module and a type. A BON class has a name, an optional class invariant, and a collection of features. A feature may be a *query* — which returns a value and does not change the system state, i.e., a side effect-free function — or a *command*, which does change system state but returns nothing. BON, according to [WN95], does not include a separate syntax for the notion of *attribute*. Conceptually, an attribute should be viewed as a query returning the value of some hidden state information. For the purposes of this paper, we will define an attribute as a *parameterless query* without a contract.

In BON, all features are typed; routines may have (optional) domains and (in the case of queries) ranges. In general, types may be *reference* or *expanded*. Where a reference type is expected, an address is to be provided; where an expanded type is expected, an object of that type is expected.

Fig. 2 contains a short example of a BON graphical specification of the interface of a class $CITIZEN$; the specification uses both reference types and expanded types (though we will consider only expanded types in Section 3.1 and thereafter). Class features are in the middle section of the diagram (there may be an arbitrary number of sections, each annotated with preface listing the clients that may access the features in the section). Routines may optionally have behavioral specifications, written in the BON assertion language in a pre- and postcondition form (in postconditions, the keyword **old** can be used to refer to the value of an expression when the routine was called; similarly, the implicitly declared variable $Result$ can be used to constrain the value returned by a query). Procedure specifications (and loop invariants, see Section 4) may be given *frames*, indicating those attributes that may be changed by the specification. Procedure $divorce$ shows an example of a frame. An optional class invariant is at the bottom of the diagram. The 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); thus, 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.

The basic form of a BON assertion is

$$\forall\, x : T \mid R \bullet P$$

where variable x of type T is the bound variable, R is the domain restriction, and P is the propositional part.

In Fig. 2, class $CITIZEN$ has seven queries ($name$ and sex, for example, are considered to be attributes) and one command. For example, $single$ is a query (which results in a $BOOLEAN$), while $divorce$ is a parameterless command that changes the state of an object. Class $SET[G]$ is a generic predefined class with generic parameter G and the usual operators (e.g., $\in$, add). The class $SET[CITIZEN]$ thus denotes a set of objects each of type $CITIZEN$.

Short forms of assertions are permitted. For example, consider a query $children : SET[CITIZEN]$. Then $\forall\, c \in children \bullet P$ is an abbreviation of $\forall\, c : SET[CITIZEN] \mid c \in children \bullet P$. The "it holds" operator $\bullet$ is right associative. The last invariant of Fig. 2 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 exactly two parents. An advantage of the object-oriented BON style of specification is that as the class is enriched with new features (e.g., $children$ and $single$), the new features become part of the specification language used for contracts.

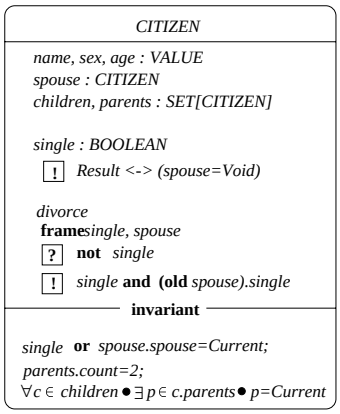

Figure 2: A citizen class in BON

BON specifications normally consist of multiple classes that interact via two kinds of different relationships.

- **Inheritance:** a child class inherits behavior from one or more parent classes. Inheritance is the subtyping relationship: everywhere an instance of a parent class is expected, an instance of a child class can appear. The inheritance relationship is drawn between classes $ANCESTOR$ and $CHILD$ in Fig. 3, with the arrow directed from the child to the parent class. In this figure, we have drawn the classes in their *compressed forms*, as ellipses, wherein details of the classes' interfaces are concealed from the reader.

- **Client-supplier:** there are two basic client-supplier relationships, association and aggregation, which are used to specify the *has-a* or *part-of* relationships between classes, respectively. Both relationships are directed, from a *client* class to a *supplier* class. With association, the client class has an attribute that is a reference to an object of the supplier class. With aggregation, the deletion of an instance of a client class also implies the deletion of the corresponding instance of the supplier class; thus, the client class has an attribute that is an object of the supplier class. Aggregation arrows visually depict the notion of *expanded type* discussed earlier. The aggregation relationship is drawn between classes $CHILD$ and $SUPPLIER$ in Fig. 3: we say that $CHILD$ is a client of $SUPPLIER$. In this paper, we will only use the aggregation relationship; thus, we will not formalize a notion of reference type.

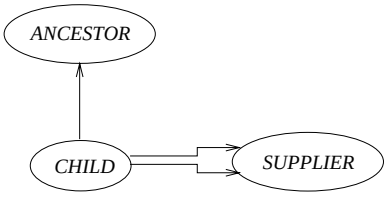

Figure 3: Class relationships in BON

Information hiding is described by dividing the features of a class interface into sections, where each section is prefaced by a list of client classes that may use the features contained in the section. Private features are described in sections prefaced by $NONE$; public features are in sections prefaced by ANY. A feature accessible only to classes A, B, and C will be in a section prefaced by the list A, B, C.

Often, especially when providing a view of a class suitable for specific clients, a *flat form* [Mey97] of a class is provided. The flat form of a class consists of all features in the class and all inherited features, as well as their contracts. As we shall see, the flat form will prove to be key to the development of a compositional refinement process.

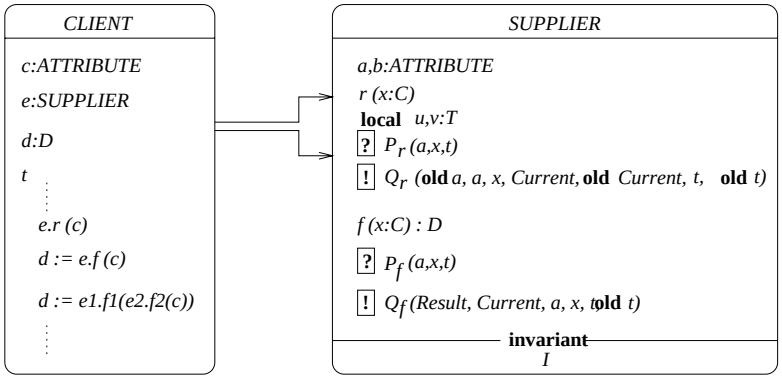

Figure 4: Class diagram for terminology and context

3.1 Semantics and terminology

The formal meaning of routines with contracts has been given in [Mey92, Mey97] in terms of Hoare tuples. We shall instead provide an equivalent meaning in terms of predicates. The reason for using predicates instead of Hoare tuples is that the former leads to simpler refinement rules for developing code smoothly from specifications. The reason for using predicates over, e.g., weakest preconditions or predicate pairs, is that predicates are immediately expressible in the specification language of the PVS theorem prover, and our goal is the construction of a refinement calculus that is amenable to automated support using existing tools.

Before presenting the meaning of a routine with a contract, we define some terminology and notation that we will use throughout the remainder of the paper.

A *double-state formula* is a predicate in a prestate and a poststate. An example, in BON syntax, is $x = \mathbf{o}ld\ x + 1$, which is a predicate in the free variables x and $\mathbf{o}ld\ x$. Postconditions in BON are expressed using double-state formulae. A *single-state formula* (or condition) is a predicate in a single state. An example is $x > 0$, which is a predicate in x. Preconditions and class invariants in BON are single-state formulae.

Consider the class diagram shown in Fig. 4. We define the remaining terminology that we use throughout the paper in terms of this diagram. We assume that for both $CLIENT$ and $SUPPLIER$, the interfaces shown in Fig. 4 are the *flat* forms, i.e., all inherited features are shown.

Class $CLIENT$ of Fig. 4 possesses a parameterless procedure t. Within t, we have shown three statements: the targeted procedure call $e.r(c)$, the assignment statement $d := e.f(c)$; and the assignment with nested calls $d := e1.f1(e2.f2(c))$; we append numbers to the entity e and the function f when we need to talk about different versions of entities and functions. Thus, $e1$ and $e2$ are different entities of type $SUPPLIER$. r is a procedure of class $SUPPLIER$, while f (in all its versions) is a function of $SUPPLIER$ that returns a result of type D. Class $SUPPLIER$ includes contracts for both r and f. In particular, these contracts show that the preconditions of the routines, namely P_r and P_f, are given in terms of attribute a, time variable t (defined in the sequel), and parameter x (we discuss the relationship between attributes and $Current$ shortly). Note that P_r and P_f are not functions with arguments: P_r, for example, is an assertion that makes reference to a and x. The postcondition of r, namely Q_r, is a double-state formula that makes reference to **old** a, a, **old** t, t, **old** $Current$, $Current$ and x. As we will discuss shortly, there is a direct relationship between the attribute a and the variable $Current$. The postcondition of f is a single-state formula in $Result$ (an implicit variable that is local to the function, whose value is returned by the function call upon termination), $Current$, a, t, **old** t, and x.

We treat the $Current$ object associated with a class as an instance of a record containing one field for each attribute of the class. Fields of the $Current$ object can be accessed using the dot notation, e.g., $Current.a$. We allow the shorthand, used in object-oriented programming languages like Eiffel and Java, of dropping $Current$ when accessing features within their class. Thus, an occurrence of the attribute a in the precondition of routine r is a shorthand for $Current.a$. This short-hand may be used in a frame; thus, a procedure annotated with $\mathbf{frame}\ a$ is a short-hand for $\mathbf{frame}\ Current.a$. We allow the short-hand $\mathbf{frame}\ Current$ which is a frame consisting of all attributes in the class.

We denote the attributes of class $SUPPLIER$ by the bunch $\mathbf{state}\ SUPPLIER$; this bunch includes all attributes in the *flat* form of $SUPPLIER$. For more on bunches, see [Heh93]; informally, a bunch is the content of a set. In this

example, $\mathbf{state}\ SUPPLIER = a, b$. Each routine r of a class may introduce a bunch of local variables $\mathbf{var}\ r$. For the example in Fig. 4, $\mathbf{var}\ r = u, v$. Note that normally the variables local to a routine would not be revealed in the class interface. We show them for the routine r in Fig.4 only to define syntax and our terminology. For each routine r with a contract, we can obtain the precondition by writing $\mathbf{pre}\ r$, and the postcondition by writing $\mathbf{post}\ r$. In both cases, the precondition and postcondition each include the class invariant. Finally, we denote the invariant of class $SUPPLIER$ by $\mathbf{invariant}\ SUPPLIER$.

A special double-state formula associated with routine r in class $SUPPLIER$ is a *specification*; it is a predicate with free variables $\mathbf{state}\ SUPPLIER$, where a free variable may optionally be prefixed with the keyword **old**. This formula, which we shall write as $\mathbf{spec}\ r$, formally describes the meaning of the routine based on its contract (see Table 1 for the differences between a specification and a postcondition).

It is convenient to use Morgan's *specification statement* syntax for writing specifications [Mor94]. Morgan's syntax allows the separation of the precondition from postcondition, and introduces the notion of a *frame* — a bunch of variables that may be changed — for a specification. A specification statement is of the form $w : \langle\!|\ S, D\ |\!\rangle$, where w is the frame, S a precondition, and D a postcondition. We use the parentheses $\langle\!|$ and $|\!\rangle$ instead of the more common [and] to distinguish between specification statements and the notion of textual substitution [GS93], which we shall use later on.

In [Mor94], specification statements are given a semantics using weakest preconditions. We instead use specification statements as a syntactic convenience, and give their meaning using the predicative notation of Hehner [Heh93]. We explicitly include time in the semantics, via time variables $\mathbf{old}\ t$ and t, following Hehner: $\mathbf{old}\ t$ represents the start time of a computation, while t represents the time at which the computation finishes – nontermination is represented as the case where $t = \infty$. Postconditions of specification statements may make mention of the time variables; if they do not, the most that can be said about time is that it does not decrease. The meaning of a specification statement is as follows.

Definition 1 *Meaning of a specification statement. Let the state σ consist of disjoint bunches of variables w and x (where w or x, but not both, may include t). Let $S(\sigma)$ be a single-state formula in σ, and $D(\mathbf{old}\ \sigma, \sigma)$ a double-state formula in σ and $\mathbf{old}\ \sigma$. Then*

$$w : \langle\!|\ S, D\ |\!\rangle \quad \widehat{=} \quad (\mathbf{old}\ S) \rightarrow (D \wedge x = \mathbf{old}\ x \wedge t \geq \mathbf{old}\ t)$$

The predicate on the right hand side of Definition 1 is a double-state formula; it is a specification because it characterizes a set of observations, i.e., the set of pre- and poststate pairs that satisfy the predicate. These observations describe the system under description. If the precondition **old** S is not satisfied by an observation, then any arbitrary (terminating or non-terminating) behaviour is permitted. If the precondition is satisfied, then the postcondition D must be established, x may not be changed, and time may not decrease. D itself may further constraint time, e.g., to require a specific time bound for a computation to complete.

Here is an example of a specification: the value of f is to be set to the factorial of n, and no more than n units of time can be taken.

$$f, t : \langle\!|\ true, f = n! \wedge t = \mathbf{old}\ t + n\ |\!\rangle$$

Using Definition 1, we see that the specification $w, t : \langle\!|\ S, true\ |\!\rangle$ means *chance* behaviour, i.e., any prestate satisfying S has a corresponding poststate that may arbitrarily change the frame w, and may or may not terminate. The specification $w, t : \langle\!|\ false, D\ |\!\rangle$ means *chaos*, i.e., it reduces to the predicate $true$ which denotes totally arbitrary behaviour — any change can be made to the variables including the frame, and the execution may be non-terminating. A *miracle*, i.e., no satisfying behaviour can be specified with $w, t : \langle\!|\ true, false\ |\!\rangle$, which reduces to the predicate $false$. We can now use specifications to give the meaning of a routine with a contract.

Definition 2 *Meaning of a routine with a contract. The meaning of routine r in class C is given by the predicate* $\mathbf{s}pec\ r$.

$$\mathbf{s}pec\ r \quad \widehat{=} \quad t, \mathbf{state}\ C : \langle\!|\ \mathbf{pre}\ r,\ \mathbf{post}\ r\ |\!\rangle$$

Table 1: Notation summary with reference to Fig. 4

state $SUPPLIER$	$\widehat{=}$	a, b
var r	$\widehat{=}$	u, v
invariant $SUPPLIER$	$\widehat{=}$	I
$a : \langle\!\langle\ S, D\ \rangle\!\rangle$	$\widehat{=}$	$(\mathbf{old}\ S) \rightarrow (D \wedge b = \mathbf{old}\ b \wedge t \geq \mathbf{old}\ t)$
pre r	$\widehat{=}$	$P_r(a, x, t) \wedge I$
post r	$\widehat{=}$	$Q_r(\mathbf{old}\ a, \mathbf{old}\ t, a, t, x, Current, \mathbf{old}\ Current) \wedge I$
$P[x_1 := e_1, x_2 := e_2]$	$\widehat{=}$	contextual substitution [GS93]
old a	$\widehat{=}$	a variable denoting the value of attribute a in the prestate of a feature call
old S	$\widehat{=}$	a predicate obtained from S by prefacing all free variables of S by **old**

We shall use this semantics in developing a collection of refinement rules in Section 4. Table 1 gives a summary of the notation and terminology we have just presented, in the context of the class diagram in Fig. 4.

(Note that t is not included in **state** $SUPPLIER$; time variables belong to no class.) We now show how to carry out refinement with BON and Eiffel, and in the process demonstrate that efinement in BON is applicable to large specifications. We do this by first showing how to refine a single BON class to an Eiffel program. Then, we present a compositional process that shows how to refine classes related by aggregation and inheritance through to an Eiffel program.

4 The Compositional Process for Features

Let us consider the case where we want to algorithmically refine a single BON class, consisting of a number of features with contracts, to an Eiffel program. This is a routine-by-routine process, where we must refine each BON routine into an Eiffel routine. The notion of algorithm refinement that we use is due to Hehner [Heh93]; refinement is just boolean implication. The refinement relation that we define can be applied, in general, to any double-state formulae. We will apply it to specifications written in the form $w : \langle\!\langle\ S, D\ \rangle\!\rangle$.

Definition 3 *Refinement. Let the state σ include variables w, and let S_1 and S_2 be single-state formulae on $\sigma, \mathbf{old}\ t$, and let D_1 and D_2 be double-state formulae on $\sigma, \mathbf{old}\ t, t$, and $\mathbf{old}\ \sigma$. The refinement relation, $\sqsubseteq$, on specifications is:*

$$w : \langle\!\langle\ S_1, D_1\ \rangle\!\rangle \quad \sqsubseteq \quad w : \langle\!\langle\ S_2, D_2\ \rangle\!\rangle \quad \widehat{=} \quad w : \langle\!\langle\ S_1, D_1\ \rangle\!\rangle \quad \leftarrow \quad w : \langle\!\langle\ S_2, D_2\ \rangle\!\rangle$$

The proof obligation to be discharged in such a refinement step is

$$((\mathbf{old}\ S_2) \rightarrow (D_2 \wedge x = \mathbf{old}\ x)) \rightarrow ((\mathbf{old}\ S_1) \rightarrow (D_1 \wedge x = \mathbf{old}\ x))$$

The statement of Definition 3 appears to imply that the two specifications in a refinement need to have identical frames. The rule below (derived trivially from Definition 1) allows the frame to be expanded or collapsed. This rule is similar to one found in [Mor94].

Rule 4.1 *Frame Change Rule. Let w and x be disjoint bunches of variables.*

$$w : \langle\!\langle\ S, D\ \rangle\!\rangle \quad = \quad w, x : \langle\!\langle\ S, D \wedge x = \mathbf{old}\ x\ \rangle\!\rangle$$

Our goal is to refine BON specifications to Eiffel programs. We must therefore show how to refine BON routine specifications into Eiffel routine implementations. The refinement process thus starts with a BON routine r with specification $\mathbf{spec}\ r$, and ends in an Eiffel routine implementation $Prog$. Along the way, intermediate double-state formulae in the form $w : \langle\!\langle\ S, D\ \rangle\!\rangle$ may be introduced. The process can be depicted as

$$\begin{array}{ll} & \mathbf{spec}\ r \\ \sqsubseteq & w : \langle\!\langle\ S_1, D_1\ \rangle\!\rangle \\ \sqsubseteq & \mathbf{if}\ b\ \mathbf{then}\ w : \langle\!\langle\ S_2, D_2\ \rangle\!\rangle\ \mathbf{else}\ \ldots \end{array}$$

Refinement is the process of implementing a specification. This process may not always succeed, because unimplementable specifications may be expressible in the specification language. In BON, refinement involves implementing routines of classes. The implementability of a routine can be shown in one of two ways. One approach is to actually succeed, using the rules, in producing an Eiffel program. However, we might like to know in advance whether or not our refinement attempts have the potential to succeed. We thus might like to know if our specifications have a valid poststate, i.e., for a routine r, show:

$$\mathbf{implementable\ spec}\ r \ \widehat{=}\ \forall\, \mathbf{old}\ \sigma \bullet \exists\, \sigma \bullet (spec\ r \wedge t \geq \mathbf{old}\ t) \tag{1}$$

where the state $\sigma = \mathbf{state}\ C, t$.

To make refinement more convenient to carry out, it is customary to define a collection of refinement rules, showing how to refine specifications to programs. We summarize a collection of rules that apply to any double-state formulae (and therefore, to any BON routine). In the following let Q, Q_1, Q_2, and $Body$ be specifications — written as specification statements — defined on variables σ. For the following rules, we let σ consist of the variable x and other variables w; x and w may be local variables or attributes of a class.

The first rule we present, taken from [PO99], allows us to reuse refinement rules of Morgan [Mor94]. Effectively, the rule states that a valid refinement rule in Morgan's calculus is a valid refinement rule in BON. Note that in Morgan's calculus, refinement is defined in terms of weakest preconditions, whereas in our calculus, refinement is boolean implication. This defines an ordering relationship between refinement relations; see [OP00] for further discussion on this issue.

Rule 4.2 *Reuse of Morgan's Refinement Rules (from [PO99]). Let Q_1 and Q_2 be specification statements, and suppose that $Q_1 \sqsubseteq Q_2$ according to Morgan's definition of refinement. Then, under a syntactic translation of the specifications Q_1 and Q_2, $Q_1 \sqsubseteq Q_2$ according to the meaning of refinement in Definition 3.*

This rule is proved in [PO99]. The implication of this rule is that, in our refinements (and particularly, in our example in Section 6) we will be able to freely make use of refinement rules from [Mor94]. One such rule that we will apply in Section 6 is the *weaken precondition rule* of [Mor94].

Let us start by giving rules for introducing imperative language constructs — specifically, the assignment statement, selection, loop, and sequential composition — in a refinement. For all statements but the loop, we define the meaning of the language construct using a specification statement. Then, Definition 3 can be applied directly in introducing language constructs in refinement.

First, we consider how to introduce a simple assignment statement (where the expression does not include a function call) in a refinement. We do this by defining the meaning of the assignment $x := e$ using a specification statement.

Definition 4 *Meaning of $x := e$. Let e be an expanded expression (that does not use function calls[a]) with type compatible [Mey97] with variable x, and let $defined(e)$ be a single-state predicate that defines under what conditions expression e is defined. Then*

$$x := e \quad \widehat{=} \quad x : \langle\!|\ defined(e),\ x = \mathbf{old}\ e\ |\!\rangle$$

[a] Arbitrary expanded expressions with function calls will be treated in Definitions 8–15.

In a refinement, we may want to introduce an assignment statement $x := e$, say, by refining a specification statement $x : \langle\!|\ S, D\ |\!\rangle$. By Definition 3, the proof obligation to be discharged in refining the specification statement by the assignment statement is

$$(\mathbf{old}\ defined(e) \rightarrow x = \mathbf{old}\ e \wedge \mathbf{old}\ S) \rightarrow D$$

The next rule is for introducing a selection, which has the following syntax.

$$\mathbf{if}\ b_1\ \mathbf{then}\ Q_1\ \mathbf{elseif}\ b_2\ \mathbf{then}\ Q_2\ \ldots\ \mathbf{else}\ Q_k\ \mathbf{end}$$

where the b_i are boolean conditions and the Q_i specifications. In the refinement rule, we consider the two-branch case, which generalizes in the obvious way to the multi-branch setting.

Definition 5 *Meaning of selection. Let b be a $BOOLEAN$ expression. Then*

$$\textbf{if } b \textbf{ then } Q_1 \textbf{ else } Q_2 \textbf{ end} \;\widehat{=}\; \sigma : \langle\!| \; true,\ (\textbf{old } b) \wedge Q_1 \vee (\textbf{old } \neg\, b) \wedge Q_2) \; |\!\rangle$$

Selections are typically introduced in a refinement by cases. Thus, if during a refinement we want to refine specification Q by the selection shown in Definition 5, then we will be required to prove the following two obligations.

$$\begin{aligned} Q &\sqsubseteq\ (\textbf{old } b) \wedge Q_1 \\ Q &\sqsubseteq\ (\textbf{old } \neg\, b) \wedge Q_2 \end{aligned}$$

Refinement becomes more complicated when we consider sequencing and repetition, because of the need to introduce intermediate states. First, the rule for sequencing. In the rule, we use the textual substitution notation of Gries and Schneider [GS93]. The syntax $P[x := e]$ reads "replace all free occurrences of variable x in P with e".

Definition 6 *Meaning of sequencing.*

$$Q_1;\ Q_2 \;\widehat{=}\; \exists\, \hat{\sigma} \bullet Q_1[\sigma := \hat{\sigma}] \wedge Q_2[\textbf{old } \sigma := \hat{\sigma}]$$

where σ is all attributes and local variables.

It is now possible to present a useful rule for simplifying a particular kind of sequential composition, where the first specification is an assignment. The *simple substitution rule* is derived from Definition 4 and Definition 6 and is similar to a corresponding rule in [Heh93].

Rule 4.3 *Simple substitution. For any variable x not in bunch w, and expression e whose type conforms to x,*

$$\begin{aligned} x := e;\ w : \langle\!| \; S, D \; |\!\rangle \quad \equiv \quad & w, x : \langle\!| \; S[x := e], \\ & (D \wedge x = \textbf{old } x)[\textbf{old } x := \textbf{old } e\,] \; |\!\rangle \end{aligned}$$

The refinement rule for loops in Morgan [Mor94] and Hehner [Heh93] involve a single-state invariant, whereas we need a more general double-state invariant, e.g., for asserting that the loop does not change certain variables. The corresponding Z rule is stronger than we need. Hence, we introduce a rule for loops in the sequel (see Rule 4.4).

Before we consider the rule for an initialized loop, we introduce some notation, allowing us to talk about the intermediate states that arise through executing a loop. We annotate specifications with primes (e.g., Q') to indicate systematic addition of primes to *free variable names* used within the specification. This notation is borrowed from Z [Wor94]. However, we must make one adaptation in BON, because of its use of **old** to distinguish pre- and poststates: a prime applied to an **old** expression removes the **old** keyword. Here is an example.

$$\begin{aligned} & (x = \textbf{old } y \wedge y = \textbf{old } (x + y))' \\ = \quad & (x' = y) \wedge (y' = (x + y)) \end{aligned}$$

Thus, the prime moves the formula forward one state (i.e., from prestate to poststate). Now we can give a rule for introducing an initialized loop. Loops in Eiffel have the following syntax.

$$\begin{aligned} \mathit{Loop} \;\widehat{=}\; & \textbf{from } \mathit{Init} \\ & \textbf{invariant } I \\ & \textbf{variant } v \\ & \textbf{until } b \\ & \textbf{loop } \mathit{Body} \textbf{ end} \end{aligned}$$

$Init(\mathbf{old}\ \sigma, \sigma)$, $I(\mathbf{old}\ \sigma, \sigma)$, and $Body(\mathbf{old}\ \sigma, \sigma)$ are specifications that will be refined to code. b is a condition, and $v(\sigma)$ is a variant. In Eiffel, the invariant and variant must be boolean and integer expressions, respectively. In refinement, we allow the invariant to include reference to the prestate, via **old**. Thus, the invariant will also be a specification, and so it may include a frame. The invariant must be established by the loop initialization, and must be true after each execution of the body of the loop. Invariants that refer to prestate cannot be directly translated into Eiffel, but they do provide advantages in refinement, in that they allow us to more easily ensure that a loop establishes a postcondition that makes reference to pre- and poststate. This is similar to loop rules based on a two-state invariant used with VDM [Jon90].

Here is the refinement rule for loops.

Rule 4.4 *Introducing an initialized loop. Let b be a boolean expression, I a loop invariant, v a loop variant, and $Loop$ a loop as above. Then*

$$Q \sqsubseteq Loop$$

provided that

$$\begin{aligned} \mathbf{old\ pre}\ Q \wedge Init &\rightarrow I \wedge v \geq 0 \\ I \wedge b &\rightarrow Q \\ I \wedge \neg\, b \wedge Body' &\rightarrow I[\_ := \_'] \\ I \wedge \neg\, b &\rightarrow v \geq 0 \\ I \wedge \neg\, b &\rightarrow v < \mathbf{old}\ v \end{aligned}$$

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

The first "provided that" clause says that when enabled, the initialization must establish the invariant, and must set the variant to a non-negative value (as required by Eiffel). The second clause says that on exit the invariant must establish the required specification under refinement. The third clause is a triple-state formula – it asserts that every execution of the body begun with the invariant true must preserve the invariant (note that the invariant is a double-state formula). The final two clauses assert, respectively, that every execution of the body must decrease the variant, and that the variant must never be less than zero.

Finally, we present the means for introducing a variable that is local to the body of a routine. Local declarations in Eiffel have the following syntax.

$$\mathbf{local}\ v : T;$$

(The semicolon separates the declaration's type from the routine body to which the declaration applies.) A local declaration is immediately followed by the body of a routine. Local variable introduction is just existential quantification, as the following definition indicates.

Definition 7 *Introduce local variable. Let x be a fresh variable name of type T.*

$$\mathbf{local}\ x : T\ ;\ Q \quad \widehat{=} \quad (\exists\, x, \mathbf{old}\ x : T \bullet Q)$$

The existential quantification is over both pre- and poststate because values of $\mathbf{old}\ x$ may be used within Q (Q may be, for example, a sequence of specifications).

The mechanisms that we have presented so far are adequate for the refinement of specifications to imperative programs. They are inadequate in an OO setting. In object-oriented programs, there are two fundamental instructions – procedure and function calls – which are used to produce changes in the state of objects. Techniques are needed for introducing procedure and function calls. We do this by providing a formal meaning to these two programming constructs.

Recall Fig. 4, which we will use to again define the context for giving the meaning to procedure and function calls. We start by dealing with the simple cases: a targeted procedure call where the argument is a primitive expression

(involving no function calls), and similarly for a function call. We then use the function call in defining a generalized assignment law.

First, we define the meaning of the procedure call $e.r(c)$, where e is an object and c a primitive expression. We assume that, in our BON specifications, all procedure calls are explicitly targeted, either with an object e, or with the current object $Current$. The meaning of the call will be supplied by the contract of r, with a slight twist: a substitution must be made on the variables used in the contracts in order to indicate that the call to r uses the attributes of object e.

Definition 8 *Meaning of a targeted procedure call.*

$$\begin{aligned} e.r(c) \quad \widehat{=} \quad & t, e.a : \langle\!| \ P_r[a := e.a, x := c, Current := e], \\ & \qquad Q_r[\mathbf{old}\ a := \mathbf{old}\ e.a, \mathbf{old}\ Current := \mathbf{old}\ e, a := e.a, \\ & \qquad\quad x := c, Current := e] \ |\!\rangle \end{aligned}$$

The frame of the specification is the bunch $t, e.a$ because, as the postcondition Q_r of routine r in Fig. 4 expresses, only attribute a and the global clock can be changed by the call.

Now we turn to function calls. The approach we take, for the purposes of generality, is to give a meaning to a function call appearing in a program; then, we can use this meaning in defining, for example, the assignment of a function result to an attribute. The specification that we provide will assert, under the assumptions of a function precondition, what is true about the *result* of a function call.

We first introduce a small piece of syntax. Let e be an object whose class has the function f which takes an argument c. We let $e.P_f(c)$ represent the *precondition* of f, and $e.Q_f(c)$ represent the *postcondition* of f, with the following substitutions.

Definition 9 *Targeting pre- and postconditions for a function call. Consider Fig. 4. For function f, we define*

$$\begin{aligned} e.P_f(c) \quad \widehat{=} \quad & (\mathbf{pre}\ f)[a := e.a, Current_e := e, x := c] \\ e.Q_f(c) \quad \widehat{=} \quad & (\mathbf{post}\ f)[a := \mathbf{old}\ e.a, Current_e := \mathbf{old}\ e, \\ & \qquad Result_f := e.f(c), x := c] \end{aligned}$$

In other words, $e.P_f(c)$ represents the precondition of f, suitably instantiated for the object e that is invoking f (and similarly for the postcondition). The use of **old** in the postcondition substitution is necessary to deal with cases like the assignment $d := d.f(c)$, where a function result is used to effect a change in the state of the invoking object.

In Definition 9, we have added subscripts to the variables $Current$ and $Result$. This is primarily for convenience and to promote readability, specifically in situations where we are dealing with several nested calls to the same function. In such situations, when we are simplifying specifications and predicates, we may have several instances of $Result$ or $Current$, and it may be helpful to be able to distinguish instances. Where necessary (i.e., when manipulating specifications with several instances of $Result$ or $Current$), we will always attach a unique distinguishing subscript to each instance of these variables to help with the readability of the specifications we write: the distinguishing subscript will be formed by combining the name of the object and the feature name, together with a unique digit; this mimics the *name mangling* carried out by compilers for object-oriented programming languages that replace method calls with function applications. It is not strictly necessary to carry out such subscripting, since $Result$ and $Current$ will be substituted away eventually (as demonstrated in Definition 9, and in the example of Section 6), but the subscripts will help reduce confusion when carrying out the substitutions.

The meaning of the function call $e.f(c)$ that appears in a program (where c is an expression constructed solely from primitive types) is similar to a procedure call. The meaning is again supplied by the contract of f.

Definition 10 *Meaning of a function call appearing in a program.*

$$e.f(c) \quad \widehat{=} \quad t : \langle\!| \ e.P_f(c), e.Q_f(c) \ |\!\rangle$$

This defines the meaning of a call that appears in a program; it is not to be used to substitute for any occurrences of $e.f(c)$ that appear *within* specification statements (such as in Definition 11 below). Notice that the frame of the specification is just t; the function call changes nothing but, potentially, the clock. The semantics of specification statements given in Definition 1 ensures that a function call does not decrease time. With this specification, we can now easily describe the meaning of an assignment involving a simple function call.

Definition 11 *Meaning of an assignment involving a simple function call.*

$$d := e.f(c) \quad \widehat{=} \quad d, t : \langle\!| \; e.P_f(c), e.Q_f(c) \wedge d = e.f(c) \; |\!\rangle$$

The second conjunct in the postcondition is necessary for those functions f that do not possess postconditions. An example of such a function, the feature $item$ of class $ARRAY$, will be used in the example of Section 6. Such features often belong to implemented libraries, where the implementation language is not expressive enough to capture all the requirements of the postcondition.

These rules suffice for all cases when the argument to the function or the procedure is simple, i.e., it does not involve another function call. For nested function calls, the rule is more complex. We start by providing a rule that gives a meaning to nested function calls of the form $e1.f1(e2.f2(c))$. From this, we produce a rule for a nesting of a procedure and a function call.

Definition 12 *Meaning of nested function calls.*

$$\begin{aligned} e1.f1(e2.f2(c)) \quad \widehat{=} \quad & t : \langle\!| \; e2.P_{f2}(c) \wedge \\ & (e2.Q_{f2}(c) \rightarrow e1.P_{f1}(e2.f2(c)))[\mathbf{old}\ t := t], \\ & e1.Q_{f1}(e2.f2(c))[\mathbf{old}\ t := t_1] \wedge e2.Q_{f2}(c)[t := t_1] \; |\!\rangle \end{aligned}$$

This definition requires some explanation. The precondition of the specification is the conjunction of the precondition of $f2$ and an implication. The implication expresses that the postcondition of $f2$ must establish a state in which $f1$ is enabled (with the result of $e2.f2(c)$ as its argument). The substitution on the time variables in the precondition is so that the time for the inner call can be taken into account. The postcondition is the conjunction of the postconditions for $f1$ and $f2$, where again we substitute the result of the inner-most call for the outermost call's argument. In this substitution, we add the time taken for the inner-most call to the time taken for the outer call (via the substitutions involving the time variables t and $\mathbf{old}\ t$).

Now we can use this rule in defining the effect of assigning the result of a nested function call to an attribute.

Definition 13 *Assignment of the value of a nested function call.*

$$\begin{aligned} d := e1.f1(e2.f2(c)) \quad \widehat{=} \quad & d, t : \langle\!| \; \mathbf{pre}\ e1.f1(e2.f2(c)), \\ & \mathbf{post}\ e1.f1(e2.f2(c)) \wedge \\ & d = e1.f1(e2.f2(c)) \; |\!\rangle \end{aligned}$$

The last rule we present is for a nesting of a procedure call with a function call as an argument, of the form $e3.r(e1.f1(c))$. The meaning is developed directly from the previous rules. First, we provide a definition similar to Definition 9, but for procedures instead of functions.

Definition 14 *Targeting pre- and postconditions for a procedure call. Consider Fig. 4. For procedure r, we define*

$$\begin{aligned} e.P_r(c) \quad & \widehat{=} \quad (\mathbf{pre}\ r)[a := e.a, \mathit{Current}_e := e, x := c] \\ e.Q_r(c) \quad & \widehat{=} \quad (\mathbf{post}\ r)[\mathbf{old}\ a := \mathbf{old}\ e.a, \mathbf{old}\ \mathit{Current}_e := \mathbf{old}\ e, \\ & \qquad\qquad a := e.a, \mathit{Current}_e := e] \end{aligned}$$

The main difference between Definition 14 and Definition 9 is that procedure postconditions can make reference to **old** values of expressions. Now we can express the meaning of a nested procedure and function call.

Definition 15 *Meaning of a procedure call with function result argument.*

$$
\begin{aligned}
e3.r(e1.f1(c)) \quad \widehat{=} \quad e3.a, t : \langle\!| \; & e1.P_f(c) \wedge \\
& (e3.Q_f(c) \rightarrow e3.P_r(e1.f(c)))[\mathbf{old}\ t := t], \\
& e3.Q_r(e1.f(c))[t := t_1] \wedge \\
& e1.Q_{f1}(c)[\mathbf{old}\ t := t_1] \; |\!\rangle
\end{aligned}
$$

The substitution on time variables in the postcondition is necessary to ensure that the time for the inner function call is correctly substituted for the start time of the procedure call.

These rules suffice to generate a meaning for any composition of function calls and procedure calls.

We first attempted to write the rules for feature calls in the standard way by introducing local variables to do the bindings from formal to actual arguments [Mor94]. This is more complicated for the purposes of automated reasoning than the rules presented above, since each local variable declaration introduces existential quantifiers; our rules use only contextual substitutions which are straightforward to automate (e.g., using lambda bindings or record overriding). In some cases, rules for introducing procedure calls in refinement are hard to get right due to subtle interactions between the substitution operator that renames the free variables of a program and the rules for parameterized procedures [CS99]. In our case, Eiffel has a rather simple rule governing what operations can be done to a formal argument x: the body of the routine may not apply any direct modification such as an assignment to x of the form $x := \ldots$. This is more restrictive than "call by value". We can thus define the complete semantics of a call by the use of contextual substitution alone, without the need to use auxiliary local variables and existential operators. Contextual substitution can be fully automated in a simple way, whereas the introduction of existential variables will often require user assistance when theorem provers are used, in order to generate appropriate instantiations for bound variables.

4.1 Correctness of definitions and rules

Two kinds of rules are presented in the previous subsection. The first kind are definitions of the meaning of Eiffel programming constructs. For example, Definition 5 defines the meaning of the selection construct by using a Hehner-style predicate [Heh93]. The predicate characterizes all observable behaviours of the construct as executions of the construct in terms of the prestate $\mathbf{old}\ \sigma$ and the poststate σ. These definitions can be checked for correctness by examining the set of behaviours that satisfy the predicate to see if they are consistent with the model of execution provided for Eiffel constructs in [Mey92]. Since the model of execution is presented only semi-rigorously, this process cannot be fully formal. In fact, these definitions and rules may be seen as an attempt to rigorously specify the model of execution in a way that also allows for program refinement. A further requirement of all definitions is that they be implementable as described in equation (1). Definition 3 expresses what our notion of refinement is — it is simply that any behaviour of the concrete program must also be a behaviour of the abstract specification.

The second kind of refinement rules are derivations such as simple substitution Rule 4.3. These derivations must be produced from the definitions using predicate logic. There are many other helpful derived rules that can easily be obtained (e.g., using Rule 4.2).

5 Class Compositional Process

Refinement, in a setting that combines object-orientation and design-by-contract, can be applied compositionally to large software systems. To illustrate the process, we examine, without loss of generality, the following OO system which possesses four classes and three relationships.

Consider Fig. 5, and 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 aggregation through 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, as shown in Fig. 5, that A has a feature f. Due

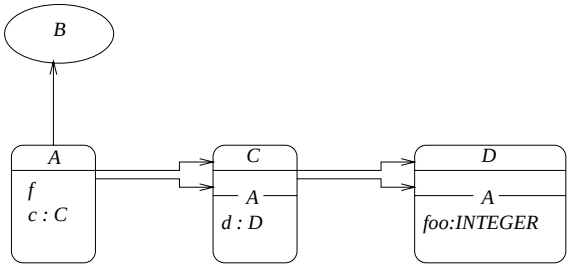

Figure 5: System structure for demonstrating the compositional process

to the aggregation between A and C, A also has a feature $c : C$; the aggregation 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 (acceptable, in the sense that it obeys the BON metamodel).

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

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. So we will need to define a transitive closure of the inheritance and aggregation relationships, which takes into account information hiding, in order to determine the features needed to correctly carry out refinement.

Our composition process will hold for any software architecture that satisfies the following properties.

- All class relationships must be directed, from a source to a target class. Undirected relationships, as permitted in UML, cannot be used.
- Routines must not change the value of attributes outside of their class, i.e., the only way to change the value of attributes is by using commands associated with a class.

(We discuss the need for these properties after presenting the process.) Suppose we want to refine A to an Eiffel program. We must carry out the following steps, in order.

1. Determine $\mathbf{ind}\ A$, the set of all classes that A depends[2] on by client-supplier or inheritance relationships. This set consists of the following classes.
   - **Ancestors:** all classes in the *flat* form of A, i.e., all ancestors of A. In Fig. 5, only B is an ancestor of A.
   - **Direct suppliers:** all classes with which A has a client-supplier relationship drawn in the model. In our example, this subset consists of class C. We call these classes *direct suppliers.*
   - **Locals and parameters:** all classes that A uses to declare local variables or parameters associated with routines. We need classes that are used to declare locals since the local variables may be used in a refinement.
   - **Indirect suppliers:** an indirect supplier of A is a class for which an object of that class appears in any contract of a feature in the *flat form* of A. In Fig. 5, this subset consists of class D. Note that this also includes any indirect suppliers of ancestors of A, because the flat form of a class includes all features acquired from ancestors of the class (as well as all the features' contracts).

   For Fig. 5, $\mathbf{ind}\ A = \{B, C, D\}$ — this is the set of classes that are *independent* of A, whereas A itself depends upon this set containing B, C and D.
2. Determine $\mathbf{spec}\ A$, the set of contracts for all routines in the flat form of 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 $\mathbf{spec}\ A$.

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

3. Show that each element of $\mathbf{spec}\ A$ is implementable. To show implementability, we can prove

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

   Alternatively, refining each contract in $\mathbf{spec}\ A$ to an Eiffel program itself demonstrates implementability.

4. To refine class A to an Eiffel 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 other elements of $\mathbf{spec}\ A$ and any contracts in classes that A depends upon, i.e., contracts of routines from classes in $\mathbf{ind}\ A$. However, we do not need *all* the contracts from classes in $\mathbf{ind}\ A$, because of information hiding. BON and Eiffel allow one or more features of a class to be annotated by a list of classes who can access the features. If a BON model properly obeys the information hiding rules of BON described in [WN95] (i.e., features are only used by client classes who have been given access to them), then we will need only the following restricted set of contracts from the classes in $\mathbf{ind}\ A$, in order to refine elements of $\mathbf{spec}\ A$:

   - the contracts of features belonging to ancestors of A (including contracts used by the indirect suppliers of the ancestors of A).
   - the contracts of features of direct suppliers of A, which are exported to A.
   - the contracts of features of indirect suppliers of A, which are exported to A.

   This calculation assumes that the BON model correctly obeys the information hiding rules of the language.

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 $\mathbf{ind}\ A$. No implementations of the features of $\mathbf{ind}\ A$ are needed. In most cases, the entire system need not be considered when refining a class, since $\mathbf{ind}\ A$ will only involve the contracts of a subset of all classes in the system. Thus, refinement can be done compositionally, class-by-class and feature-by-feature.

To refine an entire BON specification to an Eiffel program, we can start from the root class. The root is refined to code, using only the contracts of the classes in $\mathbf{ind}\ ROOT$ and the contracts of $ROOT$ itself. Then, each class that the root depends upon (i.e., all elements of $\mathbf{ind}\ ROOT$) is 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* [Par72]. In particular, information hiding as it exists in BON requires that only the procedures of a class can effect changes in the state of objects of that class; clients of a class cannot change state directly via assignment to attributes of an object. In other words, the attributes of a class are read-only to clients. If this level of information hiding was not used in BON, then clients could change attributes of a class, and hence $\mathbf{ind}\ A$ would possibly need to contain all classes in the system. Similarly, if undirected relationships, as present in modeling 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 compositional 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 BON, demonstrating the use of the class and feature compositional processes. The refinement will transform a BON specification into an Eiffel program.

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 including the subtle case in which the target

of an assignment invokes a function 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 BON specification for the class FOO.

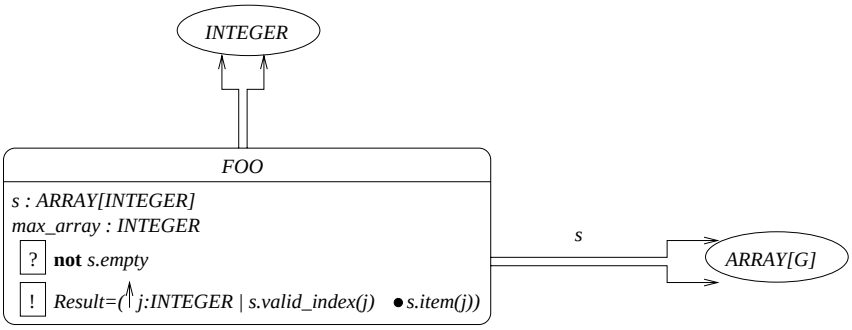

Figure 6: Class FOO for refinement example

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 given by

$$Result_{max\_array} = (\uparrow j : INTEGER \mid s.valid\_index(j) \bullet s.item(j))$$

demonstrates this syntax. For brevity we will use $Result_{ma}$ as an abbreviation for $Result_{max\_array}$. The goal of the refinement process is to implement the mathematical operator $\uparrow$ by the routine max.

Class FOO has two aggregation relationships: with class $ARRAY$ (shown in Fig. 6) and with $INTEGER$, via the return value of function max_array, which calculates the maximum of the array s. The function introduces a local variable, i, as a loop index. A local variable $Result$, automatically declared for the function, 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.

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 $lower \leq upper$
end

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

Figure 7: Excerpt from interfaces of *ARRAY* and *INTEGER* classes

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 function of $ARRAY$, has no postcondition; in this sense, we can view $item$ as an atomic specification unit, one whose meaning is not denoted by any other, perhaps more concrete, representation.

We can now refine the specification of FOO to code. In the process, we will use the contracts, but not the implementations, of the classes on which FOO depends. The process of Section 4 starts by calculating the independent

classes of FOO. These are $INTEGER$ and $ARRAY[INTEGER]$. 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 **ind** FOO that are accessible to FOO (in this example, we have shown only publically accessible features of $INTEGER$ and $ARRAY[G]$ that will be needed in the refinement; in general, all publically accessible features can be used in refinement of a client or descendent class).

First, we present the refinement tree for max_array, shown in Fig. 8. In the refinement, we use the subscript ma to indicate variables associated with the routine max_array.

spec max_array

$=$ $< definition\ of\ max\_array >$

$Result_{ma} : (\!|\ \mathbf{not}\ s.empty,$
$Result_{ma} = (\uparrow j : INTEGER \mid s.valid\_index(j) \bullet s.item(j))\ |\!)$

$\sqsubseteq$ $< (a) :\ introduce\ local\ variable\ (Def.\ 7);\ \ frame\ change\ rule\ (4.1) >$

local $i : INTEGER;$

$i, Result_{ma} : (\!|\ \mathbf{not}\ s.empty,$
$Result_{ma} = (\uparrow j : INTEGER \mid s.valid\_index(j) \bullet s.item(j))\ |\!)$

$\sqsubseteq$ $< (b) :\ introduce\ loop\ (4.4) >$

local $i : INTEGER;$

from $P :\ i, Result_{ma} : (\!|\ \mathbf{not}\ s.empty, i = s.lower \wedge Result_{ma} = s.item(i)\ |\!)$

invariant $I :\ Result_{ma}, i : (\!|\ true, i \leq s.upper \wedge s.valid\_index(i) \wedge$
$Result_{ma} = (\uparrow j : INTEGER \mid s.lower \leq j \leq i \bullet s.item(j))\ |\!)$

variant $V :\ s.upper - i$

until $i = s.upper$

loop $W :$

$i, Result_{ma} : (\!|\ s.valid\_index(i) \wedge i \leq s.upper,$
$i = \mathbf{old}\ i + 1 \wedge Result_{ma} = \mathbf{old}\ Result_{ma} \uparrow s.item(i)\ |\!)$

end

$\sqsubseteq$ $< (c) : refine\ initialization\ P\ (see\ Fig.\ 7);\ \ refine\ loop\ body\ W\ (see\ Fig.\ 8) >$

local $i : INTEGER;$

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

invariant $I :\ Result_{ma}, i : (\!|\ true, i \leq s.upper \wedge s.valid\_index(i) \wedge$
$Result_{ma} = (\uparrow j : INTEGER \mid s.lower \leq j \leq i \bullet s.item(j))\ |\!)$

variant $V :\ s.upper - i$

until $i = s.upper$

loop

$i := i + 1;\ Result := Result.max(s.item(i))$

end

Figure 8: Refinement tree for max_array (the last step produces executable Eiffel code)

In Fig. 8, 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. 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. 8 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 novel refinement step, unique to object-oriented development, occurs in refining the loop body to the sequence of assignments. The second assignment is:

$$Result_{ma} := Result_{ma}.max(s.item(i))$$

which is a nested function call. To introduce this assignment, Definition 13 must be applied. The refinement and proof obligations are explained in detail in the appendix. However, we point out that in all of these proof steps, including that for the function calls in Definition 13, 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 [CO95] would have little difficulty discharging all the obligations for this proof automatically.

The last step of Fig. 6 produces executable Eiffel code. The subscript may be dropped from $Result$ as $Result$ now unambiguously refers to the result of function max_array. The conjunct containing the $maximum$ quantifier in the loop invariant would have to be a comment. However, the latest versions of Eiffel support the notion of an "agent" 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

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

Currently, we have developed a mapping of BON 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 has easily extended to support client-supplier and inheritance relationships, and has allowed non-trivial theorems to be discharged. A particular benefit of translating BON 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].

8 Related Work and Conclusions

Design-by-contract has been shown to be useful for producing reliable, robust, reusable, and maintainable software [Mey97, WN95]. In this paper, we have provided novel rules for object-oriented refinement of specifications (contracts) to code in the BON/Eiffel setting, including rules for feature calls and a new loop refinement rule for double-state invariants. BON allows specifications to be structured using deferred classes, inheritance and the client-supplier relationship. Refinement is compositional and works on the class as the unit of composition; only the contracts of dependent classes are needed (not their implementations). Compositionality and the object-oriented structuring mechanisms make this process applicable to large systems.

Compositional refinement is critically dependent on the fact that all the BON relationships are directed (i.e., the inheritance and various client supplier relationships are directed relationships). The direction indicates class dependencies and thus defines the refinement context. By contrast, UML allows undirected relationships which will undermine compositionality. Thus, to effectively carry out algorithm refinement with UML and its constraint language, OCL, there will have to be restrictions on the modeling constructs that can be used, or on when refinement can be carried out, e.g., after undirected relationships have been replaced by directed or navigable ones.

We are not aware of any other OO approaches that provide refinement rules down to an actual industrial-strength programming language. Furthermore, there are several differences between the BON/Eiffel approach and alternative OO approaches. Object-Z, for example, 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 BON is based on first-order logic, making it easier to support reasoning via available tools, such as PVS.

The work on the Extended Static Checker [DL98] for Java has focused on 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 verification of programs, rather than refinement of specifications to programs. It handles reference types as well as primitive types. We view such an approach 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 likely be more appropriate when verifying library classes or pre-existing applications, rather than when developing new classes or applications.

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 procedure or function 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 BON/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 BON/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. BON/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 BON/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 BON and 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 exceptions. It has no refinement rules defined. Work is underway on integrating JML with the Extended Static Checker [DL98].

This paper did not deal with reference types nor did it show how to automate the refinement procedures. Automation and the extension to reference types are currently under investigation; section 7 reported on the current status 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 Proof of Refinement Steps

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. 8. 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

$$i := s.lower;\ Result_{ma} := s.item(s.lower)$$

The refinement will be by applications of Definition 13, Rule 4.3, and propositional logic. The proof that P is refined by the above sequence is shown in Fig. 9. The proof is carried out by expanding and simplifying the above sequence, and then by showing that the expansion implies P.

$$
\begin{array}{ll}
 & i := s.lower;\ Result_{ma} := s.item(s.lower) \\
= & < \text{Defn. 13} : \textit{assignment to function call} > \\
 & i := s.lower;\ Result_{ma} : (\!| s.valid\_index(s.lower), \\
 & \qquad\qquad Result\_ma = s.item(s.lower) |\!) \\
= & < \textit{postcondition of valid\_index} > \\
 & i := s.lower;\ Result_{ma} : (\!| s.lower \leq s.upper \rightarrow \\
 & \qquad\qquad (s.lower \leq s.lower \leq s.upper), \\
 & \qquad\qquad Result_{ma} = s.item(s.lower) |\!) \\
= & < \textit{propositional logic} > \\
 & i := s.lower;\ Result_{ma} : (\!| true, Result_{ma} = s.item(s.lower) |\!) \\
= & < (4.3)\ \textit{simple substitution} > \\
 & Result_{ma}, i : (\!| true, Result_{ma} = s.item(s.lower) \wedge i = s.lower |\!) \\
\rightarrow & < \text{Defn. 3} : \textit{refinement} > \\
 & P
\end{array}
$$

Figure 9: Proof of refinement for loop initialization

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 procedure call. Fig. 10 shows the tree for the refinement of W.

Most of the proof obligations that arise from this refinement tree are straightforward to discharge. The most interesting, and challenging, obligation comes in the last step of the refinement, where we want to introduce a nested function call in an assignment. We now demonstrate how to discharge this obligation. The approach we take is to start with the assignment statement, apply definitions and rules of logic, and produce a new specification that logically implies the original. The proof is shown in Fig. 11.

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.

[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.

$$
\begin{array}{ll}
 & W \\
\sqsubseteq & < \textit{textual substitution } [\text{GS93}] > \\
 & i, Result_{ma} : \langle\!| \ (s.valid\_index(i) \wedge i - 1 \neq s.upper)[i := i + 1], \\
 & \qquad (i = \textbf{old}\ i \wedge \\
 & \qquad Result_{ma} = \textbf{old}\ Result_{ma} \uparrow s.item(i))[\textbf{old}\ i := \textbf{old}\ i + 1] \ |\!\rangle \\
\sqsubseteq & < \textit{frame change } (4.1),\ \textit{simple substitution } (4.3) > \\
 & i := i + 1; \\
 & Result_{ma} : \langle\!| \ s.valid\_index(i - 1) \wedge i - 1 \neq s.upper, \\
 & \qquad i = \textbf{old}\ i \wedge Result_{ma} = \textbf{old}\ Result_{ma} \uparrow s.item(i) \ |\!\rangle \\
\sqsubseteq & < s.valid\_index(i - 1) \wedge i - 1 \neq s.upper \rightarrow s.valid\_index(i); \\
 & \ \textit{precondition weakening } (\textit{Morgan Rule } 1.2\ [\text{Mor94}]) > \\
 & Result_{ma} : \langle\!| \ s.valid\_index(i), \\
 & \qquad Result_{ma} = \textbf{old}\ Result_{ma} \uparrow s.item(i) \ |\!\rangle \\
\sqsubseteq & < \textit{ proof described in Fig. } 11 > \\
 & Result_{ma} := Result_{ma}.max(s.item(i))
\end{array}
$$

Figure 10: Refinement tree for the loop body

$$
\begin{array}{ll}
 & Result_{ma} := Result_{ma}.max(s.item(i)) \\
= & < \textit{meaning of a nested function call } \text{Defn. } 13 > \\
 & Result_{ma} : \langle\!| \ Result_{ma}.P_{max}(s.item(i)) \wedge s.P_{item}(i), \\
 & \qquad Result_{ma}.Q_{max}(s.item(i)) \wedge s.Q_{item}(i) \wedge \\
 & \qquad Result_{ma} = Result_{ma}.max(s.item(i)) \ |\!\rangle \\
 & \\
= & < \textit{by } \text{Defn. } 1 : Result_{ma}.P_{max}(s.item(i)) = true, \text{and } s.Q_{item}(i) = true > \\
 & Result_{ma} : \langle\!| \ true \wedge valid\_index(x)[Current_{item} := s, x := i], \\
 & \qquad (Result_{max} = Current_{max} \uparrow x) \\
 & \qquad [Current_{max} := \textbf{old}\ Result_{ma}, x := s.item(i), \\
 & \qquad\ Result_{max} := Result_{ma}.max(s.item(i))] \\
 & \qquad \wedge true \wedge Result_{ma} = Result_{ma}.max(s.item(i)) \ |\!\rangle \\
 & \\
= & < \textit{do substitutions; } \text{GS } (3.39)\ \textit{identity of conjunction } true \wedge p = p > \\
 & Result_{ma} : \langle\!| \ s.valid\_index(i), \\
 & \qquad Result_{ma}.max(s.item(i)) = \textbf{old}\ Result_{ma} \uparrow s.item(i) \wedge \\
 & \qquad Result_{ma} = Result_{ma}.max(s.item(i)) \ |\!\rangle \\
 & \\
\rightarrow & < \textit{transitivity of equality} > \\
 & Result_{ma} : \langle\!| \ s.valid\_index(i), Result_{ma} = \textbf{old}\ Result_{ma} \uparrow s.item(i) \ |\!\rangle
\end{array}
$$

Figure 11: Proof of final refinement step

[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.

[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.

[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.

[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.

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

[OP00] J.S. Ostroff and R.F. Paige.: The Timed Predicative Calculus as a Framework for Comparative Semantics. Technical Report CS-2000-01, Department of Computer Science, York University, April 2000.

[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.

[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.

[dRE98] 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.

[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.