

---

# Integrating a program design calculus and a subset of UML

RICHARD F. PAIGE

*Department of Computer Science, York University, Canada*

*Email: paige@cs.yorku.ca*

---

**The predicative programming design calculus is integrated with an object-oriented method that uses a subset of the Unified Modelling Language. The integration is carried out so as to make the calculus and refinement techniques more applicable to the development of large-scale object-oriented software. The two methods are integrated using a *meta-method* for formal method integration based on heterogeneous notations. We demonstrate how the methods being integrated complement each other, and outline the use of the integrated method in a case study.**

*Keywords: method integration, UML, refinement, formal methods, heterogeneous notation*

*Received ; revised ; accepted*

---

## 1. INTRODUCTION

It has been suggested that a single software development method is insufficient for all situations 1. This is due, in part, to the complexity of the problems being solved, the diversity of expertise among developers, and the limitations of a single set of notations and processes. Method integration 2 is a technique that can be used to abet multiple-method use. Method integration has been studied in the context of combining specific methods 3, 4, and in settings where systematic approaches for integration have been constructed 2, 5, 6, 7. Our intent with this paper is to follow the latter path and continue the work of 6, where a meta-method for formal method integration was first presented.

This paper has two main goals. The first is to present an integration involving the program design calculus of predicative programming 8 and a representative object-oriented method that uses a core subset of the Unified Modelling Language (UML) 9. The integration is carried out using the meta-method of 10, and is presented so as to show how to make a program design calculus—and more generally, any formal method—more appropriate for large-scale software development, by combining it with an object-oriented method. The integration will allow the design calculus and its refinement techniques to be *restrictably* applied, where developers think that it will help, with full rigour. It will also present a technique for changing the extent of formality that is explicitly used in a development.

The second goal of this paper is to show the use of the integrated method in a case study, and in the process explain how the method integration approach can be used to help scale up formal methods that apply refinement techniques to large-scale development.

### 1.1. Heterogeneous notations and specifications

Notations play an important role in software development methods, in writing descriptions of requirements, systems, and programs, and in documenting the development process. Notations will play a key part in how we integrate methods: we combine notations as the first step in combining methods. A *heterogeneous notation* is composed from several different notations and is used to write heterogeneous specifications.

DEFINITION 1.1. A heterogeneous specification is composed from parts written in two or more different notations.

Heterogeneous notations are useful for a number of reasons: for extending expressive capabilities of single notations; for producing simpler specification languages 11; for writing simpler specifications than might be produced using a single language 11; for ease of expression; and because they have been proven successful in practice 4, 11, 12. Heterogeneous notations are ideally applied in a setting where the complementarity of the separate notations is evident, and when the notations being combined can be used together without significant alteration.

A formal semantics for a heterogeneous specification can be given by formally defining the meaning of the composition of partial specifications in some notation. If we combine exactly two notations, where one is formal and the other informal, we can formally define specification compositions by formalizing the informal notation in the formal one. This approach lets specifiers treat the compositions as if they were written in the formal notation. More generally, we may want to build a heterogeneous notation from more than two formal

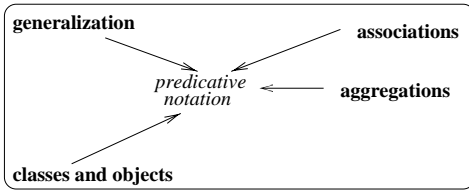


FIGURE 1. The heterogeneous basis with UML concepts

or informal notations. In this case, we can construct a *heterogeneous basis* 6, a set of notations and translations between notations, which gives a semantics to heterogeneous specifications via translation to a homogeneous specification. Constructing a heterogeneous basis is usually hard, and requires careful understanding of the semantics and interpretations of all the notations, as well as a means to resolve syntactic and semantic incompatibilities across different notations.

In this paper, we are interested in combining the use of predicative programming and a method that uses a subset of UML. The manner in which we integrate methods will result in a new method that produces and uses heterogeneous specifications composed from predicative specifications and UML specifications<sup>2</sup>. To be able to formally reason about heterogeneous specifications, a mathematical semantics must be provided. To this end, we will select and formalize a subset of UML in predicative notation. Formalizing parts of UML in predicative notations means that, for free, predicative specifications and UML specifications can be composed and thereafter reasoned about using predicative rules, laws, and techniques. For more on heterogeneous bases, their construction, and use, see 6. By formalizing UML specifications in predicative notation, we effectively construct a very simple heterogeneous basis consisting of selected UML constructs, and predicative notation. This basis is depicted in Figure 1.

In Figure 1, arrows from UML constructs to the formal notation represent formalizations. We have omitted the dynamic constructs of UML in the heterogeneous basis, such as state transition and collaboration diagrams. This omission is intentional. The integration replaces the use of these dynamic notations with the predicative notation. In Section 4, we discuss notation replacement in more detail. It is not our purpose to formalize all of the UML, but to formalize a subset so that it can be usefully applied with predicative programming.

If we were to extend the heterogeneous basis by defining translations from a new formal notation—e.g.,  $Z$ —to predicative notation, then we would effectively have the capability to use  $Z$  with UML specifications. It is this basis extension process that can be used to generalize our results. We discuss this more in later sections

<sup>2</sup>For consistency and to avoid ambiguity, we use the term specification throughout this paper to refer to concrete documents written using a notation.

of the paper.

## 1.2. Integrating methods with heterogeneous notations

Before outlining the approach we take to integrating methods, we define some terminology.

We use the term *process* to refer to the micro-scale level of the software construction phase 13. In other words, a process is concerned with technical issues, and not the higher-level software engineering tasks like project management, people issues, end-user involvement, et cetera.

A *method* for software development has a notation component and a process component. The process component guides the developer by setting milestones in the form of document products, and suggests ways to construct these products. The notation component is used in writing (and reasoning about) the documents.

A *meta-model* captures the document products and structures produced by a method, as well as the process phases and their temporal ordering 7. A meta-model is used to admit well-formed specifications, and to reject ill-formed ones.

*Formal method integration* is method integration involving at least one—and possibly several—formal methods 10.

It is suggested in 6 that constructing heterogeneous notations can be a first step in integrating formal methods with other methods. Informally, integration can occur by formalizing the meaning of compositions of partial specifications written in the notations of interest; this also requires ensuring that the heterogeneous notation can be parsed. Once this is done, integration continues by informally defining relationships between the processes of the methods that are being combined, outlining how the process of one method is interleaved with the process of a second method. More details are given in Section 3.

We suggest that the process of integrating formal methods with other methods can be partially systematized by constructing heterogeneous notations. Their construction allows the notation-related complications associated with formal method integration to be dealt with first. The heterogeneous notation can therefore provide a basis for defining relationships between the processes of the methods being combined. We discuss this more in Section 3. We do not claim that a heterogeneous approach to method integration is always sufficient. We identify situations where other approaches may be necessary in Section 3.

## 1.3. Overview

In this paper, we summarize a meta-method for formal method integration based on heterogeneous notations that was presented in 6. Then we apply it in combining a UML-based method with predicative programming.

We discuss why the combination is useful and justify the method choices. The general organization of the paper is as follows. Section 2 contains an introduction to the predicative programming method and its notations. We concentrate on explaining the notations relevant to object-oriented specification and design, particularly the *bunch* type theory. In Section 3, we summarize the meta-method of 6, and contrast it with more traditional approaches. In Section 4, we apply the meta-method and integrate the UML-based method and predicative programming. In the process, we construct the heterogeneous basis of Figure 1. In Section 5, we summarize the details of a case study using the integrated method, focusing on the roles that the individual methods will play in the combination, and on justifying the use of formality throughout development. Finally, in Section 6, we discuss the integration, and suggest how it can be extended to other formal methods.

## 2. PREDICATIVE PROGRAMMING

Predicative programming 8 is a program design calculus in which programs are specifications. In this approach, programs and specifications are predicates on pre- and poststate (as in Z, final values of variables are annotated with a prime; initial values of variables are undecorated). The weakest predicate specification is  $\top$  (“true”), and the strongest specification is  $\perp$  (“false”). Refinement is just boolean implication.

**DEFINITION 2.1.** A specification  $P$  on prestate  $\sigma$  and poststate  $\sigma'$  is refined by a specification  $Q$  if  $\forall \sigma, \sigma' \cdot (P \Leftarrow Q)$ .

Refinement in predicative programming simpler than in Z or VDM; refinement rules for predicative programming are laws of boolean logic. Thus, the refinement relation enjoys various properties that allow specifications to be refined by parts, steps, and cases. As well, specifications can be combined using the familiar operators of boolean theory, along with all the usual program combinators, including combinators for concurrency and communication through channels.

Predicative programming can be used to specify objects and classes. To do so, we need to introduce the predicative notation for types, namely bunches.

### 2.1. Bunches and types

Bunches were introduced in 14, and are used in 8 as a type system. A bunch is a collection of values, and can be written as in this example:  $2, 3, 5$ . A bunch consisting of a single element is identical to the element. Some bunches are worth naming, such as *null* (the empty bunch), *nat* (the bunch of natural numbers), *int* (the bunch of integers), *char* (the bunch of characters) and so on. More interesting bunches can be written with the aid of the solution quantifier  $\S$ , pronounced “those”, as in the example  $\S i : int \cdot i^2 = 4$ . We use the asymmetric

notation  $m, ..n$  for  $\S i : int \cdot m \leq i < n$ . The bunch cardinality operator is  $\#$ ; thus  $\# m, ..n = n - m$ . Bunches can also be used as a type system, as in

$$\mathbf{var} \ x : nat$$

perhaps with restrictions for easy implementation. Any bunch (including an element, or *null*) can be used as a type. Bunches can also be used in arithmetic expressions, where the arithmetic operators distribute over bunch union (comma):

$$nat = 0, nat + 1$$

We have previously used a colon in expressions involving bunches; more generally,  $A : B$  is a boolean expression saying that  $A$  is a subbunch of  $B$ . For example,

$$\begin{aligned} 2 & : nat \\ nat & : int \end{aligned}$$

When we use a bunch as the type of a variable, as in the declaration of  $x$  above, we can thereafter use the variable to represent zero or more elements of the specified type. So, for our example declaration for  $x$  of bunch *nat*,  $x$  can stand for zero, one, two, or any other quantity of natural numbers. So  $x$  could be the bunch 1 or the bunch 5, 6, 7 for example, and the subbunch relation  $x : nat$  is still true.

We write functions in a standard way, as in the example  $\lambda n : nat \cdot n + 1$ . The domain of a function is obtained using the  $\Delta$  operator. If the function body does not use its variables, we may write just the domain and body with an arrow between. For example,  $2 \rightarrow 3$  is a function that maps 2 to 3, which we could have written  $\lambda n : 2 \cdot 3$ , with an unused variable.

When the domain of a function is an initial segment of the natural numbers, we sometimes use a list notation, as in  $[3; 5; 2; 5]$ . The empty list is  $[nil]$ . We also use the asymmetric notation  $[m; ..n]$  for a list of integers starting with  $m$  and ending before  $n$ . List length is  $\#$ , and list catenation is  $+$  (raised plus).

Function formation distributes over bunch union, and so a function whose body is a union is equal to a union of functions.

$$(\lambda v : D \cdot A, B) = (\lambda v : D \cdot A), (\lambda v : D \cdot B)$$

A union of functions applied to an argument gives the union of the results

$$(f, g) \ x = fx, gx$$

A function  $f$  is included in a function  $g$  according to the *function inclusion law*.

$$(f : g) = ((\Delta g : \Delta f) \wedge (\forall x : \Delta g \cdot fx : gx))$$

Thus we can prove

$$(f : A \rightarrow B) = ((A : \Delta f) \wedge (\forall a : A \cdot fa : B))$$

By letting  $list = \lambda T : \Delta list \cdot 0, ..\#(list\ T) \rightarrow T$  then  $list\ T$  consists of all lists whose items are of type  $T$ .

The selective union  $f \mid g$  of functions  $f$  and  $g$  is a function that behaves like  $f$  when applied to an argument in the domain of  $f$ , and otherwise behaves like  $g$ .

$$\begin{aligned}\Delta(f \mid g) &= \Delta f, \Delta g \\ (f \mid g)x &= \text{if } x : \Delta f \text{ then } f\ x \text{ else } g\ x\end{aligned}$$

One of the uses of a selective union is to write a (selective) list update. For example, if  $L = [2; 5; 3; 4]$  then

$$2 \rightarrow 6 \mid L = [2; 5; 6; 4]$$

changes element 2 of list  $L$  to 6. Another use is to create a record structure, as in “ $name$ ”  $\rightarrow$  “ $Smith$ ”  $\mid$  “ $age$ ”  $\rightarrow$  33 which is included in “ $name$ ”  $\rightarrow list\ char \mid$  “ $age$ ”  $\rightarrow nat$ .

## 2.2. Functional refinement

Functional refinement in predicative programming is similar to imperative refinement (Definition 2.1). A function  $P$  is refined by a function  $S$  if and only if all results that are satisfactory according to  $S$  are also satisfactory according to  $P$ . Formally, this is just bunch inclusion,  $S : P$ . Typically, when writing refinements, we prefer to write the problem  $P$  on the left, and the solution  $S$  on the right. We therefore write  $P \vdash S$  (read as “ $P$  is refined by  $S$ ”), which means  $S : P$ . We will use functional refinement in the case study of Section 5.

## 3. A META-METHOD BASED ON HETEROGENEOUS NOTATIONS

A meta-method for formal method integration was presented in 6, 10. The meta-method describes an informal approach to integration, via combining notations and relating processes. We recap the steps of the method here for the sake of completeness. Then, we discuss differences between this approach, and alternatives.

1. **Select a base method.** A base method provides a process that is to be supported and complemented by other (invasive) methods. Selecting a base method is a step to assist integrators in determining roles that the separate methods can play in the integrated approach.
2. **Choose invasive methods.** Invasive methods augment, are embedded in, or have their processes interleaved with that of the base method. The invasive methods are chosen to complement the base methods, through notation, through process, or through user preferences. The invasive methods may overlap with the base method; that is, the invasive methods may duplicate notations or process phases provided by the base method. In choosing

the invasive methods, it must also be decided how to reconcile these overlaps, for example, by restricting the use of a method.

3. **Construct or extend a heterogeneous basis.** This is accomplished by defining translations, or by adding notations from the base and invasive methods to an existing one. A single formal notation from the basis that is to be used to provide a formal semantics to heterogeneous specifications can be chosen and fixed at this point. The basis notation provides the underlying semantics in which reasoning about heterogeneous specifications is carried out; the basis notation should be chosen to make this reasoning as simple as possible to do. Specifiers and developers otherwise use the separate notations from the individual methods in using the integrated method, and apply the basis notation when reasoning about heterogeneous specifications. It may be determined that the notations being combined overlap syntactically or semantically. The notation integrator must decide how to deal with such overlap, e.g., by restricting use of one notation.
4. **Define how the individual methods cooperate.** It is informally described (using an entity-relationship style of notation) how the processes of the methods are to work together in the new method. Two forms of cooperation are particularly common.
  - **Generalization.** The process of the base method is generalized to use heterogeneous notations constructed from those of the base and invasive methods. Effectively, notations are added to a method, and its process is generalized to use the new notations.
  - **Interleaving of processes.** Relationships between the process of the base method and the processes of the invasive methods are defined. Relationships are specified using an informal entity relationship-like notation (an example is given in Figure 8). Examples of relationships include the following.
    - *Linking* of processes, by defining a translation between notations of different methods. An example is in 15, in which a translation from Z to the Business Object Notation links the Z ‘established strategy’ with the BON method.
    - *Replacement* of entire process steps in a base method by process steps of an invasive method. The invariant in such a replacement is that the steps being added must do at least the tasks of the steps they are replacing.
    - *Supplementation* of process. Specific phases of one method are identified and

are supplemented by phases from a second method. The integration of Z and predicative programming in 16 provides an example; the Z established strategy is supplemented by refinement rules for reasoning about time and space.

- *Parallel use* of processes, by describing relationships that interleave the use of two or more separate processes.

5. **Guidance to the user.** Hints, examples, and suggestions on how the integrated method can be used is provided.

### 3.1. Discussion and comparison

The meta-method described in the previous section is informal. Informal specifications of process relationships are described, and the meaning of specifications created during use of the integrated method is made formal. The meta-method is intended primarily as a *thinking* tool, to support method integrators in selecting methods and in determining the roles each will play in the final product.

This approach is aimed directly at formal method integration, which always involves at least one formal method. To use formal methods in cooperation with other methods requires a formal semantics for all specifications that are produced, otherwise proof rules, refinement, and other formal analyses cannot be carried out with full rigour. For example, to refine specifications by *parts* over a specification combinator requires a proof that refinement is monotonic over the combinator. Without a formal semantics for a heterogeneous specification, such a proof cannot be carried out, and so developers have no formal justification for carrying out such steps.

It seems likely that this approach will not suffice for integrating all methods in all situations, especially when integrating tools like CASE diagrammers and theorem provers, which support methods. In such cases, where we need more rigour in defining integrations, we may want to move to approaches where meta-models are applied, e.g., 5, 17.

In situations where meta-models may be useful, the heterogeneous approach may still prove to be helpful as a *front end*. By first attempting to integrate via heterogeneous notations, we can obtain indications of the incompatibilities between the methods, and may discover ways to resolve these clashes. The meta-method could therefore be used to help select methods, to clarify complementarity, and to determine the roles that the individual methods will play in the integration. Then, if the integration does not provide sufficiently precise descriptions of process, we can move to other techniques, and can therein apply our knowledge from the integration via heterogeneous notations. Even when combining methods using meta-models, a formal semantics for heterogeneous specifications may be desirable, especially

when reasoning and proof needs to be carried out.

The main difference between meta-modeling approaches and our approach is that we use heterogeneous notations as the basis for integration. The meta-method requires that notations be given a formal semantics, so that we can apply formal reasoning to specifications. In a meta-modeling approach, explicit compositions of specifications written in different notations typically do not arise: relationships between specifications are constructed, so that information can be shared. Use of heterogeneous notations may be hard to justify in settings where the notations to be combined have to be changed a great deal. And in the case where the notations being combined present overlapping views of a system, consistency of such views must be established (though this may be hard).

Finally, the meta-method is restricted to integrations involving at least one formal method. Meta-modeling approaches are intended to integrate all forms of methods, though they have been used most frequently to integrate semiformal methods 2.

## 4. AN INTEGRATION OF A UML-BASED METHOD AND PREDICATIVE PROGRAMMING

In this section, we describe an integration of an object-oriented method with predicative programming. The object-oriented method, which uses a core subset of UML notations, is an attempt at a representative microprocess that has similarities to those in many popular methods; for the sake of clarity, we call the method *OOM*. We use this process over ones like the Rational Unified Process 13 or OPEN 18 because it is simpler to present concisely, because these processes focus on more than just the microscale software construction activities, and because our example microprocess contains phases that appear in both processes and other, related methods. Thus, it should be possible to see how to migrate this work to alternative processes, or even alternative OO methods.

We describe the relationships between the *OOM* process and the predicative programming process, as well as the roles that the individual notations and methods will take in the integration.

### 4.1. Justifications

Integrating predicative programming with *OOM* is useful for many reasons.

- The integration demonstrates how an object-oriented method can be extended to (restrictably) work with a formal method.
- It shows how a program design calculus can be made more accessible and attractive to developers through its combination with an object-oriented method, which allows restrictable application of the formal method's refinement techniques.

- It provides an example of how to scale refinement-based methods up to deal with large-scale problems, by integrating them with object-oriented techniques.
- It provides evidence to support our claims about the systematic integration capabilities of the meta-method of Section 3.

A subset of UML was chosen as the notation for *OOM* because it is quickly becoming a standard object-oriented modelling notation, and because it is process independent; thus, it should be possible to see how this work could be adapted to fit other object-oriented methods or processes. Predicative programming was chosen because it is among the simplest program design calculus, and because it is very general, immediately applicable to specifying and reasoning about object-oriented, real-time, and concurrent systems without extension.

We could have chosen to integrate *OOM* with *Z* or *VDM*; in doing so, we could make use of the existing work on formalizing OO concepts in *Z* or *VDM*. But it is harder to carry out refinement in *Z*, in part because *Z* is not a wide-spectrum language. Further, *VDM* is based on a three-valued logic; our preference is to use boolean logic so that we can use refinement rules that are just theorems of boolean logic. This also allows us to make use of standard software tools, such as *PVS*, to abet our specification and reasoning. Finally, refinement in predicative programming is simpler than in *VDM* and *Z*. Since our focus is to make refinement more appropriate for realistic problems, it is to our advantage to consider a simpler refinement technique.

We shall not use all of UML in the integrated method; we will only use a subset of the UML that is not superseded by predicative notation. When methods are combined, there is inevitably overlap in the documents or tasks that the separate methods support (see step 2 and 3 of the meta-method). Part of the task of method integrators is to deal with these overlaps, either by restricting the use of features from separate methods, or by reconciling the overlap in some other way. We restrict use of UML here, because the replacement notation is more concise and amenable to refinement and formal reasoning than the dynamic UML constructs.

#### 4.2. Applying the meta-method

In terms of applying the meta-method, we make the following decisions.

- The base method will be *OOM*, since we intend to use its specification techniques and the UML notation to guide the production of a system specification. Predicative programming will be the invasive method.
- The predicative notation will serve as a basis notation. This is done so as to minimize the need for translation: if we were to use a third notation (e.g., an object-oriented theory from 19) as the seman-

tic basis, then both predicative notation and UML would have to be translated to this notation. Further, predicative notation is rich enough to specify the subset of UML that we need (as well as being able to handle the obvious extensions), and predicative programming's refinement rules are powerful and simple enough to prove useful theorems about specifications. To this end, we shall formalize the UML constructs that we need in predicative notation.

- Predicative programming will *supplement* and *replace* parts of the *OOM* process. Predicative notation will be selectively used to specify classes and features. State transition diagrams, collaboration diagrams, and all other dynamic diagrams from UML will not be used; they will be replaced with predicative notation. Further, the *OOM* process will be supplemented by predicative refinement rules.
- When applying the integrated method, predicative programming will be used to specify and design those classes in the system with complex functionality or complex data, as well as those classes which are central to the correct operation of the system. UML will be used in specification, and the *OOM* process will be applied in all other situations. The predicative refinement rules will be used to transform predicative specifications to code.

It is important to note that the *extent* to which predicative programming is applied is not fixed; the *role* in which predicative programming is used has been fixed, but the formal method can be used within the integrated technique to the extent which the developer thinks is useful. So, for example, a developer can choose to not use predicative programming (and just rely on *OOM*), or use predicative notation for specifying all class features, or also use predicative refinement rules. The developer can also use the heterogeneous basis to change the extent to which predicative programming is used during development. This differs from other integrations involving formal methods, e.g., 20, wherein the role of the formal method was defined and the extent to which the formal method was used was also fixed.

Predicative programming and *OOM* are complementary methods; we direct the reader to 21 for discussion on complementarity. In 21, it is suggested that complementarity of methods can be explained in terms of notation *and* process. In terms of notation, *OOM* offers constructs better suited to description of architecture and large-scale design than does predicative programming; several UML notations are visual, while predicative notations are strictly text-based. Thus, combining the use of *OOM* and predicative programming is justified.

The meta-method of Section 3 requires formalizing UML notations in order to integrate predicative programming and the *OOM*. We now present new formal-

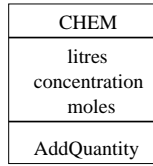


FIGURE 2. An example class in UML notation

izations of selected UML notations, in predicative notation.

### 4.3. Formalizations of UML constructs

UML notations include *class diagrams*, for describing classes and relationships such as *generalization*, *aggregation*, and *association*. UML also includes *object diagrams*, for expressing instantiation and links between objects. It also provides *dynamic notations*, such as *collaboration diagrams* and *state transition diagrams*, denoting the reactive behaviour of objects. In this paper and the case study, we will not use the last two categories. Behaviour of classes and objects will be specified using predicative notation, rather than transition diagrams. However, at the end of this section, we outline how we might formalize the dynamic notations in predicative notation, and suggest why these notations may be redundant in the integrated method.

We now present formalizations of the static UML notations we will use in later sections. The notations will be formalized in predicative programming, based on an object-oriented style for predicative specification. For alternative formalizations of object-oriented notations, the reader is referred to the work of Hall 22 and Hammond 20 (who considers Shlaer-Mellor), or an algebraic approach to the static diagrams of OMT in 23. 19 presents several rich theories of object-oriented concepts that are more expressive and powerful than our own. The work of the Precise UML Group has focused on formalizing and reasoning about UML 24, 25. Our intent in formalizing UML concepts in predicative notation is to be able to integrate predicative programming and *OOM*; it is not to present a novel new theory of objects. Formalizing object-oriented concepts in predicative notation means that we can use predicative programming for reasoning about object-orientation, and the integration process is thereafter simplified.

#### 4.3.1. Classes and objects

A key part of object-oriented development is the identification of classes. A class is an abstract data type with a possibly partial implementation 26. An example of a class, in UML, is given in Figure 2. Class *CHEM* has attributes *litres*, *concentration*, property *moles*, as well as a procedure *AddQuantity* which increases the available volume of a chemical.

A class is formalized in predicative programming as a bunch of attributes with zero or more associated pro-

cedures; the attributes of the bunch may be state components, properties, or functions. Procedures will be formalized as functions on objects.

A class formalization in bunch theory has three components: an *interface*, in which the names and signatures of attributes are specified; a class *definition*, which defines the properties and functions; and, zero or more procedures. The class definition or procedure specifications can be omitted or postponed, for example, if formal meanings of functions, properties, or procedures have not yet been determined.

The formalization process for a class specified in UML is as follows.

1. Define signatures for the attributes of the class.
2. Specify a class interface, which is a bunch with name, say,  $CInt$ . This bunch has components with the same names as the attributes of the class, and with the signatures constructed in step 1.
3. Give a *class definition*  $C$  that defines the functions and properties of  $CInt$ .  $C$  will typically have the form

$$C = \S c : CInt \cdot P$$

where  $P$  is a predicate that gives definitions of the functions and properties.

An interface for class *CHEM* in Figure 2 is

$$\begin{aligned} CHEMInt = & \text{“litres”} \rightarrow real \mid \text{“moles”} \rightarrow real \\ & \mid \text{“concentration”} \rightarrow real \end{aligned}$$

(In fact, a class can have several interfaces— as is permissible in UML and other modeling languages. However, only one class definition is permitted, in which functions and properties are defined and all class interfaces are merged, so as to maintain consistency.)

To express the meaning of functions or properties, we provide a class definition. This requires us to restrict the interface specification of *CHEMInt* to include only those elements that satisfy the required definition of the *moles* property. The *CHEM* class definition is therefore

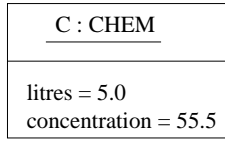
$$\begin{aligned} CHEM = \S c : CHEMInt \cdot c \text{“moles”} = \\ c \text{“litres”} \times c \text{“concentration”} \end{aligned}$$

The bunch *CHEM* consists of all instances of bunch *CHEMInt* that satisfy the definition of *moles* given above. *moles* is a property in UML; it can be implemented as an attribute or as a function.

With this formalization of classes, formalizing instantiation and objects is thereafter straightforward. An object is an instance of a class; an object is shown in Figure 3. It shows the instantiation of an object  $C$  (with specified attribute values) from class *CHEM*.

We can formalize instantiation of objects as variable declarations. To formalize Figure 3, we could write

$$\begin{aligned} \text{var } C : CHEM \cdot \\ \text{“litres”} \rightarrow 5.0 \mid \text{“concentration”} \rightarrow 55.5 \mid C \end{aligned}$$



**FIGURE 3.** Instantiation of object  $C$  (with attribute values) from class  $CHEM$

Nothing about this formalization restricts us to using objects in a sequential setting; predicative programming includes techniques for specifying and reasoning about concurrency and communication. Further, we are not restricted to statically declared objects. Because we formalize classes as bunches, creating new objects dynamically requires no further notation.  $C$  can represent zero or more instances of class  $CHEM$ . We can also restrict variables to represent unique objects; for example, we could restrict object  $C$  to be able to represent only one instance of class  $CHEM$ . Since  $C$  is a bunch, to say that  $C$  can represent exactly and only one object of type  $CHEM$ , we write

$$\# C = 1$$

A class possesses two kinds of methods: functions and procedures. We do not allow ‘hybrid’ methods, like those expressible in C++, which are both functions and procedures. Procedures are formalized as functions on objects. For example, consider a procedure  $AddQuantity$ , which adds a quantity of chemical (in litres) to an object of class  $CHEM$ . This would be specified as follows.

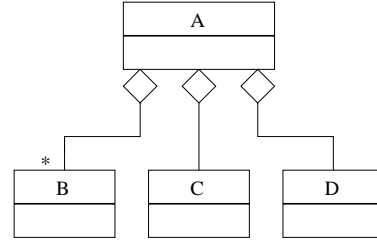
$$AddQuantity = \lambda c : CHEM \cdot \lambda q : nat \cdot \\ \text{“litres”} \rightarrow (c\text{“litres”} + q) \mid c$$

So  $AddQuantity$  is a function that takes an object as one argument, and returns a different object.  $AddQuantity$  would be used as follows. To invoke  $AddQuantity$  on object  $c$ , we write  $c.AddQuantity$  which is sugar for  $c := AddQuantity(c)$ .  $c.AddQuantity$  would likely be implemented as a method that operates on the local state of the invoking object  $c$ . This formalization mimics approximately how method calls in object-oriented programming languages are compiled.

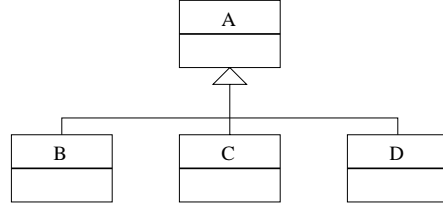
#### 4.3.2. Static relationships

We now examine how the preceding might be used in expressing further UML concepts. We start by considering notations to describe relationships between classes and objects. Aggregations can describe class assembly relations, i.e., how classes are put together from other classes. Consider the diagram in Figure 4.

Figure 4 describes static relationships between a class  $A$  and classes  $B$ ,  $C$ , and  $D$ . An instance of  $A$  is assembled from instances of  $B$ ,  $C$ , and  $D$ . In the predicative formalism, we can express  $A$  as a new class having components from classes  $B$ ,  $C$ , and  $D$ ; this allows us to formally distinguish the meaning of aggregation from that



**FIGURE 4.** Aggregations



**FIGURE 5.** Generalization

of inheritance (see below). We choose fresh attribute names,  $b$ ,  $c$ , and  $d$ , and first define  $BaseA$  as a class interface with (at least) these attributes.

$$BaseA = \text{“}b\text{”} \rightarrow B \mid \text{“}c\text{”} \rightarrow C \mid \text{“}d\text{”} \rightarrow D$$

UML notation allows annotation of aggregations with cardinality constraints. The multiplicity constraint  $*$  on the line between  $A$  and  $B$  indicates that in the formation of  $A$  there can be zero or more instances of class  $B$ . We express such cardinality constraints in producing a bunch definition for class  $A$ . The example in Figure 4 is formalized as

$$A = \S a : BaseA \cdot (\# a\text{“}c\text{”} = 1 \wedge \# a\text{“}d\text{”} = 1)$$

In class diagrams, generalizations are used to describe classification, subtyping, and reuse; an example is shown in Figure 5.

In Figure 5, an instance of  $B$  is also an instance of  $A$  (and similarly for  $C$  and  $D$ ). To formalize this, we first formalize class  $A$ .  $A$  has a class interface or a class definition (because in a class diagram, generalizations can be applied to interfaces or classes). Then, the interfaces of classes  $B$ ,  $C$ , and  $D$  are constructed by merging the interface or definition of  $A$  with any new attributes or functions that the derived classes will provide. For example, if class  $D$  is to provide new features  $f$ ,  $g$ , and  $h$  (where each is an attribute, function, or property), then its class interface is

$$DInt = A \mid \text{“}f\text{”} \rightarrow T_1 \mid \text{“}g\text{”} \rightarrow T_2 \mid \text{“}h\text{”} \rightarrow T_3$$

If any of  $f$ ,  $g$ , or  $h$  are functions or properties, then a class definition  $D$  must be provided for  $DInt$ , defining the functions.

We can prove that the child class,  $D$ , or its interface,  $DInt$ , is a subclass of its parent  $A$ .  $D$  and  $A$  are bunches, so we can apply bunch inclusion directly, and it will be

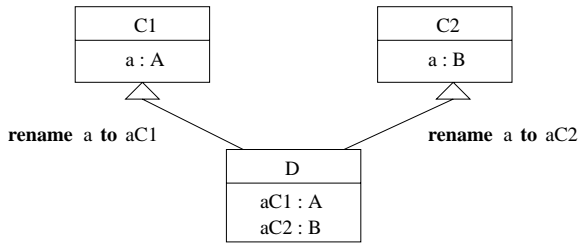


FIGURE 6. Multiple inheritance

the case that  $D : A$ . By definition,  $D$  will be a sub-bunch of  $A$  because every value satisfactory to  $D$  is also satisfactory to  $A$ ; bunch  $A$  includes all its extensions 14. In other words, this formalization of inheritance is the subtyping relationship.

The names of the features (attributes, properties, or functions) of class  $A$  and  $f, g$ , and  $h$  may coincide. If  $f$  is also the name of a feature in  $A$ , then the occurrence of  $f$  in  $DInt$  will *override* the definition of  $f$  in  $A$ . In order for  $DInt$  to remain a subbunch of  $A$ , we shall require that the type of the overriding feature is a subtype of the feature being overridden. We can describe extension-only inheritance relationships by requiring that the names of new features and the names of inherited features do not coincide.

Generalization can also be used to represent multiple inheritance; an example of this is shown in Figure 6. In this example (ignoring the **rename** annotations for a moment), class  $D$  inherits from both class  $C1$  and  $C2$ .  $D$  is both a  $C1$  and a  $C2$ .

Multiple inheritance in object-oriented notations has long been seen as problematic. To see why problems can arise, let us try to formalize Figure 6. We might think that we can immediately formalize Figure 6 in predicative notation, by saying that the interface for class  $D$  is the merge of the class definitions for  $C1$  and  $C2$ , as follows.

$$D = C1 \mid C2$$

However, classes  $C1$  and  $C2$  share a commonly named attribute,  $a$  (the instances of which have different types). We have a *name clash* that should be resolved.

The Eiffel programming language 26 offers a very clean and simple solution to the problems of multiple inheritance, which we adopt. When expressing multiple inheritance, any names that appear in two or more parent classes are renamed to eliminate the clash. If we are to formalize multiple inheritance in predicative notation, we can require that names do not clash, and can use a renaming technique to ensure this. Renaming can be expressed as a substitution on a bunch. If we rename the  $a$  from  $C1$  to  $aC1$ , and the  $a$  from  $C2$  to  $aC2$  in producing  $D$ , then there are no name clashes.

$$\begin{aligned} D &= C1[aC1/a] \mid C2[aC2/a] \\ &= \text{“}aC1\text{”} \rightarrow A \mid \text{“}aC2\text{”} \rightarrow B \end{aligned}$$



FIGURE 7. An association

Renaming is also the way in which repeated inheritance can be handled. If in a derived class we want multiple occurrences of an inherited attribute, we rename accordingly. Otherwise, we can rely on the definition of inheritance as selective union, and simply order the parent classes so as to express the repeated attribute that we want in the child class.

Procedures have been formalized as functions on objects. Inheritance of procedures has not yet been formalized explicitly. However, for such methods, the definition of inheritance that we use means that we can apply all procedures of the parent class to objects of the child class. So procedures of the parent class are inherited by child classes, in the (loose) sense that the methods can be specialized to be used on objects of a child class.

Associations are relationships between classes. An example of a simple association is shown in Figure 7.

The annotated arc between the *Book* class and the *Borrower* class indicates a relationship. The annotations on the relationship indicate cardinality constraints. The association in Figure 7 can be formalized and expressed in predicative notation. Associations are formalized as relations (i.e., functions with a predicate body) on bunches, optionally with constraints added to the participants in the relation.

Consider the *checkedOut* relationship between class *Book* and class *Borrower* in Figure 7. The classes are formalized as bunches with the same names. *checkedOut* is formalized as a relation.

$$\begin{aligned} checkedOut &= \lambda a : Book \cdot \lambda b : Borrower \cdot \\ &\quad \dagger b : 0, 1 \wedge P \end{aligned}$$

where  $P$  is a predicate on bunches  $a$  and  $b$  that expresses any details regarding the relationship (it may simply be  $\top$  if nothing is known).  $P$  can be used to formalize association classes via existential quantification. The cardinality constraint on  $b$  says that  $b$  must represent 0 or 1 instance of class *Borrower*.

Associations can describe many possible relationships between classes. If a formalizer knows more about the meaning of a relationship, then more specific formalizations can be produced. For example, if a client-supplier relationship is specified between *Book* and *Borrower* (e.g., that *Borrower* is a client of *Book*), then the relationship can be formalized by including a reference to bunch *Book* in bunch *Borrower*.

To formalize links between objects, we can first formalize associations between the classes of the linked objects. Then, links can be formalized as instances of

the association (i.e., as a relation applied to objects). In predicative notation, an instance of a relation is a predicate.

#### 4.3.3. Dynamic notations

We have not presented formalizations of dynamic UML notations, such as state transition diagrams or collaborations. With our planned integration of the *OOM* and predicative programming, the use of the predicative notation will supersede the use of dynamic UML notations. Instead of specifying the behaviour of classes using state transition diagrams, behaviour will be specified using predicative notation, by capturing the properties, functions, and procedures of the class. In this manner, the integrated method will be akin to the Business Object Notation 27, where class behaviour is specified by giving feature pre- and postconditions, as well as class invariants.

The value of state transition diagrams in object-oriented specification has been questioned in the past 27. Instead of specifying object or class behaviour by transition diagrams, we use predicative specifications of class features. This has the advantage of more concise specification, and of having a form amenable to rigorous reasoning; transition diagrams have the advantage of being amenable to simulation.

It is possible to infer a state transition diagram for a class from a predicative class specification (which includes specifications of methods). A precondition of a predicative specification implies a guard expressing when a method can be used, while a postcondition implies a set of states resulting from a transition; preconditions can be extracted from specifications by existential quantification. We could also directly formalize state transition diagrams in predicative notation. Predicative notation is suitable for expressing such dynamic behaviour 8, because of its concurrency and communication features. For examples of how to formalize state transition diagrams, see 20 which uses *Z* as the formalization notation. 6 has translations between *Z* and predicative notation.

Other dynamic UML notations, like collaboration diagrams or sequence diagrams, pose no significant difficulties in terms of their formalization in predicative notation. Collaboration diagrams, for example, can be formalized as sequences or parallel compositions of method calls; the predicative notation's process communication and concurrency facilities 8, which can specify asynchronicity, are useful in this task. However, we do not use such diagrams in the case study in the following sections, and so do not present formalizations of such diagrams herein.

The focus in this paper is on integrating methods, rather than on formalizing notations. We are more concerned with obtaining a (possibly partial) formalization that will allow us to use two methods together, rather than in producing a complete formalization of a set of

notations.

#### 4.4. The UML meta-model

The UML meta-model explains how the UML is to be used to produce well-formed specifications. The UML meta-model is itself expressed in UML. The meta-model is not intended for formal reasoning; it is intended to abet hand-specification of UML models, and to help in the production of UML-compliant tools, which require rules to admit or reject UML models.

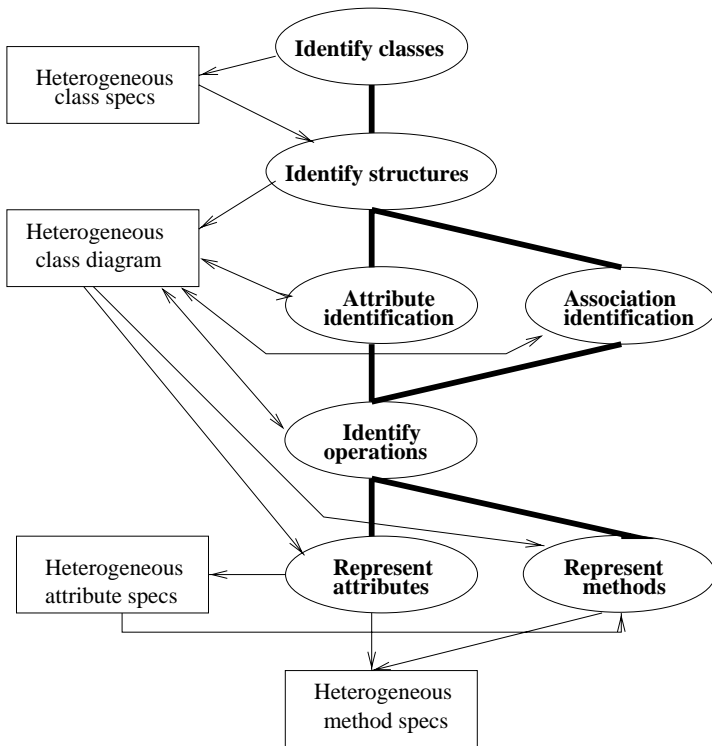
The formalization of UML concepts given previous can be used with the UML meta-model. We have provided a semantics for UML concepts, but have not explained how to use the UML concepts. Effectively, we assume that before applying the formalizations, we have a valid UML specification. Thus, the meta-model is applicable for admitting or rejecting UML specifications. The formalizations that we provide can be used to give a formal meaning to well-formed specifications. As well, since the meta-model is written in UML, our formalizations could be used (with some extension) to formalize the meta-model. This in effect would produce a set of axioms dictating how to produce (formal) UML models. We leave such considerations to another paper. The interested reader might refer to 28 for an approach to formalizing a generic object-oriented meta-model in *Z*.

#### 4.5. The *OOM*-Predicative Method

We continue with applying the meta-method, moving to Step 4: description of informal process relationships. Here, we informally describe how the two microprocesses, of *OOM* and predicative programming, will be used together. We depict an example of a simple technical process for a method that combines *OOM* and predicative programming in Figure 8. Ellipses are steps in the new microprocess (the thick lines between ellipses represent ordering) and boxes represent heterogeneous specifications. Arrows out of and into ellipses denote "production" and "use" of specifications, respectively. The process shown is intended to be representative of steps taken in many object-oriented methods. Different processes could and should be constructed for different development circumstances.

In more detail, the integrated method is as follows.

1. **Identify classes.** These are specified using UML, or using the predicative specification of classes discussed in Section 2; or, a combination. Predicative notation is used for those classes with complex functionality, or those for which the developer wants a rigorous specification for reasons that are dependent on development context.
2. **Identify structures.** Structures of classes, e.g., aggregations and generalizations, are modeled using the UML class diagrams, or by their formalizations as discussed in Section 2.



**FIGURE 8.** Integrated *OOM*-Predicative Programming Process

3. **Attribute and association identification.** Attributes are determined and added to class diagrams or predicative class interfaces of specifications. Attributes can be state components, properties, or function signatures. Associations are drawn between classes, using UML. For associations involving classes with predicative specifications, class diagrams for these entities are drawn, except the diagrams are drawn as *dashed rectangles*, so as to denote their formal nature. See Figure 13 for an example.
4. **Identify operations.** Class operations (procedures in predicative programming) are identified and named. The names are added to the specifications previously generated by adding names to the UML diagrams, or by specifying predicative functions that operate on objects.
5. **Represent attributes.** This can be done using a combination of predicative type definitions and semiformal data notation (e.g., extended BNF). Predicative notation is used for complex data structures, or for those data structures that may be used by predicative specifications of methods.
6. **Represent methods**, i.e., procedures of classes. This can be done using a combination of predicative notation and UML notations (e.g., pseudocode). Predicative notation will be used for writing specifications of methods with complex functionality. Predicative refinement can be used to implement specifications of operations.

In the method, the *extent* to which predicative programming is used remains unspecified. Predicative notation can be used to specify those system aspects that developers think will require use of formalism; informal notations can be used everywhere else. The heterogeneous basis can also be used, during application of the integrated method, to change the extent to which the predicative notation is used.

Formal methods have been integrated with object-oriented methods in the past, e.g., see 20, for a combination of Z with Shlaer-Mellor 29. A difference between the approach of 20 and our own is in the process of the resulting integrated method: there is a direct transformation from the Shlaer-Mellor specifications into Z, and interleaving of process steps from Shlaer-Mellor and the Z established strategy is not directly considered. In the integration presented herein, there need not be a direct transformation from using UML to using predicative programming. Further, in 20, specifications are written in Shlaer-Mellor notation, or in Z, but not both. In the integration of UML and predicative programming, specifications can be heterogeneous.

## 5. CASE STUDY

The case study, for which we present some details, involves constructing a system for carrying out textual analysis—generating statistics for a text document. The system includes a graphics subsystem that is used to display visual representations of collections of gathered statistics (as 3D polygons), and to transform these representations. The statistics gathered are to be used to quantitatively and qualitatively contrast (families of) documents, e.g., for performing authorship detection. A graphical interface for the X Windows system is to be implemented. There are no provisions for handling other display systems, but extensibility is a design goal. The system is to be implemented in C++, for reasons of efficiency, portability, and reusability—we have a small C++ class library at our disposal, consisting of classes for X Windows library software. We shall reuse several classes in the design and implementation.

The system must provide for the computation of the following text-based statistics.

- The number of words in the document.
- The number of sentences in the document. A sentence ends in a period, exclamation mark, quotation mark, or question mark.
- The average word length, number of syllables per word, and sentence length for the document.
- The average number of ‘pause marks’ per sentence. A pause mark is a comma, semicolon, colon, or dash.
- The number of words with at least three syllables.

The system facilities to be offered by an implementation must include the following: *calculate*, *load*, and *save* the above-mentioned statistics for a specified text

document; *plot* loaded statistics in a viewing frame (a three- or two-dimensional space) as 2- or 3-dimensional polygons, with reference axes; *define* each reference axis in terms of the statistic that it represents; and *rotate* the plotted polygons around an axis within the viewing area or volume. Any of the reference axes can be selected, and the rotation may be of arbitrary angle (0–360 degrees).

We use the integrated method from Section 4. The *OOM* process is followed, with UML and predicative notation being applied where appropriate. Predicative notation will be used to specify those classes, objects, and methods that have complex functionality or complex data structures. The use of predicative notation will be focused on those classes for managing calculation of statistics, and on classes that manage communication between other components. The design calculus will also be used to refine these system components into code. For system components that do not present complex functionality, or for components that are reused from existing systems, we will use UML for their specification, and the process of *OOM* for their development.

### 5.1. Identification of classes and objects

The first step in the method of Section 4 is to identify classes and objects. Through study of the problem, we identify a set of classes; we describe select examples here, beginning with several that deal with the graphical representation aspects of the system.

- *GraphicsSystem*: an abstraction that encapsulates the GUI component of the system. It also provides the facilities that interface the GUI subsystem with the textual analysis subsystem.
- *Quantizer*: the abstraction that represents a colour map and the operations to manage it. It is useful to isolate such routines in a class, for possible extensions: in order to build a portable GUI, we may need quantizer objects for different machines and architectures. Also, during rendering, it may be necessary to use different colour maps, and therefore different *Quantizer* objects.
- *Xdriver*: the abstraction that represents and manages the X window related information. *Xdriver* must provide functionality and abstractions to open an X window, draw to an X window, and so on.
- *Clip*: a class that encapsulates representations of viewing frames<sup>3</sup>, as well as routines for clipping drawable objects to fit the viewing window. We may need different clipping approaches for different data sample sizes, and we may need to be able to clip to several different viewing frames.
- *DrawableObject*: an abstract class which is the primitive shape which can be rendered.

<sup>3</sup>A viewing frame is a 2- or 3-dimensional area or volume to which drawable objects are rendered.

- *Axis*: a class that represents an axis of the coordinate system, in particular, the current coordinates of the axis, axis colours, and operations to plot and transform the axis. For this system, we will only want to manipulate two or three axes.
- *Cluster*: a class representing a cluster of document statistics within the graphics system, as well as the operations to plot and transform them. The graphics system may contain any number of clusters.
- *Matrix*: a class that can be used to represent the transformations that can be applied to plotted clusters and axes, as well as operations for constructing and manipulating matrices in their usual ways.
- *Vector*: a class that represents the points (and standard operations thereupon) that make up drawable objects. Points may be two-dimensional or three-dimensional.

Three more abstractions are evident from the requirements: an abstraction to handle the gathering and calculation of the statistics; an abstraction for representing words in documents; and an abstraction to handle the lexical analysis of the documents. We represent these entities formally, because of their complex functionality and data structures, and because we want to develop them formally.

A *Calc* component is an abstraction used to represent the raw data for the textual analysis, as well as for calculating the statistics and generating reports. Its interface and definition (it has no functions) is as follows.

$$\begin{aligned} \text{Calc} = & \text{“wc”} \rightarrow \text{nat} \mid \text{“numletter”} \rightarrow \text{nat} \\ & \mid \text{“syllables”} \rightarrow \text{nat} \mid \text{“totalsyll”} \rightarrow \text{nat} \\ & \mid \text{“ws”} \rightarrow \text{nat} \mid \text{“letter”} \rightarrow \text{nat} \\ & \mid \text{“threesyll”} \rightarrow \text{nat} \mid \text{“numpause”} \rightarrow \text{nat} \end{aligned}$$

Formalization of the object model will ensure that there is exactly one *Calc* object in the system. *wc* is the word count for a document; *sentences* is the sentence count; and *totalsyll* the syllable count. *threesyll* is the number of words with at least three syllables, and *numletter* and *numpause* count the total number of letters and pause marks in the document, respectively.

A *Text* class represents words. Its interface includes two attributes: a string, and a cursor position (which will be useful in counting syllables).

$$\text{TextInt} = \text{“string”} \rightarrow \text{list char} \mid \text{“pos”} \rightarrow 0, \dots, \# \text{string}$$

*TextInt* is now a type, and we can declare as many *TextInt* objects as needed. We can add functions to *TextInt* by selective union, when they are identified. The class definition for *Text* will be based on *TextInt* and will provide definitions for identified functions.

The final formal specification we give is for the lexical analyzer class. Its attributes include three *Text* tokens: one for the current token being analyzed, one for the previous token, and one to hold the title of the document being processed. A filename for the document, an

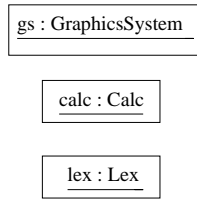


FIGURE 9. Static object specification for system

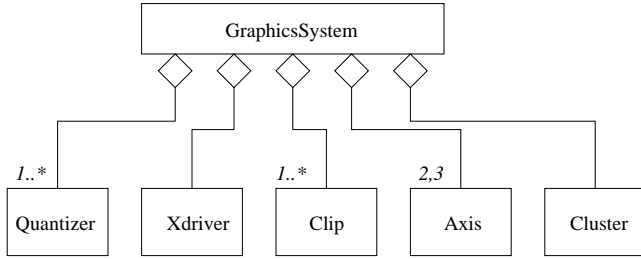


FIGURE 10. Aggregation for graphical representation subsystem

attribute representing the number of sentences in the document and an index into the file are also included.

$$\begin{aligned} Lex = & \text{“title”} \rightarrow Text \mid \text{“token”} \rightarrow Text \\ & \mid \text{“previous”} \rightarrow Text \mid \text{“index”} \rightarrow nat \\ & \mid \text{“filename”} \rightarrow list\ char \\ & \mid \text{“numsentences”} \rightarrow nat \end{aligned}$$

Formalization of the object model will ensure that there is exactly one *Lex* object in the system.

Figure 9 shows the static object model for the classes defined so far: the diagram shows that there should be exactly one instance of *GraphicsSystem*, *Calc*, and *Lex*. The diagram is formalized as a constrained declaration.

$$\begin{aligned} \text{var } calc : Calc \cdot \text{var } gs : GraphicsSystem \cdot \\ \text{var } lex : Lex \cdot \not\llcorner calc = 1 \wedge \not\llcorner gs = 1 \wedge \not\llcorner lex = 1 \end{aligned}$$

## 5.2. Identify structures

Structures among the specified classes and objects can now be determined. We identify two of note. The first structure is for the graphical representation subsystem. It can be depicted as an aggregation. The semantics of such relations are expressed in predicative programming. We draw the aggregation in Figure 10.

We also note that the *Axis* and *Cluster* classes share common features: they are derivations of the *DrawableObject* class. This structure is depicted in Figure 11.

*DrawableObject* is a data abstraction for arbitrary drawable shapes (its drawing routines are typically implemented as polygon renderers). *Axis* and *Cluster* specialize *DrawableObject* to particular subtypes. In the derived classes, it may be useful to override plotting and transformation methods from the base class. Very simple transformation methods and very simple plotting routines can be used to draw an axis, for example.

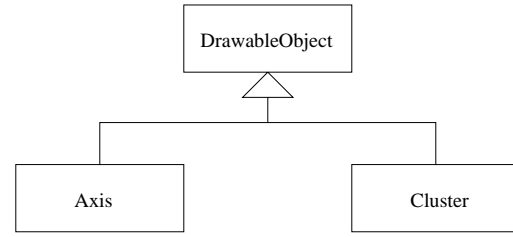


FIGURE 11. Generalizations for drawable objects

## 5.3. Identify attributes, operations, and associations

The next phases involve identifying operations and attributes, as well as specifying associations. Detailed specifications of operation behaviour will be produced in the design phases. Here, we provide informal definitions for some examples: the *GraphicsSystem*, *Vector*, and *Cluster* objects, and those objects formally specified earlier.

*GraphicsSystem* Its methods will include:

- *Plot*: plot a requested number of axes (typically two or three), and all clusters represented in the system. The number of axes requested will dictate the number of coordinates used in plotting a cluster.
- *MakeTransforms*: produces the affine transformations required to transform a cluster or an axis as requested by the user.

Attributes of *GraphicsSystem* will include: *camera*, a description of the camera position and orientation of the viewer; and *viewport*, a description of the two-dimensional device coordinate viewing area where all pixel plotting will be performed. It will also include an arbitrary number of *Clusters*, as well as several axes. The class is shown in Figure 12.

*Vector* The methods of a *Vector* class include:

- *Add*: add two vectors
- *CrossProduct*: calculate a normal vector to two vectors
- *Length*: calculate the  $L_2$ -norm length of the vector

The class is shown in Figure 12.

*Cluster* Methods of the *Cluster* class include:

- *Plot*: plot the cluster in the viewing frame and under the constraints dictated by the *GraphicsSystem*.
- *Transform*: apply a transformation to the cluster, and then replot the result.

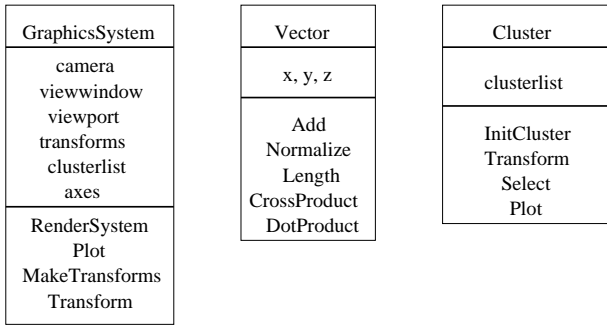


FIGURE 12. The *Vector*, *Cluster*, and *GraphicsSystem* classes

An attribute of the class is the arbitrary-length list of *Vectors* representing some gathered statistics, *coordinates*. The class is also shown in Figure 12.

We now define operations for those components that earlier had attributes formally specified. We provide specifications of the operations when we get to design; here, we are concerned with naming and understanding the purpose of operations applicable to the classes.

*Calc* Methods of *Calc* include:

- *UpdateStatistics*: update the statistics by analyzing and computing statistics for the current token emitted by the lexical analyzer.
- *Report*: calculate further statistics, e.g., document readability, and prepare a formatted report on the document for the user.

*Text* Methods include:

- *Length*: return the length of a *Text*.
- *Substring*: given an initial position in the *Text* and a substring length, return the corresponding substring of the *Text*.
- *CountSyllables*: the number of syllables in the *Text*.

The interface specification for the *Text* class is now

$$\begin{aligned}
 \textit{TextInt} &= \textit{"string"} \rightarrow \textit{list char} \\
 &| \textit{"pos"} \rightarrow \textit{nat} \\
 &| \textit{"Length"} \rightarrow \textit{nat} \\
 &| \textit{"Substring"} \rightarrow \textit{list char} \\
 &| \textit{"CountSyllables"} \rightarrow \textit{nat}
 \end{aligned}$$

We fill in the definitions for the functions later.

*Lex* The state of any *Lex* object records the lookahead *Text* token, as well as the current *Text* token. Operations include:

- *Init*: initialize the lexan with a filename.
- *GetToken*: get the next token from the specified file and invoke the statistical analyzer upon it.

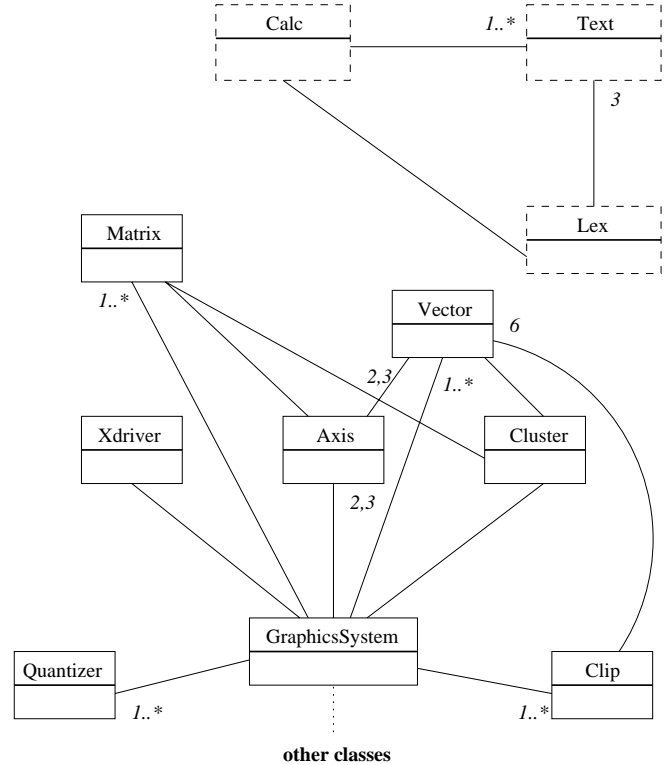


FIGURE 13. Associations for the example

We now draw associations. Associations are shown in the class diagram of Figure 13 (in the figure, we do not specify all attributes and methods for each classes, nor do we describe all the classes). Some classes have predicative specifications. We denote these, in Figure 13, as UML diagrams but with dashed boxes. In this manner, a reader of the diagram can immediately see the breadth of application of the predicative method. The diagram can also provide a reader with exact information on when it will be necessary to interface classes written in predicative notation with classes written in UML notation. It is these interfaces that may be the most complex to specify and design, since they will involve both formal and informal partial specifications. The heterogeneous basis is necessary to give a formal meaning to these combinations, and to refine the combinations to the level of code.

We could also use the basis to formalize the diagrams completely (in predicative notation), so as to check the consistency of our specifications, or to determine what is missing from the specification.

Here is an example. Consider the associations between classes *Calc*, *Lex*, and *Text*. These can be formalized (according to Section 4) as

$$\begin{aligned}
 \textit{ScanUses} &= \lambda l : \textit{Lex} \cdot \lambda t : \textit{Text} \cdot \wp l = 1 \wedge \wp t = 3 \\
 \textit{CalcUses} &= \lambda c : \textit{Calc} \cdot \lambda l : \textit{Lex} \cdot \wp l = 1 \wedge \wp c = 1 \\
 \textit{CalcHas} &= \lambda c : \textit{Calc} \cdot \lambda t : \textit{Text} \cdot \wp c = 1 \wedge \wp t \geq 1
 \end{aligned}$$

The static model in Figure 9 states that, for example, there is one instance of *Lex* in the system. The associ-

ation involving *Lex* must be consistent with the static model This means that we must prove

$$ScanUses(lex) \Rightarrow \dagger lex = 1 \wedge CalcUses(lex) \Rightarrow \dagger lex = 1$$

where, recall, *lex* is the name of the declared lexical analysis variable. This proof is straightforward. Thus, we have shown that the partial class diagram in Figure 13 is consistent with the static object model for the lexical analysis component.

#### 5.4. Data representation and functional design

We now provide designs for some of the attributes and methods of classes. We commence by describing the methods for the formally-specified objects, since they already have much of their data described.

The *Calc* class can be used to to represent the gathered statistics, and has methods for calculating derived values, and for generating reports. We now specify and properly name a few operations for *Calc*. Before proceeding, we specify the bunch of vowels, *vowel*.

$$vowel = 'a, 'e, 'i, 'o, 'u, 'y$$

(In predicative notation, character constants are prefixed with left apostrophes.) The *Init* method for class *Calc* takes a *Calc* object and sets its attributes to 0.

$$\begin{aligned} Init &= \lambda c : Calc \cdot \\ &\quad \text{"numletter"} \rightarrow 0 \mid \text{"letter"} \rightarrow 0 \\ &\quad \mid \text{"wc"} \rightarrow 0 \mid \text{"sentences"} \rightarrow 0 \\ &\quad \mid \dots \mid c \end{aligned}$$

The method *UpdateStats* applies to a *Calc* object and a *Text* object; it adjusts the document statistics to take into account the statistics associated with the *Text* object. The *Text* object argument is the dataflow returned from a message to the *lex* object from the *calc* object; in particular, it will be the current value of *lex* "token".

$$\begin{aligned} UpdateStats &= \lambda c : Calc \cdot \lambda t : Text \cdot \\ &\quad \text{"numletter"} \rightarrow t \text{"Length"} \\ &\quad \mid \text{"wc"} \rightarrow c \text{"wc"} + 1 \\ &\quad \mid \text{"totalsyll"} \rightarrow c \text{"totalsyll"} + \\ &\quad \quad t \text{"CountSyllables"} \\ &\quad \mid \text{"threesyll"} \rightarrow c \text{"threesyll"} + \\ &\quad \quad \text{if } t \text{"CountSyllables"} \geq 3 \text{ then } 1 \\ &\quad \quad \text{else } 0 \\ &\quad \mid \dots \mid c \end{aligned}$$

We provide one operation for the *Lex* object. The initialization operation, *Init*, is straightforward. We specify the method, *GetToken*, for acquiring a token from a text file. First, we define the *wordSeq* of a document, the sequence of all words in the document; this is based on the bunch of all non-whitespace strings in the

document. We assume that any document that we are studying has been prefixed and suffixed with a space (so that we do not have to consider special cases). The interface for *wordSeq* is

$$wordSeq : list \ char \rightarrow list \ word$$

Its definition is

$$\begin{aligned} \forall d : list \ char \cdot \forall s : list \ word \cdot \\ (wordSeq \ d = s) &= (filter \ d = flatten \ s) \wedge \\ s(0, .. \#s) &= lexicon \ d \end{aligned}$$

*pWSpace* is the bunch of all punctuation and white space characters. *word* is *list (char  $\ominus$  pWSpace)*, where  $\ominus$  is bunch difference. The *lexicon* is the bunch of words in a list *d*.

$$\begin{aligned} lexicon &= \lambda d : list \ char \cdot \S w : word \cdot \\ &\quad \exists sp1, sp2 : pWSpace; i, j : 0, .. \#d \cdot \\ &\quad d[i, ..j] = [sp1]^+ w^+ [sp2] \end{aligned}$$

*filter d* produces a list that contains only those characters in list *d* that are not in *pWSpace*, while maintaining the character ordering from *d*.

$$\begin{aligned} filter \ [nil] &= [nil] \\ \forall x : word \cdot filter \ [x] &= [x] \\ \forall y : pWSpace \cdot filter \ [y] &= [nil] \\ \forall w, z : list \ char \cdot filter \ x^+ y &= (filter \ x)^+ (filter \ y) \end{aligned}$$

*flatten* is distributed concatenation; given a list of lists of type *T*, the result is the concatenation of all lists in the argument, with order maintained.

$$\begin{aligned} flatten \ [nil] &= [nil] \\ \forall s : list \ T \cdot flatten \ [s] &= s \\ \forall q, r : list \ list \ T \cdot flatten \ q^+ r &= (flatten \ q)^+ (flatten \ r) \end{aligned}$$

The operation *GetToken*, which obtains a token from a text file *f*, can be specified using these functions. Note that the token will contain letters and numbers only.

$$\begin{aligned} GetToken &= \lambda l : Lex \cdot \lambda stream : list \ char \cdot \\ &\quad \text{"previous"} \rightarrow l \text{"token"} \\ &\quad \mid \text{"token"} \rightarrow wordSeq(stream)(l \text{"index"}) \\ &\quad \mid \text{"index"} \rightarrow l \text{"index"} + 1 \mid l \end{aligned}$$

The *TextInt* class interface was defined earlier. We have also identified several methods of the class, and added them to the interface specification. We now have to define the functions *Length*, *Substr*, and *CountSyllables*. This is done by providing a restriction

on the *TextInt* bunch type, so that the new type produced is only those elements satisfying the description of the functions. The new class definition, *Text*, is

$$\begin{aligned} \textit{Text} = \& t : \textit{TextInt} \cdot (t\textit{Length} = \#t\textit{string}) \wedge \\ & (t\textit{CountSyllables} = \textit{count}) \wedge \\ & (t\textit{Substring} = \textit{substr}) \end{aligned}$$

The definitions of the functions *Substring* and *CountSyllables* are as follows. Note that the name *t* is bound in the definition of *Text*.

$$\begin{aligned} \textit{substr} = \lambda \textit{start}, \textit{len} : \textit{nat} \cdot \\ t\textit{string}[\textit{start}; ..\textit{start} + \textit{len}] \end{aligned}$$

If  $\textit{Length} \leq 0$ , or if  $\textit{start} + \textit{len} \geq \textit{Length}$ , then the *substr* operation produces the empty string, *[nil]*.

The *CountSyllables* method is more complex; it must calculate the number of syllables in a *Text* object. In general, in an English word, a syllable occurs when a vowel is followed by a non-vowel. But because of the complexity of the English language, the general rule for calculating the number of syllables does not hold for certain word fragments, or even particular words. There are a large number of such cases, which must be dealt with individually. These special-case ‘filters’ are expressed in *count* as the partial specifications *SE*, *IO*, *LE*, et cetera. We describe their details shortly.

$$\begin{aligned} \textit{count} = \textit{max}(1, \textit{VnonV}) + \\ \textit{IO} + \textit{ES} + \textit{SE} + \textit{LE} + \textit{YI} + \textit{IA} + \\ \textit{UE} + \textit{II} + \textit{UI} + \textit{ED} + \textit{EE} \end{aligned}$$

where *VnonV* (“vowel followed by non-vowel”) is

$$\begin{aligned} \textit{max}(1, \& \& i : 0, ..\#t\textit{string} - 1 \cdot \\ (t\textit{string}(i) : \textit{vowel} \wedge \neg t\textit{string}(i+1) : \textit{vowel})) \end{aligned}$$

The estimate of the number of syllables is calculated by determining the number of occurrences of vowels followed by non-vowels in *tstring*.

Because there does not appear to be a precise way to define what it means to be a syllable of an English word, we are left with enumerating special cases that describe when to adjust the syllable count given by the general rule. The partial specifications of the form *IO*, *ES*, et cetera, at the end of *count*, deal with specific word fragments. In particular, *ES* is a specification dealing with words that have *ES* endings (e.g., *tides*), and *IO* deals with words that contain an *io* fragment (e.g., *ion*, *caution*). These specifications adjust the syllable count calculated from the general rule. We provide two examples of these filters here; others can be found in 10.

Words with an ‘io’ fragment will have their syllable count incremented if the ‘io’ is at the beginning of the word (e.g., as in *ion*), or if the ‘io’ is immediately preceded by a hard sound (e.g., *caution*).

$$\textit{IO} = \& \& j : 1, ..\#t\textit{string} - 1 \cdot$$

$$\begin{aligned} (t\textit{string}(j+1) = 'i' \wedge t\textit{string}(j+2) = 'o' \\ \wedge t\textit{string}(j-1) : 'g, 'm, 'n, 's, 't, 'x, 'z) + \\ \textit{if } t\textit{string}[0; ..2] = ['i; 'o] \textit{ then } 1 \textit{ else } 0 \end{aligned}$$

Words ending in ‘es’ are slightly tricky: for an ‘es’ ending that is not prefixed by a ‘hard’ sound (i.e., one of the letters *c*, *g*, *h*, *s*, *x*, *z*) after a vowel, or ends in *les*, we decrement the count of syllables by one.

$$\begin{aligned} \textit{ES} \hat{=} -\& \& j : \#t\textit{string} - 4 \cdot \\ (t\textit{string}(j+2) = 'e' \wedge t\textit{string}(j+3) = 's' \\ \wedge \neg t\textit{string}(j+1) : 'c, 'g, 'h, 's, 'x, 'z \wedge \\ (t\textit{string}(j+1) = 'l' \vee t\textit{string}(j) : \textit{vowel})) \end{aligned}$$

We should now describe operations for the informally specified classes. This can be done using pseudocode, programming language code (i.e., in C++), or structured English. We assume that any informal specifications can be formalized in predicative notation. We omit informal descriptions of operations, though several examples can be found in 10. In completing the case study, we used both pseudocode and C++ code. The C++ code consisted of two reused classes (a *Quantizer* class and an *Xdriver* class), which were extended to fit into the new context.

## 5.5. System integration and implementation

We have specified attributes and methods for some of the classes and objects in the system. What remains is to implement the methods and attributes of each class—via refinement and writing C++ code—provide a suitable user-interface, and integrate the subsystems. We provide only brief details of this step. Here, we concentrate on refining and implementing the formal parts of the specification.

We start with the refinement of the operations that calculate the number of syllables in a word. This operation, *CountSyllables*, is a method of class *Text*. Monotonicity properties of refinement on predicative specifications 8 state that operations can be refined by parts. Since our heterogeneous specifications have a predicative semantics via the heterogeneous basis, we can refine *CountSyllables* and leave other methods of class *Text* and all other classes—whether they are specified in UML or in bunch notation—untouched, and obtain a refinement of the entire specification. Further, the calculation is done in a number of cases: a case that applies a general rule to a word, and several special cases. Because of refinement by cases, we can refine each separately. We commence with the general case. Define *P* to be the predicate

$$\begin{aligned} \& \& i : 0, ..\#t\textit{string} - 1 \cdot \\ (t\textit{string}(i) : \textit{vowel} \wedge \neg t\textit{string}(i+1) : \textit{vowel}) \end{aligned}$$

and let *Q* be the same as *P*, but with *j* in place of the 0 in the range of *i*. *P* is part of the general syllable

count case in *countbody*; the  $t$  is free in  $P$ , but bound in *countbody*—it is the *Text* object for which a syllable count is being carried out. To refine  $P$ , we start by introducing an iteration variable,  $j$ , bound locally by a lambda. We use functional refinement as defined in an earlier section.  $P$  is refined by the lambda abstraction when we supply a value 0 for  $j$ .

$$P \quad \vdash \quad (\lambda j \cdot Q) \ 0$$

We then refine the body of the new function, i.e.,  $Q$ , producing a recursive program.

$$Q \quad \vdash \quad \mathbf{if} \ j = \#t\text{“string”} - 1 \ \mathbf{then} \ 0 \ \mathbf{else} \\ (\lambda j \cdot Q) \ (j + 1) + \mathbf{if} \ isSyllable \ \mathbf{then} \ 1 \ \mathbf{else} \ 0$$

The condition *isSyllable* is the boolean expression that identifies an occurrence of a syllable in a *Text* object  $t$ .

$$isSyllable = t\text{“string”}(j) : vowel \wedge \neg t\text{“string”}(j + 1) : vowel$$

Refinement of the syllable counting mechanism continues with the partial specifications that handle special-case word fragments. We refine *IO* using a tail-recursive approach. First, define *ioCase* as follows.

$$ioCase \quad = \quad (t\text{“string”}(i + 1) = 'i \wedge \\ t\text{“string”}(i + 2) = 'o \wedge \\ t\text{“string”}(i - 1) : 'g, 'm, 'n, 's, 't, 'x, 'z)$$

The *IO* specification has the following form.

$$IO = (\#j : 1, ..t\text{“Length”} - 1 \cdot ioCase) + \\ (\mathbf{if} \ t\text{“string”}[0; ..2] = ['i; 'o] \ \mathbf{then} \ 1 \ \mathbf{else} \ 0)$$

The second operand of the operator  $+$  requires no refinement. We refine the first operand. Call this operand  $S$ , and let  $T$  be the same as  $S$ , except with  $i$  substituted for 1 in the range of  $j$ .  $S$  is implemented as follows.

$$S \quad \vdash \quad (\lambda i \cdot T) \ 1 \\ S[i/1] \quad \vdash \quad \mathbf{if} \ i = t\text{“Length”} - 1 \ \mathbf{then} \ 0 \\ \mathbf{else} \ (\mathbf{if} \ Q[i/j] \ \mathbf{then} \ 1 \ \mathbf{else} \ 0) + \\ (\lambda i \cdot T) \ (i + 1)$$

We direct the reader to 10 for the details of implementing the methods and data structures that were not formally specified.

The refinements and implementation steps that we have taken produce functional programs that are typically recursive. In coding the implementations, we removed much of the recursion (which is primarily tail recursion), and rewrote it using loops, for reasons of efficiency. A transliteration of recursive refinement structures to **while**-loops is given in 8, and was the basis of this coding process.

Integration of the system components according to the instance diagram of Figure 13 must now occur. The final C++ implementation of the system is approximately 4500 lines of code, containing 24 classes (with a cumulative 116 attributes and 167 methods).

## 5.6. Discussion

We have demonstrated the use of a combined method, where predicative programming and UML have been integrated. The problem that was solved—a graphical plotter for a statistics gathering engine—was non-trivial; the resulting system was implemented in approximately 4500 lines of C++ code. The development demonstrated several noteworthy points.

- The integrated methods each provide different facilities.
  - The *OOM* process supports the development from the construction of an initial system model, through specification, design, and implementation. It provides guidance for the modeling of classes and their relationships.
  - Predicative programming supports the writing of formal specifications of classes, methods, and data in a wide-spectrum language. It also supports their algorithm and data refinement to code.
- The use of the formal method was restricted to the specification of those system components that had complex functionality or data structures. The use of the formalisms could have been expanded or reduced as necessary, to better fit the needs of different specifiers and developers. The formal semantics and heterogeneous basis effected a scheme that allows such changes to take place even during development. It was briefly shown how to use the formal semantics to verify properties. The translations from UML to predicative notation could have been used further to explicitly formalize the specifications. Then, the specifications could have been analyzed, and missing information (e.g., method specifications) fed back in to the UML-based development.
- Reuse of code occurred. The *Xdriver* and instances of the *Quantizer* classes reused several methods from libraries and existing systems. This did not adversely affect the production of heterogeneous specifications, or the construction of the overall system. This was likely due to the encapsulation and information hiding provided by the class and object concepts. Reuse in a setting without clearly-defined interfaces might not be as straightforward.

In using the integrated method, we were able to scale the formal method up to be applicable to larger sized developments than it has been applied to in the past. We were able to eliminate problems with both methods (e.g., applicability to large problems with predicative programming, informality with *OOM*). We were also able to show that the two methods, which demonstrate very different approaches towards developing systems, could be used together successfully.

## 6. SUMMARY AND CONCLUSIONS

We have presented an application of a meta-method for formal method integration. The meta-method was designed as a design or thinking tool for method integration. It can be used to provide guidelines and support to method engineers as they are deciding roles that individual methods will play in a new integrated technique. The meta-method was used in combining an object-oriented method and predicative programming style. The integrated technique was applied to a problem for which we provided examples of specifications and designs.

The approach that we have demonstrated is not limited to combinations of object-oriented methods that use UML and predicative notation. Use of Z, VDM, Object-Z, or other formal methods is also possible. One way to introduce Z in an inexpensive fashion is to define translations between Z and predicative programming. In effect, this constructs a heterogeneous basis that is a generalization of the one in Figure 1. Then, compositions of partial specifications involving Z, predicative notation, and UML can be formally defined, via translation into homogeneous specifications written in, e.g., predicative notation. A complication with this approach is that Z and predicative notation are not equivalent in terms of their expressiveness 6. In using Z and predicates together, restrictions on notation use may have to occur, depending on how the semantics of the composed notation is defined. Another approach would be to use a third notation, e.g., pre- and postcondition pairs, to give a semantics to combinations of Z, UML, and predicative specifications.

The approach that we have used is not limited to methods that use UML. UML and the *OOM* share many common characteristics with notations from other methods, e.g., Shlaer-Mellor, Coad-Yourdon, BON, OMT, et cetera. The formalizations that we have used could be modified to be applicable in integrations involving these other methods. An integration of Z and BON is presented in 15; it followed the general approach used in this paper.

The meta-method, as we have designed it and used it, may be insufficient for all method integration situations. Because the meta-method does not construct or require meta-models 5 for integrating methods, the meta-method may not be appropriate for situations in which integrated methods are to be supported by integrated tool suites. In such settings, meta-modeling approaches seem to be necessary. However, the meta-method could be applied as a *front-end* to the more powerful, general-purpose meta-modeling approaches: the meta-method can be used to help determine roles of methods, to identify complementarity among methods, and to determine problems that may arise in linking methods together, e.g., due to semantic or syntactic differences in notations that are used in the methods. If we need more rigour in descriptions of process relation-

ships, then we can turn to alternative techniques, and can still use the knowledge acquired during application of the meta-method.

In order to integrate methods and obtain the most useful new technique, it is important to clarify roles for the methods being integrated, and to determine problems with the methods that may lead to complications in defining their interrelationships. Part of the purpose for the meta-method used herein is to systematize these aspects. And if we find that we can integrate methods by using the meta-method described here, without needing to apply meta-modeling techniques, then we will have reduced the costs of applying a method integration process.

## ACKNOWLEDGEMENTS

Thanks to Rick Hehner, Jonathan Ostroff, and the anonymous referees for their advice and corrections. This research was supported with the assistance of the National Sciences and Engineering Research Council.

## REFERENCES

- [1] Jackson, M.A. (1995) *Software Requirements and Specifications*. Addison-Wesley.
- [2] Kronl6f, K. (ed.) (1993) *Method Integration: Concepts and Case Studies*. Wiley.
- [3] Polack F., Whiston, M., and Mander, K.C. (1993) The SAZ Project: Integrating SSADM and Z. In *Proc. FME '93: Industrial-strength Formal Methods*, LNCS 670, Springer-Verlag.
- [4] Semmens, L.T., France, R.B., and Docker, T.W. (1992) Integrated Structured Analysis and Formal Specification Techniques. *The Computer Journal* **35**(6).
- [5] MetaPHOR Project Group. (1994) MetaPHOR: Meta-modeling, Principles, Hypertext, Objects and Repositories. Technical Report TR-7, University of Jyvaskyla.
- [6] Paige, R.F. (1997) A Meta-Method for Formal Method Integration. In *Proc. FME '97*, LNCS 1313, Springer-Verlag.
- [7] Saeki, M. (1998) A meta-model for method integration. *Information and Software Technology* **39**(14-15).
- [8] Hehner, E.C.R. (1993) *A Practical Theory of Programming*. Springer-Verlag.
- [9] Fowler, M. and Scott, K. (1997) *UML Distilled*. Addison-Wesley.
- [10] Paige, R.F. (1997) *Formal Method Integration via Heterogeneous Notations*. PhD Dissertation, University of Toronto, November 1997.
- [11] Zave, P. and Jackson, M. (1993) Conjunction as Composition. *ACM Trans. on Software Engineering and Methodology*, **2**(4).
- [12] Hall, A. (1996) Using Formal Methods to Develop an ATC Information System. *IEEE Software*, **13**(2).
- [13] Graham, I., Henderson-Sellers, B., and Younessi, H. (1997) *The OPEN Process Specification*. Addison-Wesley.
- [14] Hehner, E.C.R. (1981) Bunch Theory: a simple set theory for computer science. *Information Processing Letters*, **12**(1).

- 
- [15] Paige, R.F. and Ostroff, J.S. (1998) From Z to BON/Eiffel. In *Proc. Automated Software Engineering 1998*, IEEE Press, October 1998.
  - [16] Paige, R.F. (1998) Comparing Extended Z with a Heterogeneous Notation for Reasoning about Time and Space. In *Proc. ZUM '98*, LNCS 1439, Springer-Verlag.
  - [17] Brinkkemper, S., Lyytinen, K., and Welke, R. (1996) *Method Engineering*. Chapman and Hall.
  - [18] Kruchten, P. (1999) *The Rational Unified Process*. Addison-Wesley.
  - [19] Abadi, M. and Cardelli, L. (1996) *A Theory of Objects*. Springer-Verlag.
  - [20] Hammond, J. (1994) Producing Z Specifications from Object-Oriented Analysis. In *Proc. ZUM '94*, Springer-Verlag.
  - [21] Paige, R. (1999) When are methods complementary? *Information and Software Technology*, **41**(3).
  - [22] Hall, A. (1994). Specifying and Interpreting Class Hierarchies in Z. In *Proc. ZUM '94*, Springer-Verlag.
  - [23] Bourdeau, R. and Cheng, B.H.C. (1995) A formal semantics for object model diagrams. *IEEE Transactions on Software Engineering*, October 1995.
  - [24] Evans, A. (1998) Reasoning with the Unified Modelling Language. In *Proc. WIFT '98*, IEEE Press, October 1998.
  - [25] Evans, A. and Clark, A. (1998) Foundations of the Unified Modelling Language. In *Proc. 2nd Northern Formal Methods Workshop*, Springer-Verlag.
  - [26] Meyer, B. (1997) *Object-oriented Software Construction* (Second Edition). Prentice-Hall.
  - [27] Walden, K., and Nerson, J.-M. (1995) *Seamless Object-Oriented Software Architecture*. Prentice-Hall.
  - [28] Mistic, V. and Moser, S. (1997) Formal Approach to Metamodeling: A Generic Object-Oriented Perspective. In *Proc. ER '97*, LNCS 1331, Springer-Verlag.
  - [29] Shlaer, S. and Mellor, S.J. (1992) *Object Lifestyles - Modeling the World in States*. Prentice-Hall.
-