Case Studies in using a Meta-Method for Formal Method Integration

Richard F. Paige

Department of Computer Science, University of Toronto, Toronto, ON, Canada, M5S 3G4. paige@cs.toronto.edu[⋆]

Abstract. We summarize the results of several experiments in applying a meta-method for formal method integration [18, 19]. We provide a small example of using an instance of integrated formal methods, and discuss properties and difficulties associated with applying the meta-method to combining and using several formal and semiformal methods.

1 Introduction

Method integration involves defining relationships between different methods so that they may be cooperatively used together. In software engineering, method integration has seen recent research on combining specific methods [15, 25], and on the formulation of systematic techniques [14, 18, 19]. In this paper, we follow the latter theme and describe several case studies in the application (and use of the products) of a meta-method for formal method integration [18, 19].

We commence with a very brief overview of method integration and the means we take to accomplishing it. Our lightweight technique is based on *heterogeneous notations*, combinations of formal and semiformal notations. After providing background for heterogeneous notations, we summarize a meta-method for integrations involving at least one formal method [19] used for system specification and design, and then describe several case studies in which the meta-method has been applied. With one case study, we provide a very small example using integrated methods, in order to give the flavour of the approach.

Due to space restrictions, this paper only summarizes the results of several case studies. The interested reader may find details in [18, 19, 20, 21].

1.1 Method integration

Method integration involves the resolution of incompatibilities between methods, so that the approaches can be safely and effectively used together [14]. Method integration in a software engineering context is of growing research interest. A reason for this is the low likelihood that one method will always suffice in the development of complex systems [5, 12]. Method integration has also been used and has been shown to be useful in practice in various forms, e.g., at BT [25], Westinghouse [9], Praxis [8], and elsewhere.

[⋆] *Current address:* Department of Computer Science, York University, North York, ON, Canada, M3J 1P3. paige@cs.yorku.ca

1.2 Heterogeneous notations and specifications

A notation is an important part of any method. Notations are used to specify system behaviour and to clarify requirements. Notations play an important role in how we carry out formal method integration. Specifically, we combine notations as a first step towards combining formal methods with other methods, all of which that are being used for system specification and design. A *heterogeneous notation* is a combination of notations that is used to write heterogeneous specifications.

Definition 1. A specification is *heterogeneous* if it is a composition of partial specifications written in two or more notations.

We do not place any constraints on what we mean by "composition". Useful compositions will depend on the problems to be solved, and the context in which the set of notations is to be used. Useful compositions may be specification combinators, shared state or names, or synchronization on events, for example.

Heterogeneous notations are useful for a number of reasons: for writing simpler specifications than might be produced using a single language [28]; for ease of expression [2]; and because they have been proven to be useful in practice [8, 29].

The formal meaning of a heterogeneous specification is given by defining the semantics of all the notation compositions. Formal meaning is provided by a heterogeneous basis.

Definition 2. A *heterogeneous basis* is a set of notations, translations between formal notations, and formalizations of semiformal notations.

The heterogeneous basis used in the work summarized in this paper is completely presented in [18]. We discuss it in the next section, and summarize the process of its construction in Section 2.

1.3 A heterogeneous basis

A heterogeneous basis supplies a formal semantics to a heterogeneous specification [18]. It is also used to provide the first step on which formal method integration is carried out. The basis used in the case studies in this paper consists of a set of languages with mappings defined between them. It is depicted in Figure 1.

The predicate notation is from [11]; Z is from [26]; specification statements (i.e., $w : [pre, post]$) are from [17]; and the Larch languages are from [6]. The remaining semiformal notations are from SA/SD [4], and Coad-Yourdon object oriented analysis and design [3]. More notations are considered in [18, 19].

In the diagram, the arrows represent translations. The translations in Figure 1 are described in detail in [18]. We will give a few examples of translations involving formalisms in Section 2, and will summarize some example formalizations involving semiformalisms. We will also describe the general approach we take to constructing a heterogeneous basis that consists of formal and semiformal notations.

A specification written using combinations of the notations from Figure 1 is given a semantics using only one of the formal notations of the basis. The user of the basis

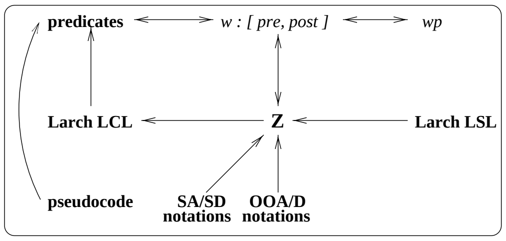

Fig.1. A heterogeneous basis

chooses the formal notation to use for their particular context, and defines the meaning of a heterogeneous specification in this formal notation. Syntactic complications must still be dealt with by the specifier, in particular, ensuring that combinations of different notations can be parsed.

1.4 Integrating methods with heterogeneous notations

Heterogeneous notations are used as a first step in formal method integration. In other words, we carry out formal method integration (for system specification and design methods) by *first combining the notations on which the methods rely.* The combination process occurs by constructing (or extending) a heterogeneous basis consisting of the notations of interest, and by resolving syntactic differences among the notations, so that combinations of notations can be unambiguously parsed. Once this is done, the method integration process can continue by generalizing method steps to use heterogeneous notations, and by embedding, ordering, interleaving (perhaps generalized) method steps from two or more different methods.

We do not claim that this is the only approach to combining formal methods with other methods. It is a lightweight technique that produces integrated methods with convenient properties (see Section 3, and the case studies) when applied to system specification and design methods, and has been used productively on a number of moderate-sized examples. One suggestion we make is that formal method integration can be partly systematized by using heterogeneous notations, since their use allows notation-related complications of method integration to be dealt with first, while the remainder of the integration process can concentrate on manipulating the steps of the methods to be combined.

1.5 Overview

We commence the paper by summarizing the process for constructing a heterogeneous basis. We then outline a meta-method for formal method integration, and clarify the role that heterogeneous notations play within it. We summarize the results of a number of example integrations involving only formal methods. One integration is applied to a very small example. Two larger integration case studies, involving semiformal methods, are outlined. We then discuss the benefits and limitations of each integration.

2 A Heterogeneous Basis

A heterogeneous basis is shown in Figure 1. This basis is created by translation: a set of mappings are given that transform a specification in one notation into a specification in a second notation. These mappings can then be used to provide a formal semantics to compositions of partial specifications written in the notations of the basis.

In this section, we summarize the process of constructing the heterogeneous basis of Figure 1. In particular, we consider the addition of formal notations to a basis (by translation), and the process of adding semiformal notations to an existing basis (by formalization). We only summarize the details here, due to space constraints; full translations, formalization procedures, and examples are given in [18].

2.1 Adding formal notations to a heterogeneous basis

A formal notation may be added to a heterogeneous basis by

- providing a translation from the formal notation into a notation already in a heterogeneous basis; and
- analyzing the expressive capabilities of each notation, viz., what can and cannot be translated with preservation of interpretation, and what is the effect of these notation differences on translation and on assigning a semantics to compositions.

In constructing the heterogeneous basis shown in Figure 1, we commenced with existing translations already presented in the literature. These included: the mapping of Z to specification statements [13]; the wp definition of specification statements [17]; and, the wp expression of an early version of a predicative programming specification [10]. To these translations, we added others. We provide several examples here, one mapping predicates without time variables into specification statements; a second mapping predicates with time variables into specification statements; and a third, mapping specification statements into Z. The translations are described as functions from notation to notation (we explain ways to handle expressive differences shortly). To simplify the process, we assume all translations use the primed/unprimed notation of Z for representing post and pre-state. More example translations are given in [18].

A predicate specification $\mathbf{frame}\ w \bullet P$ that does not refer to the time variables t, t' [11] can be translated to a specification statement [17] using the mapping $PredToSS$.

$$PredToSS(\mathbf{frame}\ w \bullet P) \mathrel{\widehat=} w : [\ true, P\ ]$$

Similar translations arise when dealing with predicate specifications that make reference to time variables t and t'. For a predicate specification $\mathbf{frame}\ w \bullet P$ that includes references to t and t' in P, we translate it using $TimedPredToSS$.

$$\begin{aligned}&TimedPredToSS(\mathbf{frame}\ w \bullet P) \mathrel{\widehat=}\\ &\quad w : [\ \forall\, t \bullet \exists\, n : nat \bullet \forall\, w' \bullet (P \Rightarrow t' \leq t + n), \exists\, t \bullet \exists\, t' \bullet (P \wedge t' \geq t)\ ]\end{aligned}$$

The specification statement $w : [\ pre, post\ ]$ can be translated into Z using function $SSToZ$.

$$SSToZ(w : [\ pre, post\ ]) \mathrel{\widehat=} [\ \Xi\rho;\ \Delta w \mid pre \wedge post\ ]$$

ρ is all state variables not in the frame w. The user of $SSToZ$ may identify input components (using a $?$), or output (using a $!$) in the schema, instead of placing all state components in Δ or Ξ components. Miraculous specifications (i.e., terminating and establishing $false$) cannot be translated using this function.

We have not guaranteed that specific notations can be feasibly used together. There are still complications that must be resolved, e.g., syntactic ambiguity, how to write and define particular compositions, combining different viewpoints, etcetera. These are all issues that must be resolved on a specific-context level, i.e., when the notations to be combined and the compositions to be defined are known.

The translations between formalisms are all written as total functions. But some specifications expressible in one notation are inexpressible in a second language. To handle these language features while using translations, we have several options.

- Restrict the use of languages to only translatable elements. This can be described as an "intersection" approach to semantics, since only mutually expressible specifications are used in writing heterogeneous specifications.
- Extend languages (e.g., as is done in [10]) so as to be able to express all features of other languages. In contrast to the first approach, this might be called a "union" approach to semantics, since all expressible specifications can be used.
- Ignore the effect of differences in expressibility, and somehow account for the changes in interpretation when using notations in proof.

We used the first option in our case studies, and always checked for untranslatable elements in our specifications. Our case studies suggest that this lightweight approach to composition is feasible and useful in application. Alternative approaches are used, for example, in [28, 29].

A catalogue of untranslatable specifications is presented in [18]. This includes specifications like havoc, magic, and angelic nondeterminism.

2.2 Adding semiformal notations to a heterogeneous basis

The heterogeneous basis shown in Figure 1 contains semiformal notations, including those from Coad-Yourdon OOA/D [3] and SA/SD [4, 27]. To include a semiformal notation in the basis, we must *fix an interpretation* for it, and then express a general specification in this notation in one of the formalisms in the basis. This extends the heterogeneous basis to allow us to formally define the meaning of heterogeneous specifications that include parts written in (so-called) semiformalisms. Once formalization has occurred, the formal expression can thereafter be used to check for ambiguity at the semiformal level.

There are many interpretations we might take for a semiformalism. In particular, an interpretation will probably be useful only for a specific problem context or particular development context. Therefore, it is important that the approach to heterogeneous basis construction be extendible to new notations, interpretations, and formalizations. Our examples have convinced us that the basis is partwise extendible and changeable; that is, we can change formalizations without affecting the rest of the heterogeneous basis.

In [18], we provide a number of formalizations for semiformalisms, including:

– a formalization of data flow diagrams in Z, from [23], is used; a similar formalization of structure charts is also applied. This formalization does not permit expression of triggering or reactive behaviour. For such artifacts, formalizations from [15, 29] might be preferred.
– a formalization of Coad-Yourdon object oriented notations (e.g., object diagrams, assembly and classification diagrams, instance diagrams, etc.) is produced based on Hall's expression of objects in Z [7]. This formalization does not consider reactive behaviour of objects.
– a formalization of SADT diagram notations as sets and relations appears in [19].
– Jackson diagrams and pseudocode are also formalized, using predicates.

The formalizations we provide can easily be changed to fit new interpretations and domains. For example, the formalization of data flow diagrams from [15] could be used without much difficulty. Formalizations *should* be changed if they do not provide adequate interpretations for particular problems and contexts.

We now consider how heterogeneous notations are used in a meta-method for formal method integration.

3 A Meta-Method for Formal Method Integration

A meta-method for formal method integration (of system specification and design methods) was given in [18, 19]. The meta-method is based on the use of heterogeneous notations for defining compositions between notations, and thereafter on defining relationships between procedure steps. We briefly summarize the meta-method here, and then describe applications in the following sections.

The meta-method itself does not place constraints or restrictions on how the methods are to be used when integrated. This is the task of the method engineer, i.e., the user of the meta-method. The meta-method is designed to support the method engineer in placing constraints on using methods in combination. Whether particular methods are to be considered *complementary* is dependent on the context in which they are to be used.

The heterogeneous notation-based meta-method is summarized as follows.

1. *Fix a base method.* A base method gives a set of steps that is to be supported and complemented by other (invasive) methods. A base method may be formal or semiformal. It may support more of the software development cycle than other methods; it may also provide the steps that a developer may want to primarily use during development.
2. *Choose invasive methods.* Invasive methods will augment, be embedded, or be interleaved with the base method. The invasive methods complement the base method through notation, procedure, or user preference.
3. *Construct or extend a heterogeneous basis.* This is accomplished by constructing or adding notations from the base and invasive methods to a heterogeneous basis. A single formal notation from the heterogeneous basis (that is to be used to provide a formal semantics to system specifications that arise in the use of the integrated method) can be chosen and fixed at this point.

4. *Generalization and relation of method steps.* The method steps for the base and invasive methods are manipulated in order to define how they will work together in combination. Either one or both of the generalization and relation manipulations can be applied. In more detail, the manipulations are as follows.
   - *Generalization.* The steps of the base or invasive methods are generalized to use heterogeneous notations; effectively, new notations are added to a method, and the method steps are generalized to using the new notations.
   - *Relation.* Relation of method steps can follow generalization. Relationships between the (generalized) base steps and (generalized) invasive steps are defined. Steps of base and invasive methods may be ordered, mutually embedded, or interleaved to form a new set of steps. Examples of relationships are given in [19]; these include linking of sets of steps, replacing steps, extending and supplementing steps, and parallel use.

   The heterogeneous basis is used to support composition (and communication) of partial specifications. It also allows flexibility in how method steps are to be interrelated.
5. *Guidance to the user.* Hints, examples, and suggestions on how the integrated method can be used is provided.

The meta-method provides a systematic technique for combining formal methods with other formal and semiformal methods. It allows many different forms of (unintrusive and intrusive) integration, and supports the formal use of the formal methods in combination with semiformal methods. It does not provide a formal model of the steps and the methods being combined, e.g., as in [16]. It requires a fixed and formal semantics for all semiformalisms used in integrated methods, and it requires that the method engineer resolve syntactic ambiguities or incompatibilities across multiple notations and methods.

4 Case Study 1: Integrating Several Formal Methods

Our first small experiment with formal method integration involved combining several formal methods, selected from predicative programming, a Z "style", Morgan's refinement calculus, and Larch. The integrations were carried out at first in a pairwise fashion, by selecting specific pairs of methods and combining them using the meta-method; later, further methods were added to the results of these first integrations. Particular examples of integrations included: a combination of predicative programming, Morgan's refinement calculus, and the Z house style; a combination of the refinement calculus and predicative programming; and an integration of Z and Larch. We briefly summarize the process of integrating predicative programming and Z here as an example.

A Z house style and predicative programming can be considered as complementary: predicative programming is designed and has been shown to be useful for procedural or operational refinement, and is particularly convenient for developing recursive, concurrent, and real-time programs [11]. Z is useful for structuring large specifications, and has been shown to be a convenient notation in which to carry out data transformation [1]. In integrating the two methods, we chose the Z house style as the base method, and predicative programming as the invasive method. The heterogeneous basis shown in Figure 1 can be used to provide a formal semantics to compositions of specifications, by using predicative programming (we restricted use of Z to non-havoc specifications). The informal specification procedures of the Z style were generalized to also use predicative programming notations. Furthermore, the proof rules of Z were supplemented with predicative programming refinement rules.

We have applied this specific integrated method to a number of examples [18], ranging from very small (1-2 pages), to more substantial (10-15 pages). We found that predicative programming fits conveniently into the Z style of specification, especially with respect to a standard style of specification, where Z expressions are documented with informal text. We also found that the addition of predicative programming to the Z method makes procedural refinement much easier to do than with just pure Z. With a combination of both Z and predicative programming, we can build specifications to facilitate both data transformation and procedural refinement, and can build specification parts using the notation in which we want to carry out particular kinds of proof.

One complication with the approach was that syntactic differences in the notations had to be resolved in order to parse the compositions of syntaxes. For example, $\wedge$ and $\vee$ were overloaded operators, and so we used $\curlywedge$ and $\curlyvee$ to represent schema operators. Furthermore, we had to restrict the use of Z and predicative programming to only translatable specifications: therefore, we could not use the specification havoc in Z, or the specification magic in predicative programming.

In order to demonstrate the use of integrated formal methods, we briefly summarize a very small example. In this example, we use predicative programming and the refinement calculus to solve a problem. We choose to demonstrate this combination of methods because an example can be presented compactly. More interesting examples can be found in [18].

4.1 Example

The simple problem is as follows: we have an unordered, nonempty list of integers, L, and need to determine the minimum value s in the list and if a value x is in the list. A specification is as follows, using the parallel composition operator || of [11].

$$\mathit{findmin} \parallel \mathit{search},$$

$\mathit{findmin}$ is:

$$\mathit{findmin} \mathrel{\widehat{=}} s' = \mathit{MIN}\ j : 0,..\#L \bullet Lj,$$

and the search is a specification statement as follows.

$$\mathit{search} \mathrel{\widehat{=}} i : [\ (0 \leq i' < \#L \wedge Li' = x) \vee (i' = \#L \wedge x \notin L[0,..\#L])\ ].$$

Integer i will be set to $\#L$ if and only if x is not in $L[0,..\#L]$.

To verify that the specification is satisfiable, we must show that:

$$\exists\, s', i' \bullet (\mathit{findmin} \parallel \mathit{search}).$$

This simplifies as follows:

$$\begin{aligned}
&\exists\, s', i' \bullet (\mathit{findmin} \parallel \mathit{search})\\
=\ &\exists\, s', i' \bullet (s' = \mathit{MIN}\ j : 0, ..\#L \bullet Lj) \wedge\\
&\qquad\quad ((0 \le i' < \#L \wedge Li' = x) \vee (i' = \#L \wedge x \notin L[0, ..\#L]))\\
&\exists\, i' : 0, ..\#L \bullet Li' = x \vee \exists\, i' : \#L \bullet x \notin L[0;\ ..\#L]\\
=\ &x \in L[0, ..\#L] \vee x \notin L[0;\ ..\#L]\\
=\ &\top
\end{aligned}$$

We specify the minimum routine as a predicate, since we envision a tail-recursive implementation that is handled conveniently by predicate refinement. To demonstrate using two formal methods together, we specify the search and develop its implementation using the refinement calculus's invariant/variant approach.

Since predicate refinement (boolean implication) is monotonic over the combinator $\|$, we can refine the specification by parts. We first deal with the specification statement. It can be refined using $\sqsubseteq$ due to the result in [18] that says that monotonicity of wp refinement is preserved over all predicate combinators. Its development can be started as follows:

$$\begin{aligned}
\mathit{search} \sqsubseteq\ &i, L[\#L] := 0, x;\\
&i : [\ I \wedge i = 0, I' \wedge Li' = x\ ] \qquad (i)
\end{aligned}$$

where invariant $I \mathrel{\widehat{=}} 0 \le i < \#L + 1 \wedge \forall j : 0, ..i \bullet x \ne Lj$. The specification statement (i) is easily refined as follows, with variant $\#L - i$.

$$(i) \sqsubseteq \mathbf{do}\ Li \ne x \rightarrow i := i + 1\ \mathbf{od}$$

The refinement of the predicate is also straightforward (due to the monotonicity of $\Leftarrow$ over $\|$). It assumes that $+\infty$ is not an element of the list L.

$$\mathit{findmin} \Leftarrow i := 0.\ s := +\infty.\ s' = \mathit{min}(s, \mathit{MIN}\ j : i, ..\#L \bullet Lj) \qquad (1)$$

The predicate at the end of (1) is refinable to a selection.

$$\begin{aligned}
s' = \mathit{min}(s, \mathit{MIN}\ j : i, ..\#L \bullet Lj) \Leftarrow\ &\mathbf{if}\ i = \#L\ \mathbf{then\ ok\ else}\\
&\quad i \ne \#L \Rightarrow s' = \mathit{min}(s, \mathit{MIN}\ j : i, ..\#L \bullet Lj)
\end{aligned}$$

And the **else** branch can be refined in the obvious way:

$$\begin{aligned}
i \ne \#L \Rightarrow s' = \mathit{min}(s, \mathit{MIN}\ j : i, ..\#L \bullet Lj) \Leftarrow\ &\mathbf{if}\ Li < s\ \mathbf{then}\ s := Li\ \mathbf{else\ ok}.\\
&i := i + 1
\end{aligned}$$

The resulting implementation is heterogeneous (involving the programming language of [11] and Dijkstra's guarded command language) and can be transliterated to a homogeneous form in either programming language subset.

5 Case Study 2: SA/SD and Predicative Programming

We combined Structured Analysis and Structured Design [4] with predicative programming in the second case study. More precisely, we carried out three different integrations: one, where predicative programming *supplemented* SA/SD; a second, where SA/SD was *linked* with predicative programming; and a third, where an SA/SD development was *extracted* from the aforementioned supplementation of SA/SD by predicative programming. Full details of the case studies and inter-relations appear in [21].

In the initial case study, we chose SA/SD as the base method (because of its modelling procedures), and predicative programming as the invasive method. We selected predicates to use as a basis notation. SA/SD procedures were generalized to use predicates, and then the predicative refinement rules were used to supplement the SA/SD procedures for decomposition, process specification construction, and implementation. This integration is shown in Fig. 2; the ellipses in the figure represent method steps, while the boxes represent method products. The arrows represent *generate* and *use* relations.

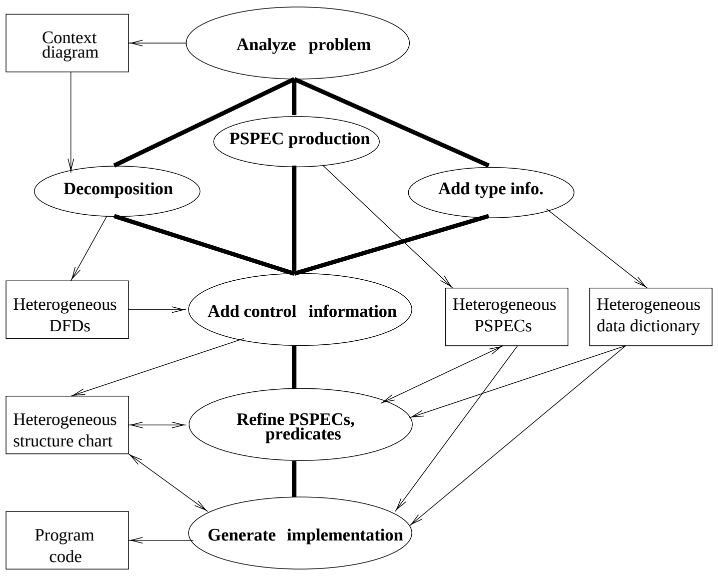

Fig.2. Combination of SA/SD and predicative programming

We applied this integrated method to a number of medium-sized examples, the largest being a construction of a simulator for a scheduler. The specification and (moderately detailed) development was approximately 20 pages long. The integration was particularly convenient to use: the predicative notation and procedures were *restrictable*, in that their use was confinable to the specification and development of a specific part of a system. In particular, we used predicative programming for specifying only those

PSPECs that seemed to have complex functionality at the requirement level; pseudocode and programming language specifications were used for the remaining PSPECs. This in turn leads us to suggest that it was possible to *gradually introduce* the predicative programming method into development, because of this restrictability feature.

The second integration of SA/SD and predicative programming mimicked the work of Semmens and Allen [25], and the SAZ project [22]. SA/SD was *linked* with predicative programming, by defining translations from SA/SD specifications to predicate specifications. The method was used by first applying standard SA/SD. Once a data flow diagram, PSPECs, and data dictionary had been produced, a pure predicative specification was constructed via the translations of Figure 1. Development could then continue, either by adding extra formal details and continuing with the predicative programming method, or by studying the produced specifications, analyzing what was missing, checking for consistency, and feeding back extra information into the standard SA/SD development.

The third integration was similar to what was described above. We commenced with a development using the SA/SD-predicative integration that is diagrammed in Fig. 2. After decomposition and PSPEC production, we *extracted* semiformal specifications from the formal specifications (e.g., following [23]), and continued the development using the semiformal specifications. This type of approach could be useful when changes in requirements occur, or when we find that during development the initial method to use in solving a problem is inappropriate.

A limitation with these approaches is that they required us to fix an interpretation and provide a formal semantics for semiformal SA/SD notations (e.g., DFDs, structure charts). This effectively fixes the domain of applicability of such notations. It must be possible to change such interpretations and formalizations, if other developers have other ideas on the meaning of the semiformal notations. Due to the partwise constructible nature of our heterogeneous basis, this seems possible. We might even envision the construction of heterogeneous bases for specific classes of applications and problems.

6 Case Study 3: Coad-Yourdon OOA/D and Z

The third application of the meta-method was to combine a Z house style with Coad-Yourdon object-oriented analysis and design [20]. The two methods are complementary: the Z house style offers formal specification techniques and proof rules; Coad-Yourdon offers procedures for the decomposition of requirements into objects and object structures.

In applying the meta-method, we selected Coad-Yourdon as the base method (since it supports a broader range of the software development process). The Z method was chosen as the invasive method. Coad-Yourdon procedures were generalized to use Z, and the Z proof procedures and specification style rules were used to supplement the Coad-Yourdon procedures. In particular, Z was used to specify particular objects of the system—objects that had complex functionality or data structures where a formal specification or development would prove to be justifiable. Z was also used to specify and develop implementations of object *methods* where the intended functionality of each method was complex. The integration is depicted in Fig. 3.

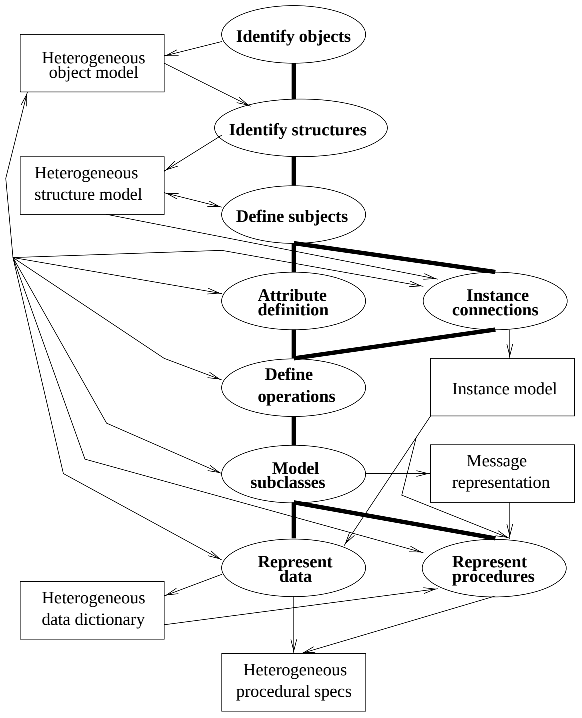

Fig.3. Integration of Coad/Yourdon OOA/D with Z method

We have applied this integrated method to a small collection of examples, the largest being the development of a text analyzer system with a graphical user interface. This required construction of X Windows code, text processing code, and statistics gathering code. Z was used in the formal development of the statistics gathering objects, and in providing a formal semantics to the specifications produced in the method. Coad-Yourdon was used in guiding the entire development. The entire specification was about 70 pages in length, resulting in approximately 4000 lines of C++ code.

We found that the combination of Z and Coad-Yourdon worked well, primarily because of the modular structure of specifications produced and required in both Z and Coad-Yourdon. That developers using this integrated method will be required to build specification parts with precise interfaces may help to make the process of using these specific multiple notations together much simpler. A further convenience using this approach is that the use of Z is again restrictable; we only used Z (and Z proof procedures) to develop those objects and methods that we felt necessary. Restrictability of formal methods may be important in allowing more wide-spread use of such techniques.

One particular difficulty in using Z and Coad-Yourdon arose when having to interface Z specifications with Coad-Yourdon specifications of object methods or attributes. At some point in a specification, it may be necessary to use semiformal attributes or methods in a formal specification (or vice versa). To avoid this problem, we made use of the heterogeneous basis. When using a semiformal method or attribute in a formal specification, we assumed that there existed a formal expression of the method or attribute in question. The interface to this method or attribute is acquired through the use of the formalization procedure in the heterogeneous basis; formalization of interfaces is inexpensive. Then, in the formal specification where we wanted to use the semiformal method or attribute, we simply used the formalization *interface*, even though we did not explicitly write down the entire formalization itself. This meant that we could not prove anything about the semiformal part, but we could use a formal *representation* of the part in a formal specification.

7 Conclusions

We have briefly described a general technique for integrating formal methods with other methods, all used for system specification and design. We have summarized the construction of a heterogeneous basis, and discussed some of the (syntactic and semantic) problems associated with using multiple notations together. We have detailed several case studies to which we have applied the integration techniques. While we have considered only a small number of studies, the evidence so far suggests to us that the lightweight approach to integration that we have taken is convenient and useful.

One issue we have not addressed so far is that of method compatibility. The meta-method itself provides a framework within which formal (and semiformal) methods can be put together and used. It does not directly address the issue of when methods can or cannot be combined. For particular methods, unresolvable incompatibilities in terms of model, syntax, or process may arise, and the meta-method may not be able to successfully integrate such methods. Future work will examine the problems of method compatibility, and will hopefully augment the meta-method to include such "soundness" constraints.

Further future work will encompass more and larger case studies, and will see us consider a wider spectrum of methods in integration. We will also look at constructing formal models of methods, in order to be able to speak precisely about the relationships we are defining between them. Finally, we will consider other approaches to giving semantics to heterogeneous specifications—particularly, union approaches, where the semantics of all specifications can be expressed in compositions.

Acknowledgements

Thanks to Rick Hehner, Pamela Zave, and the two anonymous referees for their suggestions and comments on this and other work.

References

1. R. Barden, S. Stepney, and D. Cooper. *Z in Practice*, Prentice-Hall, 1994.

2. J. Bowen and M. Hinchey. Ten Commandments of Formal Methods. Oxford University Computing Laboratory Technical Monograph, 1994.
3. P. Coad and E. Yourdon. *Object-oriented Analysis*, Prentice-Hall, 1990.
4. T. DeMarco. *Structured Analysis and System Specification,* Yourdon Press, 1979.
5. T. DeMarco. *Controlling Software Projects: Management, Measurement, and Estimation.* Yourdon Press, 1982.
6. J.V. Guttag and J.J. Horning. *Larch: Languages and Tools for Formal Specification,* Springer-Verlag, 1993.
7. A. Hall. Specifying and Interpreting Class Hierarchies in Z. In *Proc. Eighth Z User Meeting,* Cambridge, Springer-Verlag, 1994.
8. A. Hall. Using Formal Methods to Develop an ATC Information System. *IEEE Software,* March 1996.
9. J. Hammond. Producing Z Specifications from Object-Oriented Analysis. In *Proc. Eighth Z User Meeting,* Cambridge, Springer-Verlag, 1994.
10. E.C.R. Hehner and A.J. Malton. Termination Conventions and Comparative Semantics, *Acta Informatica*, 25 (1988).
11. E.C.R. Hehner. *A Practical Theory of Programming,* Springer-Verlag, 1993.
12. M.A. Jackson. *Software Requirements and Specifications,* Addison-Wesley, 1995.
13. S. King. Z and the refinement calculus. In *VDM '90: VDM and Z - Formal Methods in Software Development*, Third international symposium of VDM Europe, LNCS 428, Springer-Verlag, 1990.
14. K. Kronlöf, ed. *Method Integration: Concepts and Case Studies*, Wiley, 1993.
15. P. Larsen, J. van Katwijk, N. Plat, K. Pronk, and H. Toetenel. Towards an integrated combination of SA and VDM. In *Proc. Methods Integration Workshop*, Springer-Verlag, 1991.
16. Project MetaPHOR Group, MetaPHOR: Metamodeling, Principles, Hypertext, Objects and Repositories. Technical Report TR-7, University of Jyvaskyla, 1994.
17. C.C. Morgan. *Programming from Specifications,* Prentice-Hall, Second Edition, 1994.
18. R.F. Paige. *Formal Method Integration via Heterogeneous Notations,* PhD Dissertation, November 1997.
19. R.F. Paige. A Meta-Method for Formal Method Integration. In *Proc. Formal Methods Europe '97,* Lecture Notes in Computer Science, Springer-Verlag, 1997.
20. R.F. Paige. Using Heterogeneous Notations to Integrate a Formal and Object-Oriented Method. Submitted to *The Computer Journal,* 1997.
21. R.F. Paige. Integrating Predicative Programming and SA/SD using Heterogeneous Notations. Submitted to *PROCOMET '98.*
22. F. Polack, M. Whiston, and K.C. Mander. The SAZ Project: Integrating SSADM and Z. In *Proc. FME '93: Industrial-strength Formal Methods*, LNCS 670, Springer-Verlag, 1993.
23. G. Randell. Data flow diagrams and Z. In *Z Users Meeting '90*, Springer-Verlag, 1991.
24. K. Schoman and D. Ross. Structured Analysis for requirements definition, *IEEE Trans. on Software Engineering*, 3(1), 1977.
25. L.T. Semmens, R.B. France, and T.W. Docker. Integrated Structured Analysis and Formal Specification Techniques, *The Computer Journal* 35(6), June 1992.
26. J.M. Spivey. *The Z Notation: A Reference Manual*, Prentice-Hall, 1989.
27. E. Yourdon and L. Constantine. *Structured Design,* Prentice-Hall, 1979.
28. P. Zave and M. Jackson. Conjunction as Composition, *ACM Trans. on Software Engineering and Methodology*, 2(4), October 1993.
29. P. Zave and M. Jackson. Where do operations come from? An approach to multi-paradigm specification, *IEEE Trans. on Software Engineering,* 12(7), July 1996.