Formal Aspects of Computing (1998) 3: 1–000
© 1998 BCS

Heterogeneous Notations for Pure Formal Method Integration

Richard F. Paige[1]

[1]Department of Computer Science, University of Toronto, Toronto, Ontario M5S 3G4, Canada. paige@cs.utoronto.ca

Keywords: method integration; heterogeneous specification; refinement

Abstract. We outline an extendible approach for combining formal methods—such as Z, Morgan's refinement calculus, and predicative programming—based on composing specifications written in similar formal languages. We discuss how algorithm refinement can be extended to such a setting, and outline some examples of using integrated formal methods. We also provide justifications for why using combinations of similar methods might be helpful.

1. Introduction

Formality in software development can be applied in many ways, such as for writing specifications [Spi89], or for critical systems development [BoH94]. A kernel of recent research has focused on the synthesis of method and notation concepts [Kro93, SFD92]. It is on this general theme that we focus in this paper.

The perspective we take is that there is no one formal method that will suffice for meeting all functional and non-functional requirements. There are both technical and philosophical reasons for this. For one, each separate formal method offers a different set of expressive capabilities appropriate for specifying clearly and concisely a different set of properties [ZaJ96]. Furthermore, the complexity of the development process suggests that it is unrealistic to expect to have one method that perfectly conveys this process [Kro93].

Our interest in methods and notations has focused on combining two or more methods into a potentially more useful whole, viz., *method integration* [Kro93].

Correspondence and offprint requests to: Richard F. Paige, Department of Computer Science, York University, Toronto, Ontario M3J 1P3, Canada. paige@cs.yorku.ca

In this paper, we outline a technique for integrating only formal methods, the process for which we term *pure formal method integration.*

1.1. Methods and method integration

A *method* for software development provides notations, as well as a process that fashions a systematic way of accomplishing parts of the task of software development. A formal method includes a formal specification language as well as transformation rules, e.g., for algorithm and data refinement.

Method integration is the activity of resolving incompatibilities between methods, so that they can be effectively used together [Kro93]. Method integration has been used in practice, in combining multiple formal methods [ZaJ96, Hal96, BDW96], and for combining formal with informal methods [SFD92, Kro93].

This paper presents a technique for pure formal method integration (PFMI), the process of combining multiple formal methods. PFMI can be carried out when formal methods are complementary in some way. For example, PFMI might be performed so as to be able to say more than is possible with one notation. Or, it might be done to deal with complexity: by applying methods to those problem aspects for which they are best suited, it may be possible to better deal with large problems than with single methods [ZaJ96, Jac95].

We now briefly introduce the technique that we use for pure formal method integration, and discuss complementarity of methods in more detail.

1.2. Heterogeneous specifications and complementarity

A heterogeneous notation is syntactically and semantically composed from several separate notations. It 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 separate notations.

The semantics of a heterogeneous specification is given by formally defining the meaning of all the used notation compositions. Such a semantics can be provided by a heterogeneous basis.

Definition 2. A *heterogeneous basis* is a set of translations between notations that can be used to provide a formal semantics to heterogeneous specifications.

The heterogeneous basis that we use is shown in Fig. 1 (arrows are translations).

$$\begin{tikzcd}
 & w : [\,pre, post\,] \arrow[dl, leftrightarrow] \arrow[dr, leftrightarrow] \arrow[dd, leftrightarrow] & \\
wp(S, R) \arrow[dr, leftrightarrow] & & \mathbf{Z} \\
 & \textbf{predicative notation} &
\end{tikzcd}$$

Fig. 1. A small heterogeneous basis

The notations in Fig. 1 are all quite similar to begin with. Each makes use of first-order logic or set theory for types, and each has a similar model of state transition. This suggests that we should expect these specific notations to combine in a straightforward manner, and raises questions as to whether it is useful to combine them. We are interested in combining notations in order to combine *methods*; a notation is only part of a method. We suggest that similarity in terms of notation need not exclude complementarity in terms of method.

The methods associated with the notations Fig. 1 can be considered as complementary in terms of *notation*. Each notation in Fig. 1 supports a different style of specification: specification statements use a pre- and postcondition specification style, while Z and predicative notation use a single predicate. Further, predicative notation can explicitly talk about timing (through time variables), while the other notations express only termination. It is possible to extend Z or specification statements to talk about time—e.g., see [Fid94]—but the ability to write heterogeneous specifications can eliminate the need to do so.

The methods in Fig. 1 are also complementary in terms of *refinement technique*. For example, Morgan's refinement calculus supports development of looping programs using loop invariants. A different approach is used in predicative programming, where instead recursive programs are developed using recursive refinement [Heh93]; loop invariants are not explicitly used. Morgan's calculus and predicative programming are also complementary to Z, because the former are program design calculi, while the latter is a specification language.

The complementarity of the methods in Fig. 1 suggests that it may well be useful to consider their integration, in order to take advantage of their individual useful techniques.

1.3. Heterogeneous notations for PFMI

Pure formal method integration can be carried out in two steps. The first step is to combine the notations used by the methods of interest, so that we can write heterogeneous specifications with a formal semantics. This occurs by building a heterogeneous basis. Once this is done, PFMI continues by *generalizing* the transformation rules or proof techniques associated with the methods of interest to be applicable to heterogeneous notations. This occurs by defining how transformation rules or proof techniques from one method are to be applied to specifications from a second method. In effect, this lets developers use rules and techniques from multiple methods together in developing programs or in proving properties about heterogeneous specifications. As we shall show, this can give us a low-cost form of pure formal method integration.

Heterogeneous notations will not solve all the problems of method integration; issues such as method compatibility and tool use still remain to be dealt with. We claim that heterogeneous notations can provide a systematic first step for PFMI, and we attempt to provide some evidence to support this claim herein.

1.4. Organization of the paper

The remainder of this paper is organized as follows. In Section 2, we briefly describe how to construct the heterogeneous basis of Fig. 1. In Section 3 we summarize the process of extending algorithm refinement to heterogeneous specifications. This serves to partially integrate the methods which use the respective notations. In Section 4, we informally outline the details of several examples of using integrated formal methods that can be found in the references. Finally, in Section 5, we summarize our findings, and discuss in brief detail the issues regarding the effect of PFMI on software development. The reader is directed to [Pai99] for full technical details and further examples.

2. A Heterogeneous Basis

A heterogeneous basis is used to provide a formal semantics to a heterogeneous specification by defining language compositions. Its existence is a precondition for PFMI by heterogeneous notations: without a basis, we have no way to prove properties regarding or apply transformations to heterogeneous specifications.

In this paper, we construct a heterogeneous basis by giving translations between notations. We also take a specific approach to resolving differences in notation expressiveness. By doing so, we present a technique that describes ways to translate a heterogeneous specification into a formal specification that defines the meaning of the initial heterogeneous specification.

We depicted the translations that we use in this paper in Fig. 1. We summarize only a few of these translations here, due to space constraints; the remainder can be found in [Pai99]. Other translations can be constructed by composition. In presenting the translations, we assume that all notations use the unprimed-primed convention of Z [Spi89] to distinguish pre- and poststate. Different notations, such as CSP and Larch, are considered in [Pa97a, Pa97b].

2.1. Predicative programming and specification statements

Predicative specifications [Heh93] can come in two forms: the first makes no mention of timing or termination; the second explicitly talks about time through use of the time variables t and t'. Such variables can provide constraints on the execution time of a program that implements the specification.

Let **frame** $w \cdot P$ be a predicative specification (where P is a predicate on pre- and poststate, and w is a frame) not referencing the time variables t and t'. The specification is translated to a specification statement by the function *PredToSS*, defined as:

$$\mathit{PredToSS}(\textbf{frame}\ w \cdot P) \mathrel{\widehat=} w : [\, \mathit{true}, P \,]$$

Complications in using this translation are discussed in [Pai99].

A predicative specification **frame** $w \cdot P$ that includes references to time variables t and t' in P can be translated using *TimedPredToSS*.

$$\begin{array}{l}\mathit{TimedPredToSS}(\textbf{frame}\ w \cdot P) \mathrel{\widehat=} \\ \quad w : [\, \forall\, t \cdot \exists\, n : \mathit{nat} \cdot \forall\, w' \cdot P \Rightarrow t' \le t + n, \exists\, t \cdot \exists\, t' \cdot P \wedge t' \ge t \,]\end{array}$$

The precondition in the translation result is the exact precondition for termination due to Norvell [Heh93], while the postcondition is existentially quantified to locally bind time variable references.

To translate a specification statement into a predicative specification that does not mention time, we use *SSToPred*.

$$\mathit{SSToPred}(w : [\, \mathit{pre}, \mathit{post} \,]) \mathrel{\widehat=} \textbf{frame}\ w \cdot (\mathit{pre} \Rightarrow \mathit{post}).$$

Translations from specification statements to timed predicative specifications are in [Pai99], as are translations *PredToWp* and *WpToPred*, between predicative notation and *wp*.

2.2. Specification statements and weakest preconditions

The mapping from specification statements to weakest preconditions, *SSToWp*, can be found in [Mor94], so we omit it here. The reverse transformation, given as function *WpToSS*, is somewhat more complex and is derived from [HeM88].

$$\mathit{WpToSS}(\mathit{wp}(S, R')) \mathrel{\widehat=} w : [\, \mathit{wp}(S, \mathit{true}), (\neg \mathit{wp}(S, w' \neq w_0))[w'/w_0] \,],$$

where w is a frame of variables (determined in the usual way by the specifier) with w_0 not in w. It is assumed that $\mathit{wp}(S, R')$ is given for an arbitrary relation R'. For the calculation of *wp* in the postcondition of *WpToSS*, the w_0 are the variables.

2.3. Z and specification statements

The final translations we describe are between Z and specification statements. The translation from Z to specification statements has been noted in the past [Kin90], and so we omit it here. The reverse mapping, from specification statements to Z, requires the translator to decorate variables (as input or output) in the resulting schema. The specification statement $w : [\ \mathit{pre}, \mathit{post}\ ]$ can be translated using function *SSToZ*.

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

(The Ξ-list denotes those variables that remain unchanged by the specification.) In *SSToZ*, ρ is the list of all variables not in the frame w.

2.4. Expressiveness and semantics

The notations in Fig. 1 are not equivalent in terms of what they can express; for example, specification statements can express angelicism, while predicative notation cannot [HeM88]. Differences in expressiveness must be dealt with when composing partial specifications in different languages. There are many approaches we might take. For example, we could extend notations to represent special features [HeM88], or, we could restrict the use of language features to translatable elements. In the latter approach, the intersection of language semantics is taken in the heterogeneous notation. We have used this approach in this paper. The advantages with this approach is that the translations between notations are simple to express and use, and transformation techniques can be extended straightforwardly to heterogeneous specifications. A disadvantage with this approach is that we cannot express more with a combination of notations than we can with one notation (and in most cases, can express strictly less).

Though we have taken an intersection approach to semantics in this paper, the general approach to PFMI that we present permits use of other approaches to giving a semantics to heterogeneous specifications (examples are discussed in [Pa97a]). We use only intersection techniques in the work summarized herein.

3. Generalizing Algorithm Refinement

The heterogeneous basis of Section 2 provides translations that are used to supply heterogeneous specifications with a formal semantics. Such a semantics can be used to define how the transformation rules from individual methods—e.g., for algorithm refinement or data transformation—can be used on heterogeneous specifications. By extending transformation rules to heterogeneous notations, an explanation of how individual methods are to be used together is provided.

As a first step, we briefly outline how algorithm refinement rules can be generalized to heterogeneous specifications. We present only a selection of generalized rules here; further rules and detailed examples can be found in [Pa97a, Pai99]. Rules for data transformation and satisfiability can be found in [Pa97a].

3.1. Refinement relation extension

We give a few rules that demonstrate how to apply the refinement relations $\Leftarrow$ (of predicative programming) and $\sqsubseteq$ (of Morgan's refinement calculus) to specifications of different type. In following subsections, we summarize rules that apply over specification combinators. We omit all proofs, which can be found in [Pa97a, Pai99]

Rule 3.1. Let P and Q be predicative specifications on prestate σ and poststate σ'. Then $P \sqsubseteq Q \Leftarrow \forall \sigma, \sigma' \cdot (P \Leftarrow Q)$.

Informally, this rule states that if P is refined by Q using $\Leftarrow$, then P will also be refined by Q using $\sqsubseteq$.

A further result tells us that a specification statement is always refined by its predicative translation.

Rule 3.2. If $S \mathrel{\widehat{=}} w : [\, pre, post \,]$ is a specification statement and $pred_S$ is its predicative translation, then $S \sqsubseteq pred_S$.

Finally, we see that refinement is actually preserved over translation from specification statements to predicative notation.

Rule 3.3. For specification statements S and T, and their predicative translations $pred_S$ and $pred_T$, $(S \sqsubseteq T) \Rightarrow (pred_S \Leftarrow pred_T)$.

3.2. Refinement over conjunction and disjunction

We describe a pair of refinement rules for application over predicative conjunction and disjunction; more rules are in [Pai99]. In the following, let S, S' and T be specification statements, and let P be a predicative specification; the combinators $\wedge$ and $\vee$ are predicative operators from [Heh93].

Rule 3.4. If $S \sqsubseteq T$ then $P \wedge S \sqsubseteq P \wedge T$.

The predicative combinators $\vee$ and $\wedge$ can be used to compose specification statements. Such compositions can also be refined in a partwise manner.

Rule 3.5. Providing that S, S', and T are all expressible in predicative notation,

$$\begin{aligned}(S \wedge T \sqsubseteq S' \wedge T) &\quad\Leftarrow\quad S \sqsubseteq S',\\ (S \vee T \sqsubseteq S' \vee T) &\quad\Leftarrow\quad S \sqsubseteq S'.\end{aligned}$$

3.3. Changing notation within a refinement tree

The final collection of algorithm refinement rules that we present can be used for changing notation as a refinement step. These rules let us use translations within a calculational proof style. Changing notation during a proof might be useful for several reasons: e.g., to reformulate a specification in a new notation which is more convenient for the developer; or, to explore different strategies that are not possible in the original notations.

We present two rules for changing notation, omitting all proofs. Further rules and several proofs are in [Pai99]. In the following, let P be a predicative specification, let $X \mathrel{\widehat=} [\Delta S;\ i? : I;\ o! : O \mid \mathit{pred}]$ be a Z schema, and $T \mathrel{\widehat=} w : [\,\mathit{pre}, \mathit{post}\,]$.

Rule 3.6. If $\mathit{pre} \land \forall\, \sigma' \cdot (\mathit{post} \Rightarrow P)$ then **frame** $w \cdot P \Leftarrow T$ and **frame** $w \cdot P \sqsubseteq T$.

Rule 3.7. If $(\exists\, w' \cdot \mathit{pred}) \Rightarrow (\mathit{pre} \land \forall\, w' \cdot (\mathit{pred} \Leftarrow \mathit{post}))$, then $X \sqsubseteq T$ and $X \Leftarrow T$.

4. Use of Integrated Formal Methods

In this section, we informally summarize the details of some examples of using integrated formal methods. Space constraints prevent us from presenting examples of using integrated formal methods here. However, we have some experience in using and combining the methods discussed in this paper, as well as for other methods as well. Some of the examples of pure formal method integrations we have carried out and used are as follows.

1. **A combination of Z and predicative programming.** We used Z to write specification parts on which we want to carry out data transformation; Z was used to specify the system state, and also those operations on which we wanted to carry out data transformation. We used Z in this role because it offered the ability to carry out *piecewise* data transformation [Mor90], whereas such techniques are not always possible to use in predicative programming. Furthermore, Z provided the ability to build specifications by parts, a technique we found helpful in dealing with error conditions.
   We used predicative notation to specify the other system parts, those on which we wanted to carry out algorithm refinement. It has been suggested that a refinement calculus is easier to use for algorithm refinement than Z [Kin90]. Predicative programming, in particular, let us use *recursive refinement* rules [Heh93] to develop implementations from specifications.
   A further example of using Z and predicative programming, for reasoning about time and space, is presented in [Pai98].
2. **A combination of Z and the refinement calculus**. Z was used to specify system state and those operations to which we wanted to apply data transformation. The refinement calculus was used to specify and algorithmically refine the other operations. Unlike the approach of [Kin90], we did not transform from Z to the refinement calculus. Instead, each notation was used where it was deemed to be helpful, and transformation rules from each method were used on the notations of the method.
3. **A combination of Larch and predicative programming.** Larch LSL was used to specify a system module, i.e., an abstract data type. Predicative

notation was used to specify system operations that used the LSL specification. We used the two together because the methods are complementary: Larch LSL has tool support, while predicative programming does not. Predicative programming has algorithm refinement rules, while Larch LSL does not. Specifically, we used predicative programming for algorithm refinement, while Larch LP was used to semi-automatically prove properties about the abstract data type.

These, and other examples of pure formal method integration are presented in detail in [Pa97a, Pai99].

5. Discussion

Integrating formal methods via heterogeneous notations (or by any other approach) introduces a number of significant issues, especially associated with learning and using formal methods, tool application, and standardization. We briefly address some of these issues here.

5.1. Learning and using formal methods

A complaint about formal methods is that they are difficult to learn and apply, because of mathematical syntax and because of the difficulty in understanding the underlying modeling concepts. One implication of integrating formal methods is that this will make formal methods even more difficult to learn, use, and teach, because integrated methods will require understanding different modeling techniques and multiple syntaxes.

This is a legitimate concern. By combining several different syntaxes, it is possible to introduce ambiguities, incompatibilities, and potentially difficult-to-understand specifications. On the other hand, the use of several different notations—and hence several different methods—could in turn make formal methods easier to use [BoH94]. A project by Zave and Mataga [ZaM93] showed that using multiple formal methods could produce smaller, more concise, and more understandable specifications than by using a single notation. The case studies in [Pai98] showed that multiple formal methods could produce shorter refinements for reasoning about time. The ability to write concise, understandable formal specifications could prove helpful in making formal methods easier to learn and use. Our specific approach to heterogeneity could be beneficial in this respect. A low-cost technique for giving a semantics to heterogeneous specifications could make using multiple methods more attractive and accessible to developers, than for other approaches that require use of more complicated semantic models.

More experience with practical examples of using multiple methods is necessary before it becomes clear as to the effect of using integrated methods together on teaching and learning formal methods.

5.2. Standardization

Formal methods often use different notations for the same or similar concept, e.g., types, type constructions, specification combinators, program combinators, et cetera. One effect of combining formal methods via heterogeneous notations is that it may lead to eventual standardization of common parts of notations at both a syntactic and semantic level, e.g., representation of system state, types and operators, specification combinators, et cetera. Such standardization must be done for reasons of parsing, as well as to improve specification readability. This may in turn lead to more understandable and usable formal methods, since less new mathematical notation will have to be understood when learning a method.

5.3. Use of tools

Certain formal methods are supported by tools, e.g., for theorem proving, code generation, model checking, et cetera. Integrating formal methods will have a significant impact in terms of tool use. Certainly, if it is determined that the impact of PFMI on tool use is negative (in the sense that tools become difficult to use) then using combined methods may not be appropriate for projects that rely heavily on tools. However, if it can be shown that existing or new tools can be used to support integrated methods, then the benefits of formal method integration and use of individual tools may be further propagated.

The issue of how to extend tools to support multiple formal methods is one of ongoing research. In supporting integrated methods via heterogeneous notations, there are several strategies that might be considered. One strategy might be as follows. If a tool to support a single formal method is supplied with a heterogeneous specification—for which the tool only supports notations for parts of the specification—the tool might treat parts that it does not understand as "black boxes", which it must assume are properly typed system components, but which it cannot manipulate. So, for example, if a theorem prover was supplied with a heterogeneous specification, it might be capable of proving lemmas about parts of the specification written in a notation that it understands, but for other parts, it will have no such capability.

Another approach to tool support might make use of application frameworks, which are the topic of a case study in [BDW96]. A framework might be used to control communication and interaction between several separate tools for support of formal methods. The framework could (partially) automate the process inherent in a heterogeneous basis, by translating specifications and by supplying information to the separate tools when it is needed. Part of our current work is focusing on this approach.

6. Conclusion

Individual formal methods have been shown to be useful, on both small problems and industrial-scale projects. However, one formal method is not a panacea. Each formal method supports particular tasks well, e.g., the writing of concise specifications, algorithm refinement, data transformation, or application of tool support. For complicated projects, or for projects where functional or non-functional requirements dictate that no single method will suffice, the use of method integration techniques will be necessary.

This paper has suggested an approach for combining instances of formal methods that, on a superficial level appear to be very similar. On closer inspection, we have found that the methods are in fact complementary in many different ways, e.g., in terms of notation or transformation technique. We explored how to compose specifications from such methods, and to give the compositions a formal meaning. We also looked at how algorithm refinement can be used in such a setting. Our future work will look at broadening the scope of the approach to further methods (see [Pa97a, Pa97b] for initial work in this direction), and to providing tool support to multiple methods via tool integration. Such extensions, and method integration in general may help to further propagate the use of formal methods into practice.

Acknowledgements

This work was partially carried out while the author was at the University of Toronto. Rick Hehner and Pamela Zave provided advice and many helpful comments on the paper and the research in general. Special thanks to the anonymous referees for their excellent suggestions.

References

[BDW96] J. Biccaregui, J. Dick, and E. Woods.: Quantitative Analysis of an Application of Formal Methods. In *Proc. Formal Methods Europe '96*, Springer-Verlag, 1996.

[BoH94] J. Bowen and M. Hinchey.: Ten Commandments of Formal Methods. Oxford University Computing Laboratory Technical Monograph, 1994.

[Fid94] C. Fidge.: Adding real time to formal program development. In *Proc. Formal Methods Europe '94*, LNCS 873, Springer-Verlag, 1994.

[Hal96] A. Hall. Using Formal Methods to Develop an ATC Information System. *IEEE Software,* March 1996.

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

[HeM88] E.C.R. Hehner and A.J. Malton.: Termination Conventions and Comparative Semantics. *Acta Informatica*, 25 (1988).

[Jac95] M.A. Jackson.: *Software Requirements and Specifications*, Addison-Wesley, 1995.

[Kin90] S. King.: Z and the refinement calculus. In *Proc. VDM '90*, LNCS 428, Springer-Verlag, 1990.

[Kro93] K. Kronlöf, ed.: *Method Integration: Concepts and Case Studies*, Wiley, 1993.

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

[Mor90] J. Morris.: Piecewise data refinement. In *Formal Development of Programs and Proofs,* Addison-Wesley, 1990.

[Pa97a] R.F. Paige.: Formal Method Integration via Heterogeneous Notations, PhD dissertation, University of Toronto, November 1997.

[Pa97b] R.F. Paige.: A Meta-Method for Formal Method Integration. In *Proc. Formal Methods Europe '97*, LNCS 1313, Springer-Verlag, September 1997.

[Pai98] R.F. Paige.: Comparing Extended Z with a Heterogeneous Notation for Reasoning about Time and Space. In *Proc. ZUM '98*, LNCS 1493, Springer-Verlag, September 1998.

[Pai99] R.F. Paige.: Heterogeneous Notations for Pure Formal Method Integration. In *Formal Aspects of Computing* vol. 9(E), 1999.

[SFD92] L.T. Semmens, R.B. France, and T.W. Docker.: Integrated Structured Analysis and Formal Specification Techniques. *The Computer Journal* 35(6), June 1992.

[Spi89] J.M. Spivey.: *The Z Notation: A Reference Manual*, Prentice-Hall, 1989.

[War93] N. Ward.: Adding specification constructors to the refinement calculus. In *Proc. Formal Methods Europe '93*, LNCS 670, Springer-Verlag, 1993.

[ZaJ96] P. Zave and M. Jackson.: Where do operations come from? An approach to multiparadigm specification. *IEEE Trans. Software Engineering*, 12(7), July 1996.

[ZaM93] P. Zave and P. Mataga.: A formal specification of some important 5ESS features, Part I: Overview. AT&T Bell Laboratories Technical Memorandum, October 1993.