A Meta-Method for Formal Method Integration

Richard F. Paige

Department of Computer Science, University of Toronto, Toronto, Ontario, M5S 3G4, Canada. paige@cs.toronto.edu*

Abstract. We describe a meta-method for formal method integration [Pai97]. The approach is applied to combining formal methods with other formal and semiformal methods. We discuss the theory behind formal method integration, present two example combinations, and use an integrated method in solving a small problem.

1 Introduction

Method integration involves defining relationships between different methods so that they may be productively used together to solve problems. In a software engineering context, method integration has seen recent research on combining specific methods [LKP91, SFD92], and on the formulation of systematic techniques [Kro93, Pai97]. In this paper, we follow the latter theme and describe a meta-method for formal method integration based on heterogeneous notations [Pai97].

We commence with a brief overview of method integration and the general means we take to accomplishing it. Our approach is based on heterogeneous notations, combinations of existing formal and semiformal notations. After providing the background for heterogeneous notations, and discussing their role in method integration, we describe a meta-method for method integrations involving at least one formal method, and then briefly apply the technique to examples.

Due to space restrictions, this paper only provides high-level details concerning our approach to formal method integration and heterogeneous notations. The interested reader may find further results in [Pai97].

1.1 Method integration

When integrating methods, incompatibilities between techniques are resolved so that the approaches can be safely and effectively used together [Kro93]. Method integration in a software engineering context is a problem of growing research interest. A significant reason for this is that it is unlikely that one method will suffice for use in the development of increasingly complex systems [Jac95,

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

DeM82]; method integration provides systematic techniques for dealing with this complexity. Furthermore, method integration has been used and has proved to be useful in practice in various forms, e.g., at Rolls-Royce [Hil91], BT [SFD92], Westinghouse [Ham94], Praxis [Hal96], and elsewhere.

1.2 Heterogeneous notations and specifications

A notation is an important part of any method; it is used to describe the concrete products of the technique. Notations play a key role in how we integrate methods. In particular, we combine notations as a first step towards combining formal methods. A *heterogeneous notation* is a combination of notations. A heterogeneous notation 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 constrain what is to be allowed as a composition. Useful compositions will depend on the context and the notations to be used. Compositions may occur through use of specification combinators, by use of shared state or shared names, or in other ways. We supply some examples later.

Heterogeneous notations are useful for a number of reasons: for producing simpler specification languages [ZaJ93]; for writing simpler specifications than might be produced using a single language [ZaM93]; for ease of expression [BoH94]; and because they have been proven to be successful in practice [ZaJ95, SFD92, Hal96].

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 formalisms, and formalizations, that provides a formal semantics to compositions of specifications written in two or more notations.

The heterogeneous basis that is used in this paper is partially presented in [Pai97]. It is created by translation. We discuss it in the next subsection, and in Section 2 outline the process of its construction.

1.3 A heterogeneous basis

A heterogeneous basis supplies a formal semantics to a heterogeneous specification [Pai97]. It is used to provide the foundation on which integrated formal methods are defined. The basis in this paper consists of a set of languages with translations defined between them. It is depicted in Fig. 1.

The predicate notation is from [Heh93]; Z is from [Spi89]; specification statements (i.e., $w : [pre, post]$) are from [Mor94]; CSP is from [Hoa85]; and the two Larch languages are from [GuH93]. The remaining semiformal notations are from SA/SD [DeM79, YoC79], SADT [MaM88], and Coad-Yourdon object oriented

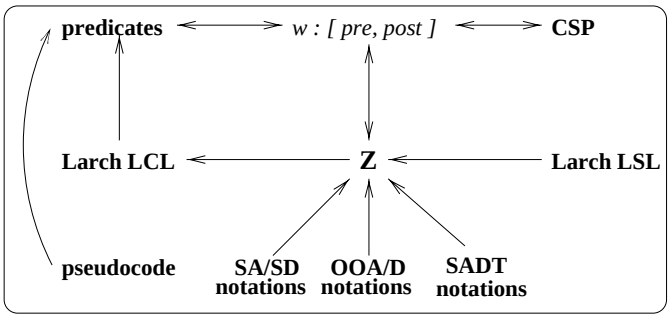

Fig. 1. A heterogeneous basis

analysis and design [CoY90]. More notations are considered in [Pai97]. Notations were chosen to be placed in the basis for a variety of reasons: because they are well-known, or because they have proven to be useful in practice, or because there are existing method integrations involving such notations that can be used for comparison.

In Fig. 1, the arrows represent translations between notations. Many of the translations in Fig. 1 are described in detail in [Pai97]. We will give a few examples in Section 2, considering both formal and semiformal notations, and will demonstrate the general technique that can be used for constructing (or extending) a heterogeneous basis that consists of formal and semiformal notations.

A specification written using the notations from Fig. 1 is given a semantics in terms of a formal specification written using only one of the formal notations of the basis. The user of the basis chooses the formal notation to use for their particular application and context, and defines the meaning of a heterogeneous specification in this formal notation. Specific context-level problems must be dealt with by the specifier and user of the heterogeneous basis, e.g., how to resolve parsing problems, and how to deal with differences in expressive capabilities (we address some of these issues in Section 2).

The existence of the heterogeneous basis means that we can give a formal semantics to compositions of partial specifications written in the notations of the basis. It does not provide us with any results regarding the feasibility (or even the possibility) of using specific notations in composition. Such compatibility issues must be examined in a case-by-case setting.

1.4 Integrating methods with heterogeneous notations

Heterogeneous specifications are written using two or more notations. Formal method integration is carried out by first precisely combining the notations used by the methods. This first step occurs by constructing or extending a heterogeneous basis consisting of the notations of interest, and by resolving syntactic differences among the notations. Once this is done, the method integration process continues by *generalizing* method steps to use heterogeneous notations (i.e.,

by adding notations from one method to another method), and by interleaving (perhaps generalized) method steps from the methods that are to be combined.

Heterogeneous notations will not solve all the problems of method integration; issues with respect to method compatibility and models of procedure remain to be dealt with. We claim that heterogeneous notations provide a systematic and lightweight basis for formal method integration, and we provide evidence to support this claim herein.

1.5 Overview

We commence the paper by describing a general process for constructing the heterogeneous basis of Fig. 1. We suggest a meta-method for formal method integration and use it to combine methods: we integrate two refinement-based methods in one example, and combine a refinement-based and structured method in a second example. We then use the first integration in solving a small problem (detailed examples using combined formal and semiformal methods are given in [Pai97]).

2 A Heterogeneous Basis

A heterogeneous basis is shown in Fig. 1. The 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. In this section, we summarize the process of constructing the heterogeneous basis of Fig. 1. In particular, we consider the addition of formal notations (by translation), and the process of adding semiformal notations to an existing basis (by formalization).

2.1 Formal translations

A formal notation may be added to a heterogeneous basis by providing a translation from the formal notation into a second formal notation already in the heterogeneous basis. In doing so, the extender of the basis must analyze the expressive capabilities of the new notation, viz., what can and cannot be translated into and from the new notation. The expressive capabilities of the notations will affect the use of the translations, and will also affect how a semantics is given to a heterogeneous specification that uses the new notation.

We present several example translations here, building on those that have been previously given in the literature [Kin90, HeM88, Mor94]. We also identify several examples of untranslatable specifications. In a meta-method, differences in notation expressiveness should be handled in a way that is most appropriate to the users of the meta-method and heterogeneous basis; this might be carried out by restricting translation domains, or by extending languages. In this paper, we restrict translation domains, and therefore take an 'intersection' approach to semantics (i.e., only mutually expressible concepts in combinations of languages are used). Alternative approaches—e.g., 'union' approaches to semantics—are

considered or discussed in [ZaJ93, Pai97]. As we shall see, the meta-method for formal method integration does not require that an intersection (or union) approach to semantics be used. However, the examples in this paper only make use of an intersection approach.

To simplify the process of integration, we assume that all languages use the primed/unprimed notation of Z to distinguish poststate from prestate. We also assume that types and type constructors can be freely translated. We retain the convention of [Pai97] and describe each translation as a function from language to language.

A predicate specification **frame** $w \bullet P$ that does not refer to the time variables t, t' [Heh93] can be translated to a specification statement [Mor94] using the mapping $PredToSS$.

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

(Translations are given in [Pai97] for handling time variables.)

A specification statement can be translated to a predicate as follows.

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

[Pai97] describes how to translate a specification statement to a predicate that includes references to the time variables t and t'. The predicative notation cannot represent angelic specifications [BaV89] and terminating but otherwise arbitrary behaviour (i.e., havoc [Mor94]), and so $SSToPred$ cannot translate these specifications and maintain their interpretation.

The Z schema $Op \mathrel{\widehat{=}} [\Delta S;\ i? : I;\ o! : O \mid pred]$ can be mapped into a specification statement using the function $ZToSS$. This result is due to [Kin90].

$$ZToSS(Op) \mathrel{\widehat{=}} w : [\ (\exists\ w' : T \mid inv \bullet pred), pred\ ]$$

(The Δ-schema denotes those state elements Op can change. The inputs to the operation are denoted by $i?$, and the outputs by $o!$. inv is a state invariant obtained from the Δ-schema in the declaration of Op, and w consists of variables in S together with the operation outputs.)

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 under maintenance of interpretation using this function.

We can add CSP [Hoa85] to the heterogeneous basis by translating from CSP to *action systems* [Bac90] following the work of [WoM91]. An action system consists of a state, an initialization, and a number of labelled guarded commands on the state (a labelled guarded command is called an *action*). An example is shown below.

$$\begin{aligned}&\textbf{var } n \bullet \textbf{initially } n := 0\\ &count : n < 100 \to n := n + 1\\ &reset : true \to n := 0\end{aligned}$$

The initialization is executed, and then repeatedly one of the labelled commands with a true guard is chosen and executed. The system deadlocks if no guard is true, and diverges whenever a command aborts.

A communicating sequential process consists of an alphabet of events, and a set of behaviours described in one of the models of CSP: traces, failures, or failures-divergences.

In [WoM91] it is shown how to construct the traces, failures, and divergences of an action system, thus mapping from action systems into CSP. First, define for any sequences of actions s and t, the *sequential composition* P_s, as follows:

$$P_{\langle\,\rangle} \mathrel{\widehat{=}} \textbf{skip}, \quad P_{\langle a\rangle} \mathrel{\widehat{=}} P_a, \quad P_{s^\frown t} \mathrel{\widehat{=}} P_s;\ P_t$$

We can now construct the traces, failures and divergences. Consider an action system P with initialization P_i and a set of actions A. Three laws from [WoM91] are used for calculating traces, failures, and divergences of an action system.

Law 2.1 *A sequence tr is a* **trace** *of P providing that* $\neg\, wp(P_{\langle i\rangle^\frown tr}, false)$.

Law 2.2 *For a sequence of actions tr and a set of actions ref, the pair (tr, ref) is a* **failure** *of P if tr is a trace of P and:*

$$\neg\, wp(P_{\langle i\rangle^\frown tr}, \exists\, x : ref \bullet gd\ P_x)$$

where $gd\ P_x$ is the guard of the action P_x.

Law 2.3 *A sequence tr is a* **divergence** *if* $\neg\, wp(P_{\langle i\rangle^\frown tr}, true)$.

Justifications for the laws are given in [WoM91].

The transformation from a CSP specification (given in terms of traces, failures, and divergences) into an action system is also possible. Suppose we have a set of traces T, a set of failures F, and a set of divergences D. First we construct a set of actions L (these are simply names for actions). An action system P for this specification is as follows. The declaration and initialization of P is

$$\begin{aligned}&\textbf{var } tr : L^*;\ \mathcal{R} : \mathbb{P}\, L \bullet\\ &\textbf{initially } tr := \langle\,\rangle;\ \mathcal{R} : [\ tr \notin D, (tr, \mathcal{R}) \in F\ ]\end{aligned}$$

and for every $l \in L$ we form the guarded command

$$l : (l \notin \mathcal{R} \wedge (tr \frown \langle l \rangle) \in T) \rightarrow tr := tr \frown \langle l \rangle ;\ \mathcal{R} : [\ tr \notin D, (tr, \mathcal{R}) \in F\ ]$$

Further details can be found in [WoM91].

The issue of whether specific combinations of formal notations in the heterogeneous basis are usable together is not directly considered here. Feasibility (or compatibility) of use depends on the context in which the notations are to be used, and on how compositions between notations are to be defined. We do provide some evidence that particular notations are compatible, and can be used productively together (see the examples, and the further case studies in [Pai97]). More work remains to be done on examining the soundness of using all combinations of the notations of the heterogeneous basis.

2.2 Semiformal translations

The heterogeneous basis contains semiformal notations, including those from SADT [MaM88], Coad-Yourdon OOA/D [CoY90], and SA/SD [DeM79, YoC79]. To include a semiformal notation in the basis, we must fix an interpretation for it and then express specifications in this notation in one of the formalisms in the basis [Pai97]. If this interpretation or formalization is not appropriate for a development setting, then it should be changed. Once a formalization of a semiformal specification has been constructed, the formalization can be used to check for ambiguity or inconsistency.

We demonstrate how to add semiformalisms to the heterogeneous basis by using examples of SADT notations. Other semiformal notations, e.g., data flow diagrams and object notations, are dealt with in [Pai97].

There are many interpretations that might be taken for a semiformalism. In particular, an interpretation and formalization 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 altering the rest of the heterogeneous basis.

An SADT actigram box is shown in Fig. 2. An actigram is made up of interconnected boxes and arrows, with boxes representing functions and arrows representing data flow. Actigram boxes may be annotated with processing details, just as data flow diagrams may be annotated with process specifications (PSPECs).

The interpretation placed on an actigram is that it represents an operation on a state; this maps conveniently into a Z style of specification. If this interpretation is inappropriate for the task at hand, it can be changed according to the users' needs. For example, with this interpretation (and formalization) it will not be straightforward to model nondeterminism or triggering conditions. If we need to

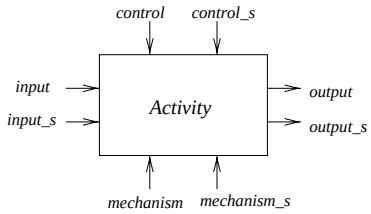

Fig. 2. An SADT actigram

model such concepts, we should use a more appropriate formalization (e.g., akin to [LKP91, ZaJ95]). We represent the actigram of Fig. 2 in Z as follows. First, we define appropriate types for the inputs, outputs, mechanisms, and controls. We create a state schema, $ActivityState$, for the actigram. This represents the part of the outputs of the actigram that are used elsewhere, and is equivalent to declaring an implicit data store.

$$\begin{array}{|l} \hline ActivityState \\ output\_s : O_s \\ \hline \end{array}$$

We next construct a Z specification of the box. In Fig. 2, flow labelled with a $\_s$ suffix comes from or goes to another box. Non-suffixed labels indicate data flow from or to the environment. In the Z schema we annotate external interactions with the Z syntax for input and output, and do not annotate the internal interactions.

A Z schema for $Activity$ is as follows.

$$\begin{array}{|l} \hline Activity \\ control? : C_e \\ input? : I_e \\ output! : O_e \\ mechanism? : M_e \\ \Xi input\_s \\ \Xi mechanism\_s \\ \Xi control\_s \\ \Delta ActivityState \\ \hline \end{array}$$

Each of $input\_s$, $mechanism\_s$, and $control\_s$ are names of state schemas declared elsewhere. They can be annotated with Δ instead of Ξ if necessary. An invariant may be added to $Activity$, by formalizing any associated processing details, if such information is important for proofs.

An SADT datagram box is shown in Fig. 3.

Boxes represent data, and arrows represent activities on data. The interpretation we place on a datagram box is that it is an entity (a set), and arrows between datagrams (or between the environment and a box) represent relations between entities. This can be modelled in Z as follows.

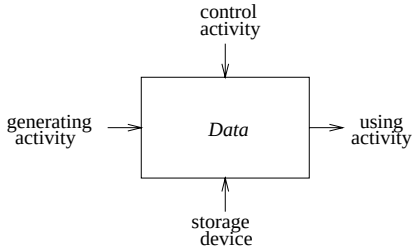

Fig. 3. An SADT datagram

1. Create a type $[ENVTYPE]$, the type of the environment. Model the environment as a state schema with component env.

$$\begin{array}{|l} \textit{Environment} \\ \hline env : ENVTYPE \\ \hline \end{array}$$

   All external interactions will be with the entity *Environment*.
2. For each datagram box *Data*, declare a type $[DATA]$. Represent the box in Z as a state schema.

$$\begin{array}{|l} \textit{Data} \\ \hline data : DATA \\ \hline \end{array}$$

3. Each arrow labelled r between the environment and a datagram box D is modelled as a one-to-one function from the environment to the box.

$$r : Environment \rightarrowtail D$$

   If there is more than one instance of D in the system that is represented by the box, then the relation should instead be one-to-many.
4. Each arrow r from a datagram D to the environment is described as a one-to-one function from the box to the environment.

$$r : D \rightarrowtail Environment$$

   If there is more than one instance of D in the system that is represented by the box, the relation should instead be many-to-one.
5. Each arrow from a datagram D_1 to a datagram D_2 is modelled as a pair of appropriately-named relations. For example, consider the arrow in Fig. 4. It is described in Z as follows.

$$\begin{array}{l} gen : D_1 \leftrightarrow D_2 \\ use : D_2 \leftrightarrow D_1 \end{array}$$

   Constraints on the domain and range of the relation can be added as invariants, e.g., to make relations one-to-one.

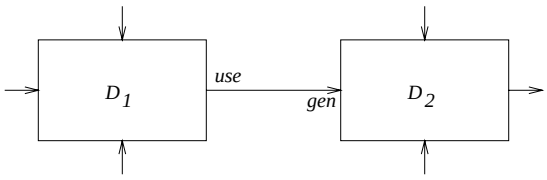

Fig. 4. A use-generate relationship

The addition of other semiformalisms (e.g., data flow diagrams, structure charts, object notations, pseudocode) is considered in [Pai97]. Therein, examples of how to extract semiformal specifications from formal specifications are also considered.

3 A Meta-Method for Formal Method Integration

Heterogeneous notations can be used in the production of a meta-method for formal method integration. We present such a technique here. The meta-method describes an abstract strategy for constructing relationships between procedural steps.

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.

1. *Fix a base method.* Fixing a base method is step aimed at assisting method engineers in determining roles that individual methods can play in the integrated method. A base method can suggest a set of steps (i.e., a partial meta-model [Met94]) that is to be supported and complemented by other (invasive) methods. A base method may support more of the software development cycle than other methods; it may also provide those steps that a developer may want to use the most during development.
2. *Choose the invasive method(s).* Invasive methods augment, are embedded, or are interleaved with the base method. In this step, possible relationships between the base and invasive methods are decided. The selection of invasive methods might be done in terms of: notational convenience, e.g., for adding operational details to flow diagrams, or for adding formality to semiformal specifications; methodological convenience, e.g., for adding new sets of procedures to a base method, such as procedural refinement to a non-refinement based technique; or, internal constraints dictated by requirements, management or company policy, or regulatory bodies.
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, notations are added to a method, and the method steps are generalized to using the new notations. An example of a generalization integration is demonstrated in [WiZ92], where SA is combined with Larch.
   - *Relation.* Relation of method steps can follow generalization. Relationships between the (generalized) base steps and (generalized) invasive steps are defined. Examples of relationships include the following.
     - Linking of method steps, by defining a translation between notations of different methods, e.g., as in the SAZ Project [PWM93].
     - Replacement of entire steps in a base method by (generalized) 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 method steps. Specific steps of one method are identified and are supplemented by steps from a second method. Supplementation does not change the ordering of steps, i.e., the ordering in the integrated method is identical to that in the method being supplemented. Invariance of ordering can be obtained by ensuring that the steps being added do not overlap with steps of the supplemented method outside of those method steps being supplemented.
     - Parallel use of steps, by describing relationships that interleave the use of two or more separate sets of method steps. An example of this kind of relation is suggested in [LKP91].
5. *Guidance to the user.* Hints, examples, and suggestions on how the integrated method can be used is provided.

The meta-method does not provide a formal (meta-) model of each method (e.g., as is done in meta-modelling techniques like [Met94]); for this reason, we consider the meta-method to be a "lightweight" approach to method integration. The meta-method also requires that all notations have (or can be given) a formal semantics, and that the method engineer eliminate syntactic ambiguity among the notations of the methods.

In the next two sections, we use the meta-method to integrate formal and semiformal techniques, and use these examples to discuss some of the properties of the meta-method.

4 Integrating Formal Methods

Of the formalisms considered in the heterogeneous basis, two include methods based on procedural refinement; the remaining techniques are specification styles (possibly with rules for data transformation), associated with informal rules for writing the specification, and for checking for its consistency.

We integrate several formal methods (a Z 'house method', Morgan's refinement calculus, predicative programming) using the meta-method of the previous section. We choose the Z house method as the base method, in order to make use of its specification style. The refinement calculus and predicative programming are selected as the invasive methods. A heterogeneous basis containing these notations (and translations between them) was constructed in Section 2. For each combination of used notations, the use of notations is restricted to those mutually expressible specifications (i.e., when combining Z and predicates, no miracles or havoc specifications are used).

In applying Step 4 of the meta-method, we first generalize the Z house method specification procedures (that require informal documentation of specification parts) to include the predicative notation and the refinement calculus notation. Then, we supplement the Z house method steps with proof rules (for procedural refinement and data transformation) from predicative programming and the refinement calculus. The supplementation step requires us to show how procedural refinement (and other proof techniques, e.g., for data transformation) apply to heterogeneous specifications. We summarize how procedural refinement applies to heterogeneous specifications here; other proof techniques are discussed in [Pai97].

The procedural refinement rules are based on the refinement relations from [Heh93] and [Mor94]. Their definitions are summarized here for completeness.

Definition 3 [Mor94]. A specification statement S is refined by a specification statement T (written $S \sqsubseteq T$) if $\forall R' \bullet wp(S, R') \Rightarrow wp(T, R')$, where R' is a relation on pre- and poststate.

Definition 4 [Heh93]. A predicative specification P is refined by a specification Q if $\forall \sigma, \sigma' \bullet (P \Leftarrow Q)$, where σ and σ' denote the prestate and poststate, respectively.

We now outline a small collection of rules for refinement over formal heterogeneous specifications. Further rules—and results on proof of satisfiability and data transformation—can be found in [Pai97].

4.1 Application of refinement

We briefly summarize several rules that demonstrate how to apply the refinement relations $\Leftarrow$ and $\sqsubseteq$ to operands of types other than predicate and specification statement. In the following, σ is the state.

Rule 4.1 *Let P and Q be predicates. If $\forall \sigma, \sigma' \bullet (P \Leftarrow Q)$ then $P \sqsubseteq Q$.*

Proof:

$$\begin{aligned} P \sqsubseteq Q &= \forall \sigma, R' \bullet (wp(P, R') \Rightarrow wp(Q, R')) \\ &\qquad \{\text{translation } PredToWp \text{ from [Pai97]}\} \\ &= \forall \sigma, R' \bullet ((\forall \sigma' \bullet P \Rightarrow R') \Rightarrow (\forall \sigma' \bullet Q \Rightarrow R')) \\ &\qquad \{\text{monotonicity}\} \\ &\Leftarrow \forall \sigma, R' \bullet \forall \sigma' \bullet ((P \Rightarrow R') \Rightarrow (Q \Rightarrow R')) \\ &\qquad \{\text{antimonotonicity}\} \\ &\Leftarrow \forall \sigma, \sigma' \bullet (P \Leftarrow Q) \end{aligned}$$

Rule 4.2 *For Z schemas S_p, S_q with invariants P and Q,*

$$S_p \sqsubseteq S_q = (\exists \sigma' \bullet P \Rightarrow \exists \sigma' \bullet Q) \wedge (\exists \sigma' \bullet P \Rightarrow \forall \sigma' \bullet (P \Leftarrow Q)).$$

Proof: By translation *ZToSS*, Definition 3, and manipulation.

We can also apply the refinement relation of the predicative notation to non-predicate operands. We show how it applies to Z schemas and Larch interface language operations.

Rule 4.3 *For Z schemas S_p, S_q with invariants P and Q,*

$$(S_p \Leftarrow S_q) = \forall \sigma, \sigma' \bullet ((\exists \sigma' \bullet Q) \Rightarrow Q) \Rightarrow ((\exists \sigma' \bullet P) \Rightarrow P)$$

Proof: By translations *ZToSS*, *SSToPred*, and manipulation.

Rule 4.4 *Let L and M be LCL functions with identical function interfaces, where both have* `modifies` *clause w, and where L and M have* `requires` *clauses P and U respectively, and* `ensures` *clauses Q and V respectively. Then*

$$L \Leftarrow M = \forall\, w, w' \bullet ((P \Rightarrow Q) \Leftarrow (U \Rightarrow V)).$$

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

Rule 4.5 *If $S \mathrel{\widehat{=}} w : [\ pre, post\ ]$ is a specification statement and $pred_S$ is its predicate translation, then $S \sqsubseteq pred_S$.*

Proof:

$$\begin{aligned} S \sqsubseteq pred_S &= \forall R' \bullet wp(S, R') \Rightarrow wp(pred_S, R') \\ &\qquad \{\text{definition, distributivity, } PredToWp\ \} \\ &\Leftarrow \forall w', R' \bullet (pre \wedge (post \Rightarrow R')) \Rightarrow ((pre \Rightarrow post) \Rightarrow R'), \end{aligned}$$

and the last line is a theorem.

We can generalize the result of Rule 4.5: two further rules allow us to introduce predicates or specification statements in the process of a development.

Rule 4.6 *Let P and Q be predicates, and $spec_P$ and $spec_Q$ their translations into specification statements (using translation $PredToSS_1$ or $PredToSS_2$). If $P \Leftarrow Q$ then $P \sqsubseteq spec_Q$.*

Rule 4.7 *Let S and T be specification statements, and $pred_S$ and $pred_T$ their translations into predicates. If $S \sqsubseteq T$ then $S \sqsubseteq pred_T$.*

Proof of 4.7 By Rule 4.5, $T \sqsubseteq pred_T$. If $S \sqsubseteq T$, then by monotonicity of $\sqsubseteq$, $S \sqsubseteq pred_T$.

Finally, we discover that refinement is actually preserved over translation from specification statements to predicates.

Rule 4.8 *For specification statements S and T, and their predicate translations $pred_S$ and $pred_T$, $(S \sqsubseteq T) \Rightarrow (pred_S \Leftarrow pred_T)$.*

Proof: By translation $PredToSS$, and since $[a \Rightarrow (c \wedge (b \Leftarrow d))] \Rightarrow [(a \Rightarrow b) \Leftarrow (c \Rightarrow d)]$ is a tautology for all a, b, c, and d.

Other results and rules are possible; they can be obtained by generalizing or specializing the results presented, and by using the basic translations.

4.2 Refinement over conjunction and disjunction

We describe refinement rules for application over conjunction and disjunction. More rules are described in [Pai97]; see [War93] for an alternative approach to combining specification statements with Z combinators. In the following, let S, S' and T be specification statements, and P and Q be predicates.

Rule 4.9 *If $S \sqsubseteq T$ then $P \wedge S \sqsubseteq P \wedge T$.*

Proof:

$$\begin{aligned} P \wedge S \sqsubseteq P \wedge T &= \forall R' \bullet (wp(P \wedge S, R') \Rightarrow wp(P \wedge T, R')) \\ &\qquad \{PredToWp, \text{ splitting law}\} \\ &= \forall R', \sigma' \bullet ((P \wedge S \Rightarrow R') \Rightarrow (P \wedge T \Rightarrow R')) \\ &\qquad \{\text{antimonotonicity}\} \\ &\Leftarrow \forall \sigma' \bullet (P \wedge T \Rightarrow P \wedge S) \\ &\qquad \{\text{monotonicity, Rule 4.7}\} \\ &\Leftarrow S \sqsubseteq T \end{aligned}$$

Rule 4.10 *Let $pred_S$ and $pred_T$ be the predicate specification equivalents of S and T (assuming S and T are not angelic). If $\forall \sigma, \sigma' \bullet (pred_S \Leftarrow pred_T)$ then $P \vee S \sqsubseteq P \vee T$.*

Combining Rule 4.10 with Rule 4.8, we determine that:

Corollary 1 *If $S \sqsubseteq T$ then $P \vee S \sqsubseteq P \vee T$.*

Specification statements that are conjoined or disjoined together can also be refined by parts.

Rule 4.11 *Providing that S, S', and T are all expressible in predicates,*

$$(S \wedge T \sqsubseteq S' \wedge T) \Leftarrow S \sqsubseteq S',$$
$$(S \vee T \sqsubseteq S' \vee T) \Leftarrow S \sqsubseteq S'.$$

Rule 4.11 gives us a form of monotonicity over predicate combinators. The proof is similar to that for Rule 4.9.

As is shown in [War93], refinement over schema conjunction and disjunction is not monotonic. However, we can combine schemas (and other specifications) via predicate operators $\vee$ and $\wedge$ and refine them. Let S_x, S_y, and S_z be schemas. Then:

Rule 4.12

$$(S_x \wedge S_y \sqsubseteq S_z \wedge S_y) \Leftarrow S_x \sqsubseteq S_z\,,$$
$$(S_x \vee S_y \sqsubseteq S_z \vee S_y) \Leftarrow (S_x \Leftarrow S_z)\,.$$

Rules for refinement over sequential composition are given in [Pai97], as are rules for heterogeneous development, i.e., rules for changing notation during a development via a refinement step. We give an example of how to use some of these rules in Section 6.

5 Combining Formal and Semiformal Methods

Structured Analysis and Design Technique (SADT) [MaM88] was invented by Ross in the early 1970s. It claims to allow easy representation of system characteristics like control, feedback and mechanism. It contains explicit procedures for group work, and is based on the specification and elucidation of diagrams. There is a rigorous set of rules for the construction of the diagrams. We consider a basic version of the SADT method here solely in the context of software specification and design.

We apply the meta-method from Section 3 in integrating SADT with predicative programming. The base method is SADT; predicative programming is the invasive method. SADT procedures will be generalized to using predicative notations. Specifically, the procedures for authoring and data modelling will be generalized to use predicative notations. After generalization, the procedures will be supplemented by predicative programming refinement rules. In particular, the SADT steps for refinement, data modelling, authoring, and implementation will be supplemented by predicative refinement rules. We depict the integrated method in Fig. 5.

In Fig. 5, ellipses represent procedure steps (and thick arrows between ellipses represent ordering of steps), and boxes describe heterogeneous products. Arrows from ellipses to boxes denote usage or creation of the product by the procedure step.

In more detail, the integrated method procedure is as follows.

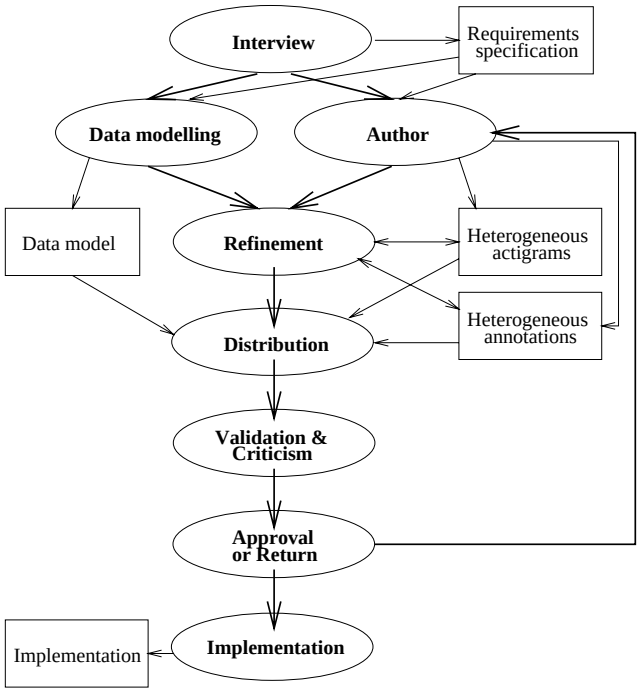

Fig. 5. SADT/predicative programming integrated method

1. *Interview.* The clients, customers, and users are interviewed so as to gather requirements. A requirements specification is written in a notation appropriate for the task.

   ASIDE. Steps 2, 3, and 4 can occur in parallel. We write them sequentially here for ease of presentation. **END OF ASIDE.**
2. *Authoring of heterogeneous specifications.* A heterogeneous system specification is constructed. Actigrams, datagrams, and annotations are produced using compositions of SADT notations and predicate notations. Typically, the predicate notation will be confined to the expression of annotations, but visual depictions of predicates could be used, too.
3. *Data modelling and data dictionary construction.* Data is designed and modelled, and a heterogeneous data dictionary (written using pseudocode, regular expressions, and predicate notations) is constructed.
4. *Refinement of heterogeneous specifications.* The actigrams and datagrams are refined hierarchically. SADT rules are used to syntactically check the steps. Predicate partial specifications are refined using Definition 4, and proof rules from [Heh93].
5. *Distribution.* The refined heterogeneous specification is distributed to a review committee. The committee should be familiar with the SADT notations and conventions, and at least one member should be familiar with the

predicative notation, for reading processing details. The specification writers should be prepared to informally explain or document the formal parts of the heterogeneous specification, and to explain the heterogeneous basis.

6. *Validation and Criticism.* The specification is reviewed and criticized. Syntax is validated according to SADT rules. The syntax and satisfiability of predicate specifications is checked. The connections between the SADT boxes and the predicate specifications are validated given the precise meaning of the composition. The SADT and predicate interfaces should be syntactically verified using the syntactic rules of SADT; predicate parts will be informally treated as SADT annotations for the purposes of review.
7. *Approval or Repair.* The specification is approved, or sent back to Step 2 (or Step 4) for repair.
8. *Implementation.* The specification is implemented in a hierarchical fashion by standard SADT practice and by predicate refinement, data transformation, transliteration, and component integration. Testing of the product should also occur.

At several steps of the method, SADT produces documentation (e.g., glossaries, diagrams, supplements, narratives; see [MaM88]). We do not discuss the effects of method integration on these products here in the interests of conserving space. One can take the view that the predicative notation is just another form of documentation for the SADT method, though it is documentation that can be formally manipulated.

Some of the properties we obtain with this integrated method are as follows.

- *Restrictability:* the predicate notation need be used only when required for describing actigram annotations. Restrictability is convenient to obtain with this integration, due to the structured style of specification offered by the use of SADT actigram and datagram notations.
- *Gradual introduction:* predicative programming can be gradually introduced into the SADT method by restricting the use of predicates to the specification and development of those system parts where the notation seems necessary, and by applying restrictability over time.
- *Semantic gaps:* the semantic gaps introduced by using a formal method in a development are reduced due to the heterogeneous basis and restrictability.
- *Method transformation:* transformations between the heterogeneous specifications of the SADT/predicate method and pure predicate or pure SADT are possible, as are partial transformations, by the translations from Section 2 and by producing informal extraction procedures.

6 A Small Formal Example

We have only the space to present a very small example of using integrated formal methods. For this reason, we choose to demonstrate the use of a combination of formal methods. The example combines predicative programming [Heh93] and the refinement calculus [Mor94]. The intent is only to give the flavour of using

multiple methods together. Further examples are presented in [Pai97]; these examples in particular include integrations of formal and semiformal methods, and further examples of integrating and using multiple formal methods.

The problem is as follows. We are presented with two equal-length lists of lower-case letters (representing, typically, English words). We are to determine if the two lists are anagrams (i.e., permutations) of each other. If they are, we are to compute the number of *position differences* over the two lists (a position difference for a character c that is in both L and M is the absolute difference in indices for c in L and M. A strategy must be developed for handling multiple occurrences of characters in lists.)

We specify the problem in two parts. The two lists are L and M. We declare a constant list $ALPH = [`a; `b; ..; `z]$. The global variables are b, which is set to true iff L and M are anagrams; and np, the number of position differences between L and M. The initial specification is as follows.

$$
\begin{array}{l}
\mathit{anagram} \ . \\
\textbf{if } b \rightarrow \mathit{numpos} \\
[]\neg b \rightarrow \textbf{skip} \\
\textbf{fi}
\end{array}
$$

(The predicate . operator is dependent composition, i.e., sequencing.) *anagram* is a predicate. It is defined as follows, using the notation of [Heh93].

$$
\begin{aligned}
\mathit{anagram} \mathrel{\widehat{=}} b' = \forall j : 0,..26 \bullet ((\natural\, \S i : 0,..\#L \bullet Li = ALPH(j)) = \\
(\natural\, \S i : 0,..\#L \bullet Mi = ALPH(j)))
\end{aligned}
$$

(§ is the bunch quantifier [Heh93], and $\natural$ is the bunch cardinality operator.) *numpos* is a specification statement that counts the number of position differences between L and M.

$$
\begin{aligned}
\mathit{numpos} \mathrel{\widehat{=}}\ np : [\ np' = \sum i : 0,..26 \bullet \sum j : 0,..\#N \bullet \\
abs([k : 0;\ ..\#L \mid L(k) = ALPH(i)]j - \\
[k : 0;\ ..\#L \mid M(k) = ALPH(i)]j) \\
]
\end{aligned}
$$

where $N \mathrel{\widehat{=}} \S k : 0,..\#L \bullet L(k) = ALPH(i)$. Note that for multiple occurrences of c, we always take the difference of the smallest *unmatched* indices.

The refinement relation $\Leftarrow$ is monotonic over dependent composition. Therefore, we can refine *anagram* without affecting *numpos* or the guarded selection. The approach to refining the *anagram* specification will be to iterate through the alphabet, and for each letter of the alphabet count the number of occurrences of the letter in each of lists L and M. L and M are permutations iff they have the same number of occurrences of each letter. A refinement goes as follows. First, define P like so.

$$
P \mathrel{\widehat{=}} ((\natural\, \S i : 0,..\#L \bullet Li = ALPH(j)) = (\natural\, \S i : 0,..\#L \bullet Mi = ALPH(j))
$$

Then:

$$anagram \Leftarrow b := \top .\ k := 0 .\ b' = b \wedge \forall j : k, ..26 \bullet P \qquad (1)$$

The predicate at the end of the dependent composition (1) is refined as a selection.

$$\begin{array}{l} \Leftarrow \textbf{if } k = 26 \textbf{ then skip} \\ \quad \textbf{else } k \neq 26 \Rightarrow b' = b \wedge \forall j : k, ..26 \bullet P \end{array}$$

The predicate in the **else**-branch can be implemented by defining three new variables, s, sM, and sL, that will be used to iterate through the lists L and M and to keep track of the number of occurrences of the letter $ALPH(k)$.

$$\begin{array}{l} \Leftarrow \textbf{var } s, sL, sM : nat \bullet \\ \quad s, sL, sM := 0, 0, 0 . \\ \quad k \neq 26 \Rightarrow b' = b \wedge \forall j : k, ..26 \bullet Q \ . \qquad \lhd \\ \quad k := k + 1 \ .\ b' = b \wedge \forall j : k, ..26 \bullet P \end{array}$$

where

$$\begin{array}{l} Q \mathrel{\widehat{=}} (sL + \natural \S i : s, ..\#L \bullet Li = ALPH(j)) = \\ \qquad (sM + \natural \S i : s, ..\#L \bullet Mi = ALPH(j)) \end{array}$$

Finally, the predicate marked with a $\lhd$ can be refined by a simple selection that updates the counters.

$$\begin{array}{l} \Leftarrow \textbf{if } s = \#L \textbf{ then } b := b \wedge (sL = sM) \textbf{ else } (\\ \quad sL, sM, s := sL + (\textbf{if } Ls = ALPH(k) \textbf{ then } 1 \textbf{ else } 0), \\ \qquad\qquad sM + (\textbf{if } Ms = ALPH(k) \textbf{ then } 1 \textbf{ else } 0), \\ \qquad\qquad s + 1 \ . \\ \quad k \neq 26 \Rightarrow b' = b \wedge \forall j : k, ..26 \bullet Q) \end{array}$$

The development is now complete. Notice that we have used a recursive refinement [Heh93] in the last step above, instead of developing a loop structure.

The next step is to refine *numpos*. We can do this using $\sqsubseteq$ due to Rule 4.8. The refinement requires two loops. The outer loop will iterate over the alphabet, while the inner loop will iterate over the lists and will calculate the lists of indices where specific letters of the alphabet appear. The refinement proceeds as follows (omitting details due to space constraints), based on the standard development steps outlined in [Mor94] for loops (this includes using *leading* and *following* assignment laws).

$$\begin{array}{l} numpos \sqsubseteq i, np := 0, 0; \\ \qquad\qquad \textbf{do } i \neq 26 \rightarrow \\ \qquad\qquad\ np : [\ (0 \leq i < 26) \wedge I, I'\ ]; \qquad \lhd \\ \qquad\qquad\ i := i + 1 \\ \qquad\qquad \textbf{od} \end{array}$$

An invariant is $I \mathrel{\widehat{=}} np = \sum r : 0, ..i \bullet R$, where R is:

$$R \mathrel{\widehat{=}} \sum j : 0, ..\#N \bullet \\ abs([k : 0;\ ..\#L \mid L(k) = ALPH(r)]j - [k : 0;\ ..\#L \mid M(k) = ALPH(r)]j)$$

A bound function is $26 - i$. The new specification statement marked with a $\lhd$ can be refined by adding three fresh variables and a loop over the list; these variables are used in determining the indices for a specific letter of the alphabet. The results of the list determination are subtracted, and this new result added to np to preserve the loop invariant. We again apply the laws leading and following assignment, and introduce the loop using the checklist from [Mor94].

$$\begin{array}{l} \sqsubseteq\ |[\ \mathbf{var}\ j : \mathbb{N};\ A, B : \mathrm{seq}_{\#L}\,\mathbb{N} \cdot \\ \quad j, A, B := 0, \langle\rangle, \langle\rangle; \\ \quad \mathbf{do}\ j \neq \#L \rightarrow \\ \quad\quad A, B : [\ (0 \leq j < \#L) \wedge J, J'; \ ]; \quad \lhd \\ \quad\quad j := j + 1 \\ \quad \mathbf{od}; \\ \quad np := np + \sum k : 0, ..\#A \bullet abs(A(k) - B(k)) \\ \ ]| \end{array}$$

The invariant J in the above loop is

$$J \mathrel{\widehat{=}} A = [k : 0;\ ..j \mid M(k) = ALPH(i)] \wedge B = [k : 0;\ ..j \mid L(k) = ALPH(i)]$$

The bound function is $\#L - j$. The final refinement of the specification marked with $\lhd$ is straightforward: a selection is added that concatenates the current value of j to the lists A and B if the conditions in the invariants are met.

$$\begin{array}{l} \sqsubseteq\ \mathbf{if}\ M(j) = ALPH(i) \rightarrow A := A \frown [j]\ \mathbf{fi}; \\ \quad\quad \mathbf{if}\ L(j) = ALPH(i) \rightarrow B := B \frown [j]\ \mathbf{fi} \end{array}$$

(where $\frown$ is list concatenation). If $\sum$ is not an implemented combinator in the programming language, then we need to refine the last sum (and addition to np) in the last line of the refinement tree. This can be done by introducing a simple loop or recursive refinement (which we omit here due to space constraints). Note that such a refinement can be done using either predicative refinement *or* weakest precondition refinement; the preferences of the developer can be taken into account.

By Rule 4.8 and the monotonicity of $\Leftarrow$ and $\sqsubseteq$ over dependent composition, the composition of the refinements is a refinement of the original specification, and we have implemented a solution.

7 Conclusions

We have briefly described a meta-method for integrating formal methods with other methods. We have provided two examples of using the meta-method: an integration of several formal methods; and an integration of a program design calculus with a structured method. The approach to integration is based on combining notations; formal method integration is based on combining notations, and manipulating procedures of methods to accommodate and use the new notations. 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 Ric Hehner, Pamela Zave, and the three anonymous referees for their excellent suggestions and advice.

References

[Bac90] R.J.R. Back. Refinement calculus II: parallel and reactive programs. In *Stepwise Refinement of Distributed Systems*, LNCS 430, Springer-Verlag, 1990.

[BaV89] R.J.R. Back and J. von Wright. A Lattice-Theoretical Basis for a Specification Language. In *Mathematics of Program Construction*, LNCS 375, Springer-Verlag, 1989.

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

[CoY90] P. Coad and E. Yourdon. *Object-oriented Analysis*, Prentice-Hall, 1990.

[DeM79] T. DeMarco. *Structured Analysis and System Specification*, Yourdon Press, 1979.

[DeM82] T. DeMarco. *Controlling Software Projects: Management, Measurement, and Estimation*. Yourdon Press, 1982.

[GuH93] J.V. Guttag and J.J. Horning. *Larch: Languages and Tools for Formal Specification*, Springer-Verlag, 1993.

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

[Ham94] J. Hammond. Producing Z Specifications from Object-Oriented Analysis. In *Proc. Eighth Z User Meeting*, Cambridge, Springer-Verlag, 1994.

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

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

[Hil91] J.V. Hill. Software development methods in practice. In *Proc. Sixth Annual Conference on Computer Assurance*, 1991.

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

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

[Kin90] 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.

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

[LKP91] 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.

[MaM88] D.A. Marca and C.L. McGowan. *SADT – Structured Analysis and Design Technique,* McGraw-Hill, 1988.

[Met94] Project MetaPHOR Group, MetaPHOR: Metamodeling, Principles, Hypertext, Objects and Repositories. Technical Report TR-7, University of Jyvaskyla, 1994.

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

[Pai97] R.F. Paige. *Formal Method Integration via Heterogeneous Notations,* PhD Dissertation, July 1997.

[PWM93] 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.

[ScR77] K. Schoman and D. Ross. Structured Analysis for requirements definition, *IEEE Trans. on Software Engineering*, 3(1), 1977.

[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. FME '93: Industrial-strength Formal Methods*, LNCS 670, Springer-Verlag, 1993.

[WiZ92] J.M. Wing and A.M. Zaremski. Unintrusive ways to integrate formal specifications in practice. In *VDM '91: Formal Software Development Methods*, Fourth International Symposium of VDM Europe, LNCS 551, Springer-Verlag, 1992.

[WoM91] J.C.P. Woodcock and C.C. Morgan. Refinement of state-based concurrent systems. In *VDM '90: VDM and Z - Formal Methods in Software Development*, Third International Symposium of VDM Europe, LNCS 428, Springer-Verlag, 1990.

[YoC79] E. Yourdon and L. Constantine. *Structured Design,* Prentice-Hall, 1979.

[ZaJ93] P. Zave and M. Jackson. Conjunction as Composition, *ACM Trans. on Software Engineering and Methodology*, 2(4), October 1993.

[ZaJ95] P. Zave and M. Jackson. Where do operations come from? An approach to multiparadigm specification, *IEEE Trans. on 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.