Heterogeneous Specifications and their Application to Software Development
A Research Proposal

Richard F. Paige
Department of Computer Science, University of Toronto
`paige@cs.utoronto.ca, http://www.cs.utoronto.ca/~paige`

October 12, 1995

Abstract

We describe a course of research examining formal and semiformal *heterogeneous specifications*, i.e., compositions of partial specifications written in different textual and visual notations. We describe why we believe this to be an interesting topic for further inquiry, and suggest why such specifications might prove beneficial for use in software development.

1 Introduction

Formal specifications are starting to become more widely used in the development of programs. As software designers begin to investigate how formalism might be applied to their work, they soon discover the wealth of specification notations and methods that are available. They must come to grips with deciding which (if any) notations or methods to utilize in their work.

Semiformal specifications have been used for many years in program development. These are notations and concomitant methods–possibly with a partially or unspecified formal semantic basis–which prove attractive to designers for many reasons, perhaps the most significant being that the semiformal notations are often easier and cheaper to apply than formal ones.

There have been many formal specification notations (and formal methods) described in the literature. The first usable formal method was Hoare logic [Hoar69]. Weakest preconditions [Dijk76] have been used for program verification with success. The refinement calculus ([Back78], [Morg94]) is a rigorous approach to program design, based on a mathematical notion of refinement. Z [Spiv89] and VDM [Jone90] have met with some success in both subsidiary support and central construction programming rôles. And predicative programming [Hehn93] is an alternative to the refinement calculus with a simpler semantics and definition of refinement.

There have been at least as many semiformal specification notations (and methods) as formal ones described in the literature. Examples include data flow diagrams, structure charts, Structured Analysis and Design [DeMa79], Jackson Structured Design [Jack81], and so on.

It is our suggestion that much can be gained from using both formal and semiformal notations and techniques in *composition* when constructing programs. Specifications written using multiple notations are called *heterogeneous*:

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

The benefits of studying such approaches to specification would include the potential for:

- obtaining more flexibility in the application of both formal and semiformal notations and techniques to software development;
- smoothing the integration of formal specification notations into development methods;
- reducing the semantic gaps [PHG91] that arise in a development due to the use of an integrated formal notation;
- constructing a "catalogue" of the relative expressiveness and limitations (both syntactic and semantic) of notations.

There are also several caveats to be kept in mind about using a heterogeneous approach:

- For a given problem, it may be more difficult to simultaneously employ multiple notations instead of a single notation.
- It may not be cost or time-effective to take up the use of multiple notations in an attempt to solve a problem.
- It may be difficult to determine a heterogeneous decomposition for a given set of functional and nonfunctional constraints.

In order to accomplish heterogeneous use, and to hopefully obtain the aforementioned benefits (while answering the caveats listed above), it is necessary to explore several questions:

1. How are the various formal and semiformal notations for software development related?
2. What kinds of theorems, development rules, and heuristics can we derive to assist us in using heterogeneous specifications?
3. What is the effect of using heterogeneous specifications within existing software development methods?

In this proposal, we discuss some of the important ideas and problems to be considered when attempting to answer such questions. We present further detailed sub-questions that need to be considered, plus a description of how we plan to attack these questions. In an Appendix, we also describe some preliminary research results, and discuss how we hope to extend them. In a second Appendix, we present several examples of heterogeneous specifications and show how they might be used in development.

We assume enough of a familiarity with the formal and semiformal methods described above to understand a fairly high-level discussion. We suggest consulting [Paig94] and the other references for further details.

Formal methods have experienced only limited use in "realistic" programming situations. By showing how the development notions used in formal and semiformal methods can be combined, we may further promote the use of formal methods in those design situations where they may be most needed.

2 Research Directions

We propose a plan of research that will have the following scope and approximate chronological course. The most significant part of the work will be tied up in Parts 2 and 3.

1. *Identify a collection of formal and semiformal notations for study.*

   We propose to examine selections from the following:

   - formal: the *refinement calculus*, *Z*, *VDM*, the *predicative notation*, *weakest preconditions*, *CSP* [Hoar85], Hoare logic, functional notations (e.g., CIP-L [Part90]), algebraic specifications and Larch [Gutt93].
   - semiformal: *data flow diagrams*, entity-relationship models, *transition diagrams*, *Jackson diagrams*, *program description languages*, subsets of *programming languages*, *structure diagrams*, object-oriented notations, and decision tables.

   It is expected that the emphasis of our work will be on the italicized notations.

2. *Compare and unify the selected notations to form a heterogeneous framework.*

   This will involve:

   - Translating specifications from one formal notation into a second formalism.
   - Giving a formal semantics to the *semiformal* notations. We will have to determine in each case the formal notation(s) to use in giving the semantics[1]. Since in carrying out such formalizations we may have to place an explicit reading on otherwise semiformal notations, we likely will have to evaluate the various interpretations to see which is the most appropriate and general for heterogeneous use.

     > *ASIDE.* The results of these first two steps give us a framework within which we can use multiple notations in writing specifications. The next stages focus on refining the framework so as to obtain a better understanding of *when* different notations can or should not be used. *END OF ASIDE.*

   - Identifying the relative expressiveness of both formal and semiformal notations. For example, we can express both *angelic nondeterminism* and havoc in the refinement calculus, but cannot do so using the predicative notation. Here, we might also examine the effects of extending notations to include previously unrepresentable specifications (if indeed such enhancements are possible). By examining such notational limitations, we provide ourselves with a better idea of when certain combinations will be beneficial, since we will have more of a clear picture regarding the facilities–both syntactic and semantic–which are lacking in each notation.
   - Attempting to see if the so-determined "less expressive" notations are *significantly* less expressive (i.e., do the specifications that cannot be expressed in a particular notation allow us to describe systems that cannot be easily represented using other means? Do they demonstrably simplify program development?)

   Preliminary results regarding these issues are discussed in Appendix A.

[1]We can use formal *heterogeneous* specifications here if it seems worthwhile.

3. *Examine methods for expressing, manipulating, and refining heterogeneous specifications.*

   More precisely:

   - Develop theorems and rules for heterogeneous specification manipulation, especially regarding:
     - *specification refinement:* monotonicity, transitivity, partwise, casewise and stepwise refinement theorems for heterogeneous specifications.
     - *heterogeneous development:* extending a refinement relation to new notations and compositions of notations. This allows a development process to be heterogeneous, i.e., it allows developers to use multiple refinement relations *and* notations in development. This is discussed more in Appendix A.
     - *data refinement.*
     - *transfers of context between partial specifications in different notations:* viz., how should it be done?
     - *satisfiability and feasibility of heterogeneous specifications.*
     - *formalization by parts:* this is a notion of specification formalization analogous to *refinement* by parts. To formalize a semiformal specification by parts means to formalize subspecifications of interest while leaving all other parts unchanged. We wish to develop this notion for various semiformal notations, for it has the possibility of being very useful for when we want to obtain restrictions on the use of a formal notation. Dually, we may want to examine "unformalization by parts", too.
   - Discuss some examples of compositions and how they can be applied to various programming situations. Instances of this might include:
     - compositions of C code and formal specifications. This would allow programmers to include C statements like `printf("%d\n",a);` in formal specifications. This might be worthwhile since it could reduce the semantic gaps between specification and implementation notation and might make the formal specification notation more attractive to programmers. Furthermore, this would also allow (in certain cases) formal refinements to C code. These issues are discussed in more detail in Part 4.
     - compositions of data flow diagrams (or other graphical notations) and formal specifications: when are they feasible, and–notationally–how is it accomplished?
     - high-level descriptions of software architecture via heterogeneous specifications.
   - Examine criteria for suggesting whether or not a composition of notations might prove to be beneficial. We might base this on: (a) the *simplicity* of combined use (with respect to refinement and semantic definitions); or (b) the *expressiveness* of combined use. We will also have to examine the basic notion of *when* notations can feasibly be combined in order for us to be able to attempt to examine when they can be *beneficially* combined.
   - For semiformal notations, determine if there are non-trivial notions of *semiformal refinement.* Since we are able to give a formal semantics to certain semiformal notations, we can perhaps extend the refinement relations of various formalisms to these semiformal domains. We will need to develop theorems describing how to use these extended relations. For example, it should be possible to formalize the notion of data flow diagram refinement in various situations.
   - For semiformal notations, determine specification combinators. For the diagram-based notations, examine the notion of "diagrammatic composition" (i.e., if we have a formal

semantics for a diagram, we can, theoretically, compose it with another diagram. How should we denote such compositions?)

- Examine the construction of a small visual notation (e.g., see [HarD88]) for representing formal specifications. Such a facility will be important for facilitating the composition of diagrams and formal specifications, and may be useful in itself for its capability of expressing formal specifications visually. We will consider the use of a higraph-like notation here.

ASIDE. We should clarify now that the emphasis of our work is *not* on the translations developed in Section 2. We are much more interested in the actual heterogeneous specifications themselves, how they are created, and how they can be used. Development of the translations is, of course, necessary in order to make our presentation concrete, but the rôle they play in our work is secondary to that of the specifications and compositions themselves. *END OF ASIDE.*

4. *Examine the effects of heterogeneous specifications on existing development methods.*

There are two particular cases of this which interest us (although all permutations will be examined in some way):

- *Formal-formal compositions:* How does our ability to compose specifications in multiple formal notations affect a software development method that is or can be based on one of the notations? (This would be partially answered in the preceding section.)
- *Semiformal-formal compositions:* How does our ability to compose semiformal and formal specifications affect a development that is based on the semiformal notation? For example, if we can compose Jackson diagrams with predicates–or, equivalently, a visual formalism used to represent predicates–how does our use of JSP or JSD change when we use the compositions in development? Can we introduce compositions in such a way so as to reduce the semantic gaps that arise between the phases of the development process?

It is this last instance of heterogeneous specifications which strikes us as quite interesting. It may be possible–by using heterogeneous specifications and, in particular, semiformal and formal compositions–to make formal methods more acceptable to the industrial programmer. By demonstrating how the use of mathematics-based methods can be combined with the use of current (and future) software development techniques and semiformal notations, we make our formal methods more viable, in the sense that the use of the formal method will be *restrictable*[2] and may be integrated into an existing software development process.

By considering heterogeneous semantics in this context, we essentially attempt to answer two important questions:

(a) How can we integrate a formal method into a software development method?
(b) How can we obtain more flexibility in using a formal method in realistic software construction situations?

Question (a) has been considered to a degree by others; for example, see [SFD92] or [PHG91]. The former work examines formal and informal integrations of Z, VDM, and algebraic specifications with Structured Analysis and Design. The latter paper considers metamorphic

[2]The method need only be used when desired (or necessary) in a particular problem.

programming. We propose attempting to formulate answers to (a) and (b) via the route of heterogeneous specifications, and in doing so hope to minimize some of the complexity and expense of applying formal methods in large-scale software construction. In the process, we also hope to examine issues of reuse, maintenance and evolution, and restrictability.

> *ASIDE.* One perspective on the work we are describing is that it is part of an attempt to fit program design calculi into software development processes. This problem (which is closely related to formal specification integration [SFD92]) has received limited discussion in the literature. There are several ways in which one might think of carrying out such an integration. The ideal place in a standard model of software development to perform the process is in the specification and design phase. This introduces some problems:
>
> - The transition from the notation used in analysis to the formal notation used in specification and design will be complicated. Significant semantic gaps arise.
> - The formal specification used in the design phase must adequately describe the software architecture specified in the analysis phases. Unfortunately, many program design calculi notations are poor at making such descriptions (mostly due to the unimplementability of certain specification combinators, and the lack of abstraction mechanisms).
> - With such an integration, making the transition to code is in many cases prohibitively costly, since expensive formal refinements (with or without detailed proofs) must be carried out.
>
> Our attempt to (partially) solve these problems is based on *not using only one notation in specification and design, but rather several simultaneously:* (at least) one formal notation in arbitrary composition with project-specific semiformal notations.
>
> Ideally, we want to be able to write specifications with arbitrary intermixings of formality and semiformality. Currently–without allowing compositions–we typically use only the two extreme cases of intermixing: formal specifications; and semiformal (or even informal) specifications. By allowing arbitrary compositions, we can reach the middle ground between the two extremes, which should prove eminently useful when used in large-scale development. *END OF ASIDE.*

One of the key ideas to keep in mind is that designers should not be restricted in terms of the notation they choose to use for specification and development: they should be able to use whatever notations (and development methods) they need–*providing that the notations in question can be formalized*[3]*!* From our point of view, there is no single programming language or formal specification notation or semiformal notation that should always be used when developing software. While there may always be a core notation or development method used in a particular setting, circumstance and developer experience should dictate the notations and development processes which are to be used for each task.

5. *Examples*

We hope to attempt a case study–perhaps one large, or several small examples–of how to use heterogeneous specifications in development. We plan to pay particular attention to the rôle that refinement plays in developing the heterogeneous specification. Three fairly simple examples are presented in Appendix B.

[3]Of course, predicated on other design constraints.

3 Related Work

Our notion of heterogeneous specification might have first arisen with the work of Hehner and Malton [HeMa88] on comparative semantics, and Dijkstra [Dijk93] and Hoare's [Hoar94] work on theory unification. In turn, these works have their roots with *multiparadigm languages* [Hail86], and *wide-spectrum languages* (e.g., see [Part90]). The terms *compositional specification* and *multiparadigm specification* (both of which are closely related to our *heterogeneous specification*) received recent notice with the work of Wing [Wing90] and Zave and Jackson [ZaJa93]. Abadi and Lamport's [AbLa93] transition-axiom method offers an approach similar to Zave and Jackson's–in that both offer a common semantics that can serve as a framework for many different notations–but the former's efforts are focused on concurrent systems, and are not intended to facilitate multiparadigm specification. Astesiano and Cerioli [AsCe93] describe an initial attempt at providing a foundation for multiparadigm specification.

Related to the compositional or heterogeneous specification notion is the concept of *method integration*–combining two or more methods to form a new, hopefully more useful technique. [SFD92] contains a good overview of (informal) integrations of Structured Analysis with VDM, Z, and other formal notations. [Kron93] is a general overview of method integration.

Zave and Jackson (whose work is closest in spirit to our own) take something of a different approach to specification composition than we do: while we are concerned with semiformal specifications and the effect of compositions on development methods, they are focused more on semantic definitions and consistency. Their work is also typically intended for use with a particular class of problems; our approach is apparently more general. One of the main distinctions between their work and ours is that they translate all notations into a new (and very general) notation, whereas we recognize that it is very likely that any heterogeneous specification will make use of only a few notations–for reasons of manageability. Under this premise, we can take one of three courses of action:

1. Translate all partial specifications in the composition into the most expressive–though not necessarily the easiest to use–notation.

2. Translate all partial specifications in the composition into the notation in which we want to carry out refinement and development, and therefore restrict our use of notation specifically to the translatable elements in each formalism.

3. Develop theorems and rules that allow us to carry out refinements (both formal and semiformal) over compositions.

Our work will focus on 2. and 3.

Zave and Jackson offer a further piece of motivation for considering heterogeneous specifications: such notions make it possible to use *much* simpler specification languages than what are now considered to be state of the art. There seems to be two reasons for this: (i) if we can compose a number of languages freely, there is no need to extend a language with features that another has; and (ii) a heterogeneous framework subsumes and can replace features–such as composition operators–found in many languages. For these and other reasons, we consider it worthwhile to examine heterogeneous specifications.

4 Conclusions

We have proposed a course of research examining heterogeneous specifications and their use in software development. We have identified several important problems that should be discussed, and have suggested some general approaches for attempting to solve them. We have also described several of the fundamental issues associated with heterogeneous specifications in general. Finally, we have provided (in an Appendix) initial research results related to formal and semiformal heterogeneous specifications, and (in a separate Appendix) examples of heterogeneous specification and development. Much work remains to be done, concentrating on expanding the results presented herein, extending the results to further semiformal domains, and integrating the results into the software development processes associated with the notations. Work has begun on studying these topics, especially with respect to the semiformal notations, such as those used in JSD and Structured Analysis. Hopefully, our results will show that heterogeneous specifications can be used productively in constructing programs.

A Initial Research Results

In this Appendix, we briefly recount some of the initial research results we have obtained. Space prevents us from including all the details and, indeed, all the results we have determined. Instead, we summarize a few of the most interesting of the preliminary findings, and leave the remainder for another time[4].

In order to avoid complications, we assume throughout this appendix that all notations use the unprimed-primed convention of Z and predicative programming to represent initial and final variables.

A.1 Comparative Semantics

Here, we describe some of our initial findings with respect to comparing the semantics of several formalisms. In other words, we show how a few formal notations–Z, VDM, the refinement calculus, weakest preconditions, and the predicative notation–are related. This is the first step towards allowing these formal notations to be used in composition.

Our research plan includes further examination and development of this material. We also plan to extend the comparative semantics to other domains, including both formal and semiformal notations.

A.1.1 Predicates and specification statements

Let **frame** $w \bullet P$ be a predicate specification (as in [Hehn93]) not involving time, i.e., there will be no references to the time variables t and t' in P. We require that the programmer specify the frame w.

The specification **frame** $w \bullet P$ is translated to the specification statement

$$w : [\, \neg\forall w' \bullet P, P \,]$$

under an appropriate syntax-directed translation. The translation is approximate since P does not talk about time while its specification statement translation does (in a restricted sense, although

[4]Say, a thesis proposal. . .

for every P there is a refinement calculus translation). The precondition in the translation is $true$ for all prestates from which P must terminate.

> *ASIDE.* In order to complete our translations with these two notations, we must also deal with predicate specifications mentioning time. While we have determined such a translation, we omit it here. We also plan to examine more thoroughly a simpler version of the translation described above, with which we lose our ability to talk about nontermination but acquire less complicated manipulation rules. *END OF ASIDE.*

The reverse translation is easy to describe. Providing that the specification statement is not angelic[5], the specification statement $w : [\, pre, post \,]$ is almost the same as:

$$\textbf{frame } w \bullet (pre \Rightarrow post).$$

The "almost" is due to the fact that the predicative notation cannot be used to write angelic specifications[6], while the specification statements have this capability (demonic specifications are permissible in both). Furthermore, we lose a bit of information in the translation and are no longer able to talk about time in the predicate formulation. This can be eliminated by transforming to predicate specifications that include time descriptions, or by including a boolean variable in the translation used to indicate termination, or lack thereof.

A.1.2 Refinement calculus and weakest preconditions

The mapping from specification statements to weakest precondition predicate transformers is well-known: Morgan gives a weakest precondition definition of specification statements in [Morg94]. The reverse transformation is somewhat more complex and is derived from [HeMa88]. We simply present the translations, and direct the reader to the references for further information and examples ($\sim$ should be read as "is translated to").

$$\begin{array}{rcl} wp(w : [\, pre, post \,], R') & \hat{=} & pre \wedge (\forall w' \bullet post \Rightarrow R'), \\ wp(S, R') & \sim & \alpha : [\, wp(S, true), (\neg wp(S, \alpha' \neq \alpha_0))[\alpha'/\alpha_0]], \end{array}$$

where α is a frame of variables determined by the programmer, with α_0 not in α. (For the calculation of wp in the postcondition, the α_0 are the variables.) Recall that $wp(S, true)$ is the weakest precondition for the existence of a time bound.

> *ASIDE.* It is also possible to translate between predicative programming specifications and weakest preconditions. A predicate specification S can be mapped to the weakest precondition
>
> $$wp(S, R') = \forall \sigma' \bullet (S \Rightarrow R').$$
>
> Given a weakest precondition $wp(S, R')$ over state σ, it can be translated to the predicate
>
> $$(\neg wp(S, \sigma' \neq \sigma_0))[\sigma'/\sigma_0],$$
>
> where σ_0 is not in σ. This latter translation, unfortunately, is not one-to-one: abort and havoc are both mapped to $\top$, while any angelic specification is mapped to its demonic dual. *END OF ASIDE.*

[5]The basic angelic specification is the *angelic update* $\{R'\}$ (for predicates R'), which satisfies $wp(\{R'\}, Q) = \exists \sigma' \bullet R' \wedge Q'$ for all predicates Q'.

[6]Interestingly, the mapping from specification statements to predicates transforms an angelic assignment into a *demonic* assignment. Thus, we can remove our restriction on not translating angelic specifications, providing that we do not mind our translation being given a demonic interpretation.

A.1.3 Refinement calculus and VDM

We now describe a relationship between the refinement calculus and VDM. We start by assuming that there is a syntax-directed translation between notations; such a mapping is not difficult to construct[7]. The translation from VDM pre- and postconditions to specification statements is then immediately obvious:

$$\mathsf{pre}\ P,\ \ \mathsf{post}\ Q \sim \alpha : [\ P, Q\ ],$$

for a frame α specified by the programmer. In fact, under syntax, this mapping is almost invertible; the postcondition Q in the mapping from the refinement calculus to VDM may contain appropriate frame axioms (although this can be subsumed by using the VDM equivalent of frames).

A.1.4 Z and the refinement calculus

The final set of translations we describe are between Z and the refinement calculus. One direction of the translation has been noted in the past [King90]; we merely summarize these results here.

The mapping from a Z schema to a refinement calculus specification statement is quite straightforward and was described in [King90]. There is first a syntax-directed translation from Z schemas to an intermediate form (mapping primed variables to unprimed variables, etc.). Then, the Z schema

$$Op ::= [\ \Delta S; i? : I; o! : O \mid pred\ ]$$

can be mapped into the following specification statement:

$$w : [\ (\exists w' : T \mid inv \bullet pred)[w'/w], pred\ ],$$

where 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 outputs. See [King90] for more details.

The reverse mapping (from the refinement calculus to Z) is less straightforward, since it has to be left to the programmer to decide how to decorate variables (as input or output) in the resulting schema, which variables to declare in the current schema, etcetera. However, supposing that this can be done, the specification statement $w : [\ pre, post\ ]$ can be translated to the following schema (ignoring input and output decorations):

$$S ::= [\ \Xi\rho; \Delta w \mid pre \wedge post\ ].$$

In the schema, ρ is the set of all scoped variables *not* in the frame. Since the specification statement does not contain declarations, further declarations are not needed in the Z equivalent (although the variables will have to be declared eventually). This mapping holds under an appropriate syntax-directed translation. Notice that Z cannot express magic, i.e., Z specifications must be feasible (unless we extend schemas to include *pre/post* pairs, which we will consider another time).

> *ASIDE.* With the translations all defined, we might like to know something about their mutual consistency, i.e., as a set, do the translations provide consistent results when applied to the same specification? While we do not pursue this question here, we plan to examine it in detail another time. *END OF ASIDE.*

[7]We do not use the logic of partial functions basis for VDM in these translations.

A.2 Heterogeneous Specifications

We have shown how to translate–as much as possible–between the five formalisms shown in Figure 1.

$$\begin{tikzcd}
 & \text{Predicative programming} \arrow[d, leftrightarrow] & \\
\text{VDM} \arrow[r, leftrightarrow] & w : [\, pre, post \,] \arrow[r, leftrightarrow] \arrow[d, leftrightarrow] & wp(S, R) \\
 & \text{Z} &
\end{tikzcd}$$

Figure 1: Notation translations

Thus, for (almost) any specification S in one notation, we can express S in (almost) any of the other notations we have mentioned. We now sketch how this facility allows us to use *heterogeneous specifications* in programming.

In each of the notations mentioned, there is a collection of specification combinators–operators for combining specifications in various ways. For example, *sequential composition*, *disjunction* and *conjunction* are all widely-used operators, though not present, nor given the same interpretation in each of the formalisms mentioned.

Since we now have a way of expressing specifications from one notation in another notation, it becomes possible to extend the domain of *defined operands* of the combinators to include specifications written in different notations.

For example, let P be a predicative specification. Then the specification

$$P; w : [\, pre, post \,] \tag{1}$$

can be given the following predicate semantics:

$$\exists \sigma'' \bullet P^{\sigma'}_{\sigma''} \wedge (\textbf{frame } w \bullet (pre \Rightarrow post))^{\sigma}_{\sigma''},$$

where the sub- and superscripts indicate that the subscript variables are to be substituted for the superscript variables. We could also give a semantics to the dependent composition by expressing P as a specification statement (although we will not obtain exactly the same interpretation, in general). This would be accomplished by using the following result.

Theorem 1 *Using ; to denote sequential (relational) composition, the following relation holds:*

$$w : [\, pre_0, post_0 \,]; w : [\, pre_1, post_1 \,] = w : [\, pre_0 \wedge \forall w' \bullet (post_0 \Rightarrow pre_1[w/w']), post_0; post_1 \,].$$

This result is quite useful. We would use it to investigate when the predicative and refinement calculus semantic bases provide different or identical interpretations.

The fact that we have described translations between notations means that we can write formal specifications as a composition of partial specifications in arbitrary (formal) notations. If one particular notation (say, Z) is useful in specifying one part of a problem, but predicative programming is useful in another part, it is considered to be perfectly acceptable to use both notations in composition. Of course, such compositions may not always prove to be beneficial; such judgments will depend upon both functional and nonfunctional constraints.

Such compositions may potentially make a design calculus, like predicative programming, even more usable in programming-in-the-large situations, since they can be combined with formal specification notations like Z which are very useful when applied in a subsidiary support rôle. Heterogeneous specifications allow us to restrict how we apply a calculus to solving a problem. This is crucial when we are using such a facility for programming in the large.

We now present some useful theorems describing properties of the refinement relations $\Leftarrow$ (of the predicative methodology) and $\sqsubseteq$ (of the refinement calculus) when applied to certain heterogeneous specifications. We begin by giving a formal definition for refinement in both cases, using a *wp* basis for the refinement calculus. In the following theorems, let P and Q be predicate specifications (assuming that both specifications start with **frame**) and S and T be specification statements.

Definition 2 *A specification P is refined by a specification Q if*

$$\forall \sigma, \sigma' \bullet (P \Leftarrow Q).$$

Definition 3 *A specification S is refined by a specification T (written $S \sqsubseteq T$) if*

$$\forall R' \bullet wp(S, R') \Rightarrow wp(T, R').$$

Theorem 2 *If $S \sqsubseteq T$ then $P \wedge S \sqsubseteq P \wedge T$.*

Theorem 3 *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 $P \vee S \sqsubseteq P \vee T$.*

Theorem 4 *If $S \sqsubseteq T$ then $P; S \sqsubseteq P; T$.*

It turns out that the theorems we have shown are useful special cases of two general theorems on refinement over heterogeneous specifications. In the following, let P and Q be in one notation (say, N), with $\hat{P}$ and $\hat{Q}$ their translations into a second notation, say $\hat{N}$. Assume that the translations of P and Q into $\hat{P}$ and $\hat{Q}$ are information preserving, i.e., P and $\hat{P}$ have equivalent definitions in a formalism at least as expressive as N and $\hat{N}$. Let $\circ$ be a combinator of $\hat{N}$ over which partwise refinement can take place, and let S be a specification in any notation whatsoever. Finally, let $\subseteq$ be a reflexive and transitive refinement relation. The proofs of these theorems follow directly from the assumption of monotonicity, and the definitions.

Theorem 5 *If $P \subseteq Q$ and $\subseteq$ is monotonic over $\circ$, then $P \circ S \subseteq Q \circ S$.*

Theorem 6 *Suppose $\subseteq$ is not monotonic over $\circ$, but there exists a refinement relation $\sqsubset$ in $\hat{N}$ which is monotonic over $\circ$. If $\hat{P} \sqsubset \hat{Q}$ then $P \circ S \sqsubset Q \circ S$.*

What these theorems tell us is that the monotonicity and transitivity of $\Leftarrow$ and $\sqsubseteq$ are *almost* completely preserved and we can apply the appropriate refinement relation to the appropriate notation over a composition without a corresponding change of notation, in most cases. Even when we are required to change notation, the transformation is (almost) completely syntactic (i.e., from $w : [\, pre, post \,]$ to **frame** $w \bullet (pre \Rightarrow post)$) and in doing so, we move to a notation with a simpler notion of specification refinement (i.e., boolean implication). From a practical viewpoint–and in terms of refinement–the change in notation is justifiable.

We previously mentioned the notion of a *heterogeneous development*, where refinement steps in a derivation can be performed in different notations. In other words, at each stage in a development, we are able to use whatever notations *or* refinement relations we so choose. A preliminary result along this line is the following one.

Theorem 7 *If* $pre \wedge \forall \sigma' \bullet (post \Rightarrow P)$ *then* **frame** $w \bullet P \Leftarrow w : [\, pre, post \,]$ *and* **frame** $w \bullet P \sqsubseteq w : [\, pre, post \,]$.

We envision expanding this concept to other formal and semiformal notations, so as to obtain more freedom in developing programs heterogeneously.

Another useful result regarding heterogeneous development is the following.

Theorem 8 *If* $S \sqsubseteq T$ *then* $\forall \sigma, \sigma' \bullet S \Leftarrow T$.

In Theorem 8, S and T are specification statements, so in the consequent there is an implicit translation into predicates. The immediate impact of Theorem 8 can be seen in its effect on Theorem 3: $P \vee S \sqsubseteq P \vee T$ also holds if $S \sqsubseteq T$.

We now briefly consider an interesting special case of formal heterogeneous specifications in order to explore some of their features. To be more specific, we consider the problem of adding so-called *specification constructors* to the refinement calculus.

This problem was examined by Ward [Ward93], who considered adding generalizations of Z schema conjunction and disjunction operators to the refinement calculus. The rationale for considering this extension was to facilitate the construction of large specifications from smaller ones. A problem with Ward's work is that the specification combinators of Z are not monotonic with respect to the refinement relation $\sqsubseteq$ (we can easily prove this using heterogeneous specifications). We take a slightly different approach–that of heterogeneous specifications–and (mostly) preserve the property of monotonicity.

Our goal here is to give a semantics to expressions such as:

$$w : [\, pre, post \,] \vee x : [\, pre', post' \,],$$
$$w : [\, pre, post \,] \wedge x : [\, pre', post' \,].$$

We do this by interpreting $\vee$ and $\wedge$ as logical disjunction and conjunction, respectively, and transform the specification statements into their equivalent predicate form. We perform manipulations and then reverse the transformation. By doing so, we obtain some useful results. Before presenting them, we make a definition, taken from [Ward93].

Definition 4 *A specification statement is* **frame complete** *if its frame consists of all variables mentioned in the pre- and postcondition.*

It is easy to make a specification statement frame complete by using the law *Expand Frame* from [Morg94]. With this definition in mind, we now present some theorems (omitting all proofs). In the following, let $S = w : [\, pre_w, post_w \,]$ and $T = x : [\, pre_x, post_x \,]$.

Theorem 9 *If neither* S *nor* T *are angelic, then*

$$S \wedge T = w, x : [\, (pre_w \wedge \neg \forall w' \bullet post_w) \vee (pre_x \wedge \neg \forall x' \bullet post_x), (pre_w \Rightarrow post_w) \wedge (pre_x \Rightarrow post_x) \,].$$

Theorem 10 *If neither S nor T are angelic, and $w = x$ then*

$$S \vee T = w : [\, (pre_w \wedge pre_x) \wedge \neg\forall w' \bullet (post_w \vee post_x), (pre_w \Rightarrow post_w) \vee (pre_x \Rightarrow post_x) \,].$$

In Theorem 10, we are assuming that the frame describes exactly the variables of interest.

Theorem 11 *If S is frame complete and not angelic then*

$$\neg S = w : [\, pre \Rightarrow \exists w' \bullet post, pre \wedge \neg post \,].$$

Further theorems and properties have been determined (especially regarding partwise refinement), and we plan to compare our results with those of [Ward93] to discover respective strengths and weaknesses.

> *ASIDE.* In work we do not present here we investigate a simpler (yet slightly less expressive) translation of predicates into specification statements. This translation leads to results different from those presented above. Part of our work will entail an evaluation of both translations. Our investigations to date suggest that the loss of expressiveness is not significant and we hope to present evidence to support this hypothesis. *END OF ASIDE.*

Space prevents us from including all the details (and proofs of our results), but part of our general plan includes expanding our results to further formal domains, and generalizing to allow compositions with semiformal specifications. We have already begun to examine such issues, as we now mention.

A.3 Semiformal Specifications

We (*very* briefly) present a summary of a few of our findings with respect to work on: (a) giving a formal semantics to various semiformal notations; and (b) describing the changes that the ability to use heterogeneous specifications place on any associated development process. Our descriptions are necessarily brief and omit all of the formal details.

A.3.1 Structured Analysis/Design-related Results

- Data flow diagrams can be formalized in Z [SFD92] or in the predicative notation (placing an interpretation on the diagram as operations on a state). This can typically be done in combination with control flow diagrams and the data dictionary. In fact, we have found that a useful extension to the data dictionary concept in such integrations is to include *data transformation* details. These formalization details will be helpful when we consider compositions of data flow diagrams with formal specifications, and in integrating the formal notation into SA/SD.

- It is also possible to extract a data flow diagram (with a particular interpretation) from certain Z specifications. We have also considered extending this notion to extraction of higraph-based specifications, or dependency trees based on Z schema interconnections, so as to reduce the information loss in the process.

- Entity-relationship diagrams can be expressed in Z [SFD92] or in the predicative notation. Particular care has to be taken with ensuring that the various relationships (many-to-one, one-to-one, etc.) and "opting in and out" of a relationship are properly described.

- The development process for Structured Analysis (based on the aforementioned diagram notations) needs to be changed, to include mention of: (a) proofs of data flow diagram refinement; (b) inclusion of formal specifications in the data flow diagram; and (c) the transition from a standard or heterogeneous data flow diagram to a formal specification or another (formal or semiformal) diagram paradigm.

- We have commenced preliminary study on heterogeneous specification integration into the SADT methodology, concentrating on: diagram alterations, changes to the modelling process, and augmentation of the iterative review process to include new notations. More work remains to be done, including considering how heterogeneous specifications should be presented to a technical review committee (e.g., possibly as SADT box abstractions).

A.3.2 JSD results

- Process connections can be formalized. In doing so, we have chosen to use predicates, but this requires a notation extension to add a program counter to the semantics (we are investigating if this can be avoided). The details of this remain to be completely worked out; furthermore, we have to evaluate if it is worthwhile to make such a formalization.

- Jackson diagrams and structure text are straightforward to formalize (although some of the loop-constructs result in complex specifications).

- Work remains to be done on exploring the changes to JSD that occur after adding heterogeneous formal specifications at both the visual and textual levels, especially with respect to how the interface with the "real world" (as described in [Jack81]) must be dealt with.

A.3.3 Programming languages and PDL results

We have commenced the determination of a formal semantics for various programming language subsets. Our goal is to allow the use of subsets of certain real programming languages in composition with formal specifications. Not only does this allow us to restrict the use of a program design calculus, but it also permits us to carry out refinements to the level of (real) programming language code. In particular, we plan to consider C and C++. For C++, we expect to come up with predicate specifications for (some of) the object-oriented constructs in the language. Another approach would be to integrate algebraic types (e.g., as in CIP-L) into a predicate-based formalism. Of course, the extent to which we consider such compositions and integrations will be dependent on the utility of such descriptions.

We have also developed a formal semantics for a typical program description language, and hope to be able to use this in composition with formal specifications. Such compositions will prove helpful when examining integrations of JSD and formal specification-based techniques.

A.3.4 Other results

We briefly summarize some of the other general research results we have (partially or completely) obtained.

- We have commenced work on combining JSP and Structured Analysis through *heterogeneous graphical specifications*. Work remains to be done on extensions to the methodological aspects of the resulting composite technique.

- We have been attempting to understand how to integrate *formalization by parts* into the framework, for use in writing and composing specifications which involve large diagrams (or large formal texts).

- We have determined how to extend a predicative *parallel composition* operator to the refinement calculus and Z. We will need to evaluate the utility of the operator when compared with, for instance, *action systems* [Back90].

- We are examining how to use a topovisual formalism (inspired by *higraphs*) to depict both large and small formal specifications. We need to understand how to describe text-based specifications in a visual manner in order to use them in composition with other visual notations, such as data flow diagrams. Our initial work on "higraphs" suggests that they will prove to be convenient for heterogeneous use, and for formalizing many of the structural aspects of large formal specifications. Our notation allows us to depict data and control flow (along with more abstract relationships), as well as structure and hierarchy, and permits us to abstract away as much detail in the specification as is necessary. See [Paig95] for more details.

- We are studying how to use heterogeneous specifications in nonlinear development processes (e.g., see [PHG91]). In such situations, it may be easier to "reverse engineer" specifications from implementations, due to the heterogeneous nature of the final specification. This might prove to be useful in allowing designers more flexibility in developing software.

This is only a brief summary of some of the results we have determined; many of the details (especially regarding Part 4 described in Section 2) and more useful examples remain to be worked out. Extension of the approach to SADT, SSADM, and other semiformal notations and methods must occur, too, although we should add that we have many of the preliminary details so far.

B Examples

In this appendix, we consider three examples presenting the use of heterogeneous specifications in development. The first example–which focuses on integrating predicate specifications with a structured method–demonstrates an instance of a semiformal development, where formal specifications are used only when required or convenient for the problem at hand. Our second example, a combination of predicates and specification statements, presents a precise formal heterogeneous specification and its development into code. This might be termed an example of a *formal heterogeneous development.* The final example describes how Jackson process diagrams and formal specifications might be combined, in the framework of an applied example from chemical systems modelling.

The reader should be aware that these "case studies" are only small examples (as they must be for presentation in this forum). They are not the only (nor necessarily the best!) approach in which heterogeneous specifications might be used to solve the problems. The manner in which such specifications should be used for a particular problem will depend heavily upon the developers, the nonfunctional constraints under which they must operate, and their interpretation of the problem. These examples are presented in order to demonstrate how heterogeneous specifications might be used in development, and to describe particular instances of some interesting–and nontrivial–specifications.

B.1 First-come first-served simulation

The problem we wish to solve is that of constructing a simulator for a first-come first-served scheduler. We wish to develop a solution for this problem using heterogeneous specifications, guided by the Structured Analysis and Design development method [DeMa79]. Our heterogeneous specifications will be based on predicates, combined with structure text, data flow diagrams, structure charts, and programming language code. Compositions between these different notations are justified by the framework (and translations) developed and described in Appendix A. We could use this basis to–completely formally–develop an implementation, but we do not follow this path here.

The initial requirements and problem explanation are as follows. We will see that, as is frequently the case with such descriptions, the presentation of what is required from the solution is rather poorly stated.

> Write a program to generate data and to simulate a first-come first-served short-term scheduler. The program will have two parts: the first will **generate** a set of data to be loaded by the scheduler. The second part will **simulate** the operation of the FCFS scheduler on the data. The program should be implemented in such a way so that the function of the generator is as independent as possible from the simulator (so that the generator part can be moved into a separate program if necessary). Recall that a short-term scheduler consists of a **queue**, holding waiting processes, and **programs** that accept arriving processes and send processes to the CPU for execution at the appropriate time.
>
> The generator part of our program constructs data to be loaded by the scheduler. It generates two vectors of data, one holding **random CPU burst lengths**, and the other holding **random arrival times of processes**. The CPU bursts should be generated so that 80% of the bursts are uniformly distributed between 0.1 and 1.0, and the remaining 20% are uniformly distributed between 1.0 and 10.0. The arrival times of the processes must have a Poisson distribution. The time interval between two contiguous arrivals is given by:
>
> $$-\lambda \log_e x$$
>
> where x is a random variable uniformly distributed between 0 and 1. The parameter λ is chosen so that the ready queue is steady, i.e., so that the rate of arrivals to the queue and the rate of dispatching processes from the queue are approximately the same. A value of $\lambda = 1.3$ should suffice for this FCFS simulation (although if a different scheduling algorithm, say, round-robin, were used, the constant would clearly be different). The vectors of data should be stored in a file, `data`, in order to meet our independence requirement.
>
> The second part of the program is the FCFS simulator. The vectors from `data` are loaded, and the CPU bursts are partitioned into ten groups. Group 0 contains bursts ranging in length from 0.1 to 1.0. Group 1 contains bursts from 1.0 to $2.0, \ldots,$ and Group 9 contains bursts from 9.0 to 10. Average wait times must be calculated for each group. The number of processes in each group, the total wait time, and the average wait time should be output from the simulator for the generated data. Waiting processes will be stored in the **ready queue**.
>
> For the most efficiency, the ready queue for the scheduler should be implemented by a circular queue of length `MQL` (a reasonable value for `MQL` is 100). When the simulation begins (i.e., at $time = 0$), the first CPU burst starts in the CPU. The next 20 bursts must already be in the ready queue when this starts (this is the **initialization** of the ready queue before commencement). The scheduler adds processes to the ready queue if they arrive before the currently-active process in the CPU finishes. Upon completion, the current process is dequeued, and the process at the head of the queue then begins execution. This process continues until either the queue destabilizes, or all the CPU bursts have been served.
>
> The statistics (average wait time, etc.), are best calculated when the scheduler dispatches a process from the ready queue to the CPU. Of course, you will have to keep track of the arrival times of all processes in the ready queue (since the wait time of a process is $T - arrivaltime$).

The problem is described in a manner which is quite hard to understand. For this reason, we initially choose to use a structured approach to development. The general scheme followed in such an approach is to first present a data view of a solution to the problem. This is refined into a modular presentation (in the design phase). Then, structure text is generated, and finally, an implementation (which we shall describe in the programming language C) is constructed. See [DeMa79] for more detail.

The first stage in our development is to generate a data flow diagram (DFD). In this example, our diagram will be heterogeneous, since we feel most comfortable constructing a solution using multiple notations. Our first (context-level) DFD appears as shown in Figure 2.

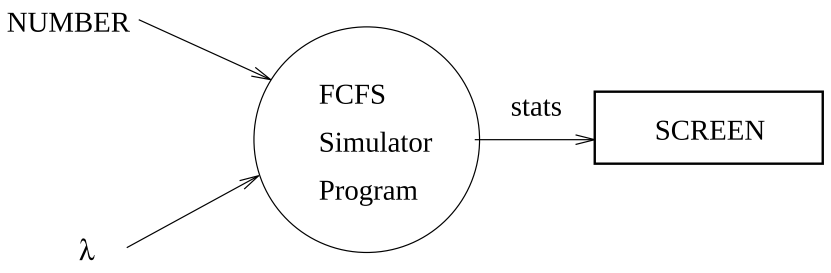

Figure 2: Level 0 context diagram

Notice that this diagram is identical to one that could be produced under a standard structured analysis approach. Note as well that if we specified the DFD bubble formally (which is certainly possible, though likely not desirable at this stage), we would be able to have a completely rigorous development of a solution *guided* by the Structured Analysis and Design process. However, the semiformal development process offered by the structured approach may very well be more convenient to use in many development situations (e.g., when building large programs) than a completely formal one. We follow the semiformal and heterogeneous development path in this example, and leave the formal approach for another time.

We now elucidate the DFD by splitting up the context diagram into its two constituent parts: the data generator, and the simulator. We specify the generator part using a (visual) predicate specification, since this is how we best understand the problem. In the DFD refinement, we add a file (an external resource) named `data`. The reason for performing this introduction is so that later, once we have reached code level, it will be possible to separate (both syntactically and in terms of execution) the generator and simulator code portions. This elucidation is shown in Figure 3.

In Figure 3, *random* is a function generating a random *real* uniformly distributed between 0 and 1 (it can be implemented, for example, by the C function `drand48()`). As well, MQL is the maximum length of the simulator's circular queue, and we use ; to denote relational composition of predicates.

Notice that in the visual predicate specification of generator, we have made use of several data objects. These should be added to our data dictionary (along with their formal types, where known) and can be used in specifying other parts of the solution (constrained, of course, by scope). Note as well that we are not restricted to using these precise data objects throughout the solution; we can data refine them away as necessary, providing that we make the appropriate changes to the data dictionary[8]. We could also use them–in certain cases, e.g., when data refinement is simply supertyping–in such a way so as to take advantage of *foreknowledge* regarding data transformation:

[8]This process might be called *data dictionary refinement.*

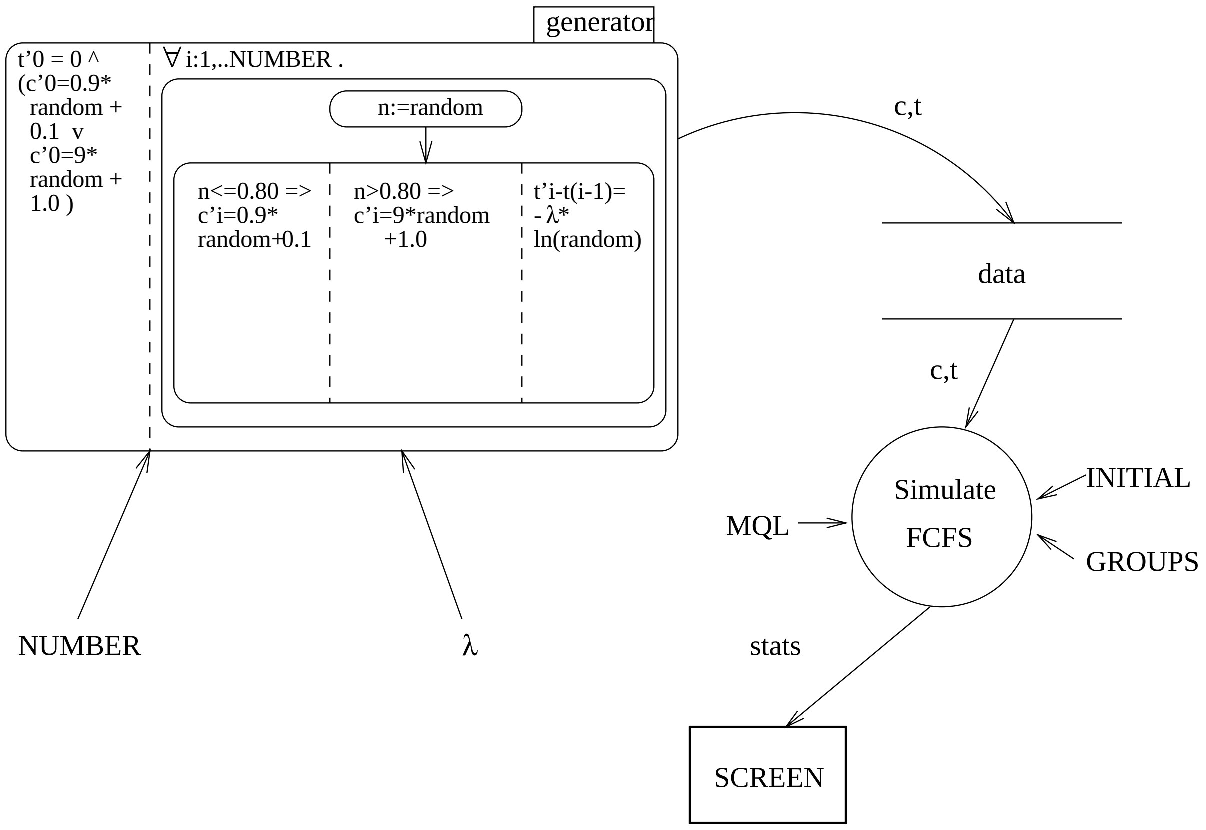
generator
t'0 = 0 ^ (c'0=0.9* random + 0.1 v c'0=9* random + 1.0)
∀ i:1,..NUMBER .
n:=random
n<=0.80 => c'i=0.9* random+0.1
n>0.80 => c'i=9*random +1.0
t'i-t(i-1)= - λ* ln(random)
c,t
data
c,t
Simulate FCFS
MQL
INITIAL
GROUPS
NUMBER
λ
stats
SCREEN

Figure 3: Level 1 Diagram

for example, if we know that a data object is to be reified to an object of a particular type (and this reification is suitably documented), we can use the object as if it had the reified type. We make the following data definitions now, and use them throughout the specification as needed.

$$\begin{aligned}
MQL, INITIAL, NUMBER &: nat \\
GROUPS &= 10 \\
\lambda &: real \\
c, t &: [NUMBER * real] \\
cell &= \text{“}burstlength\text{”} \rightarrow real \mid \text{“}arrivaltime\text{”} \rightarrow real \\
ready &: [MQL * cell] \\
boundary, head, tail, length &: nat \\
bursts, arrivals &: [NUMBER * real]
\end{aligned}$$

The next step is to elucidate the *SIMULATE FCFS* bubble. We do this using a heterogeneous data flow diagram, as shown in Figure 4 (note that Figure 4 shows *only* the refinement of the relevant bubble, and *not* the entire flow diagram).

We have not specified predicates Q and R in Figure 4, to keep the complexity down[9]. The precise definition of Q is as follows:

$$Q \quad = \quad bursts'(i) = ci \wedge arrivals'(i) = ti.$$

The reason why we have introduced arrays *bursts* and *arrivals* in the specification of Q is so that when implementing the simulator code portion we can eliminate array variables c and t, which we see are not needed due to the presence of the file.

To specify the predicate R, we note that it is a description of the FCFS simulator. Since it is often best to use operational specifications to describe simulations, we make use of some program code in this case. Recall that the queue to hold processes will be implemented in a circular fashion, and that when the simulation begins there are already `INITIAL` processes in the ready queue.

```
time:=0; arriving:=INITIAL;
while (stable) do (
  updatecount;
  marr:=MAX j:arriving,..NUMBER• (time+boundary>arrivals(j));
  for i:=arriving;..marr do (length≠MQL) ⇒ enqueue(bursts(i),arrivals(i)-boundary);
  arriving'=marr ∧ current'=front;
  dequeue)
```

In the specification, `stable = length>0 ∧ length≠MQL`, the constant `boundary` is the arrival time of process $INITIAL - 1$, and the predicate `updatecount` updates the statistics for the simulation (we do not worry about its specification here, but it is trivial). The functions *front*, *enqueue* and *dequeue* can also be formally specified; we do so because we understand them well in such a format. *enqueue* is defined as follows:

$$\begin{aligned}
enqueue = \lambda a, b : real \bullet (head \neq (tail + 1 \bmod MQL)) \Rightarrow ready' = (tail; \text{“}burstlength\text{”}) \rightarrow a \mid \\
(tail; \text{“}arrivaltime\text{”}) \rightarrow b \mid ready \wedge tail' = (tail + 1) \bmod MQL \wedge length' = length + 1.
\end{aligned}$$

[9] We certainly could write them visually, if we so desired.

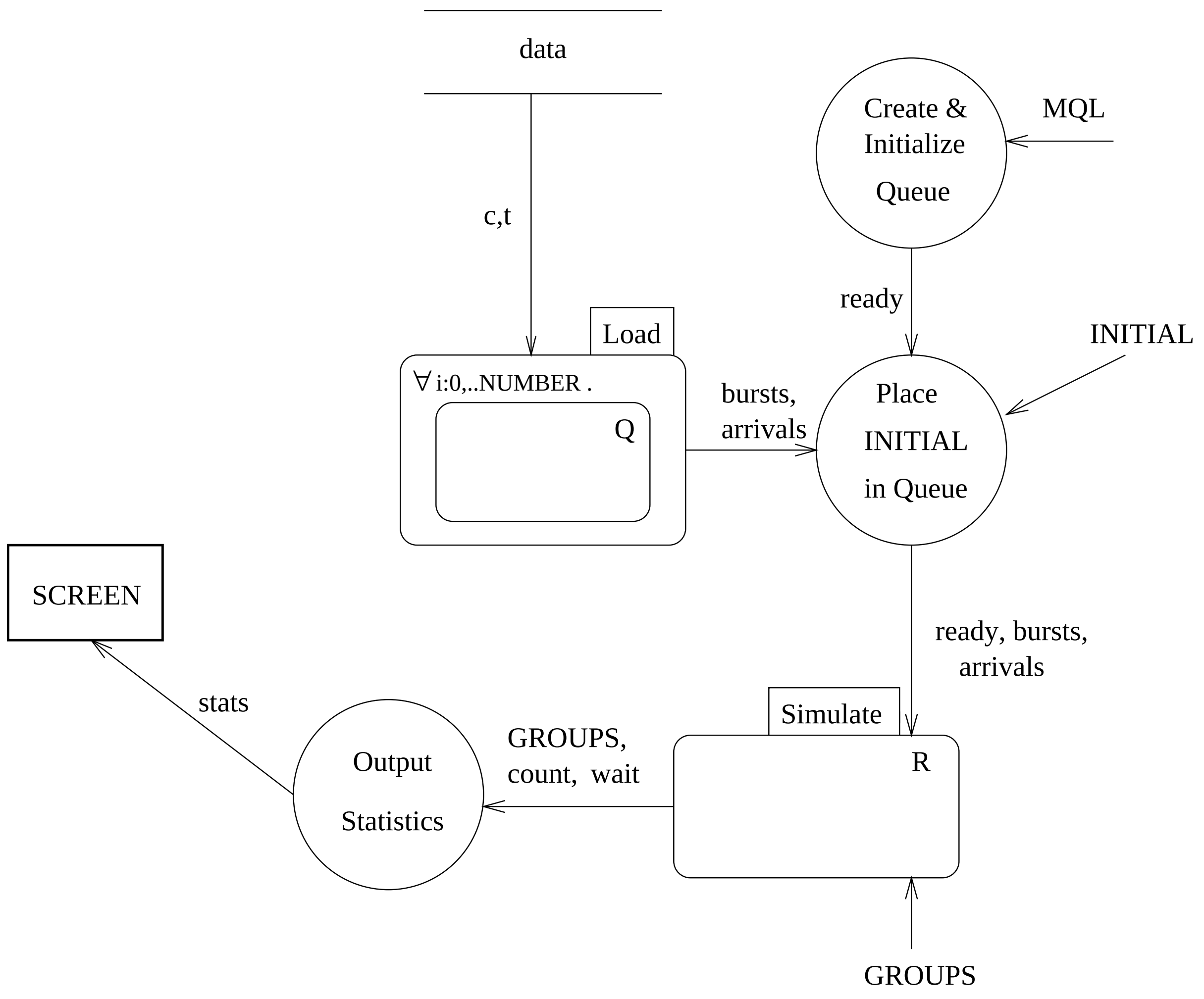

Figure 4: Refinement of *SIMULATE FCFS*

The function *dequeue* is much simpler:

$$dequeue = (head \neq tail) \Rightarrow head' = (head + 1) \bmod MQL \wedge length' = length - 1,$$

while *front* is the simplest of all:

$$front = ready(head).$$

We now feel that we have sufficiently specified the problem so as to be able to generate an implementation. (The structured approach suggests that we refine DFD bubbles until they are at a level from which pseudocode can be easily generated. The heterogeneous variant of the structured approach would extend this DFD refinement process to include the generation of *any* appropriate text-based specifications–both formal and semiformal). However, the next stage in standard structured analysis and design would be to describe a structure chart. Our heterogeneous structure chart would appear something like the one shown in Figure 5.

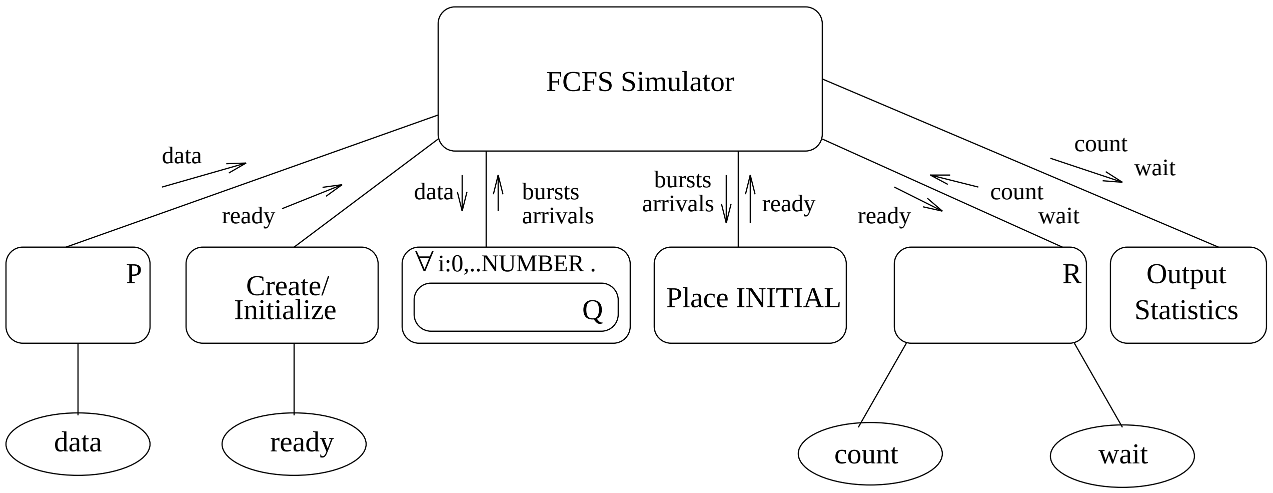

Figure 5: Heterogeneous structure chart

In Figure 5, P is the formal specification of the generator, and Q and R are as above. However, one might reasonably say that a solution to our problem is suitably determinable from the DFD[10]. Thus, the usefulness of the heterogeneous structure chart can be called into question in this situation. It is certainly possible to construct structure text (i.e., pseudocode) from the data flow diagram with little difficulty. In fact, if at this stage we decide to continue the development in a completely formal manner, we could formalize the DFD–or the structure chart–and proceed rigorously, after integrating P, Q, and R into the mix.

ASIDE: **Different Paths**

At this point in the development process–after the construction of a suitably low-level data flow diagram–there are several paths we might choose to follow:

1. Continue with the SA/SD-guided heterogeneous development method, where predicate parts will be included in composition with the standard structured notations.

[10]It never hurts to describe the system's modular structure using the structure chart.

2. Unformalize the predicate parts (say, to process specifications expressed as programs or pseudocode), and proceed with a standard SA/SD development.

3. Formalize the DFD (i.e., translate the structure of Figure 4 into a formal notation) and proceed with a completely rigorous development.

These approaches run the gamut of development methods, ranging from completely semiformal (Method 2.), to completely formal (Method 3.), with an arbitrary mixture of formal and semiformal development in between (Method 1.). While we will follow 1. in developing a solution here (since we believe that it best captures our understanding of the problem decomposition), we briefly consider aspects of 2. and 3. here, for the sake of exploring some of the alternatives that other developers may find preferable.

Unformalization

This development process step might arise when unforeseen or unfortunate nonfunctional constraints come in to play. One might envision a scenario in which, after having introduced formality (in some quantity and some manner) into a development, our nonfunctional constraints are altered in such a way so that we are unable to fully take advantage of the formal nature of aspects of our specification. For example, perhaps the deadline for our project has changed, so that we do not have the time to produce formal developments for the appropriate part of our system. Or, perhaps we have lost the services of some of our formal methods experts, and the personnel remaining on the project do not have the expertise necessary in order to read, write, and utilize the formal specifications at hand. What we need to do here is to *unformalize* relevant parts of the heterogeneous specification (or, from another perspective, *backtrack* through the steps of heterogeneous specification construction), and continue with development using whatever semiformal approaches there are at hand.

The unformalized[11] data flow diagram for Figure 4 might look something like what is shown in Figure 6 (note that there are many other unformalizations that may present more or less process detail than is shown in Figure 6):

This is an obvious (structural) unformalization of Figure 4, and is one that will let us continue, straightforwardly, with standard structured analysis and design (note that we may have to elucidate the unformalized process bubbles). However, in unformalizing the heterogeneous DFD, we have lost a lot of the information[12] that was present in Figure 4. It may be possible to save some of this information; there are two obvious ways in which we might do this:

- unformalize not only the structure of the heterogeneous specification but also the *text aspects* of the heterogeneous parts, and add this text to the (homogeneous) DFD as process specification parts (PSPECS).

  For example, from Figure 4, we might gather the following information for the `generator` part:

  process `generator`

  inputs: NUMBER, λ

  outputs: c, t

[11]This is something of an unfortunate term: relatively speaking, Figure 6 is *more* informal than Figure 4, but circumstance dictates whether or not the heterogeneous DFD should be considered formal in the first place.

[12]This is unavoidable: unformalization is the inverse of a process which is by its very nature information-adding.

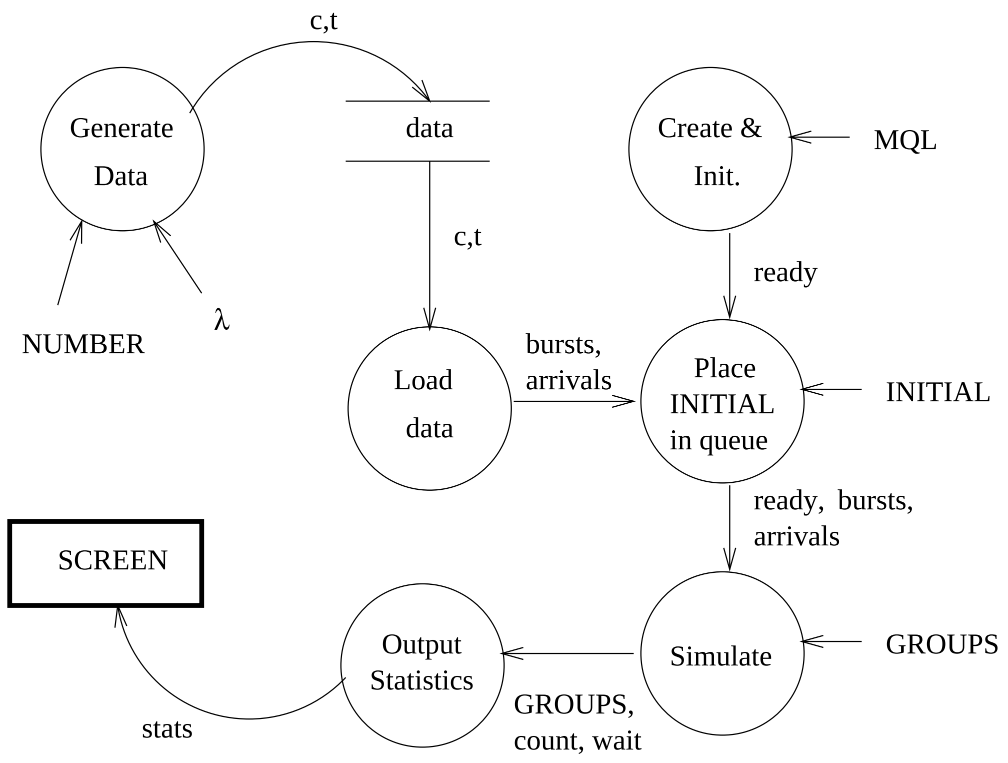

Figure 6: An unformalization of a heterogeneous specification

logic: Compute the values of arrays c and t at 0. Fill in the rest of the arrays with random numbers, with 80% of the burst times between 0.1 and 1, uniformly, and 20% uniformly between 1 and 10, and with the arrivals having a Poisson distribution.

Again, information is lost in the unformalization process, but the amount in question is much smaller providing we add the above details as a process specification.

- Unformalize the structure of the heterogeneous parts *recursively*, i.e., attempt to capture (some of) the structure of the formal specification parts as DFD process bubbles. Then, we can try and apply the above process (i.e., unformalization of text parts) to gather even more information semiformally.

  The result of such a process might be depicted as shown in Figure 7; the dotted lines encircle the recursively unformalized parts. Notice that in this diagram we do not depict any of the more algorithmic details, which can be captured in process specifications as described above.

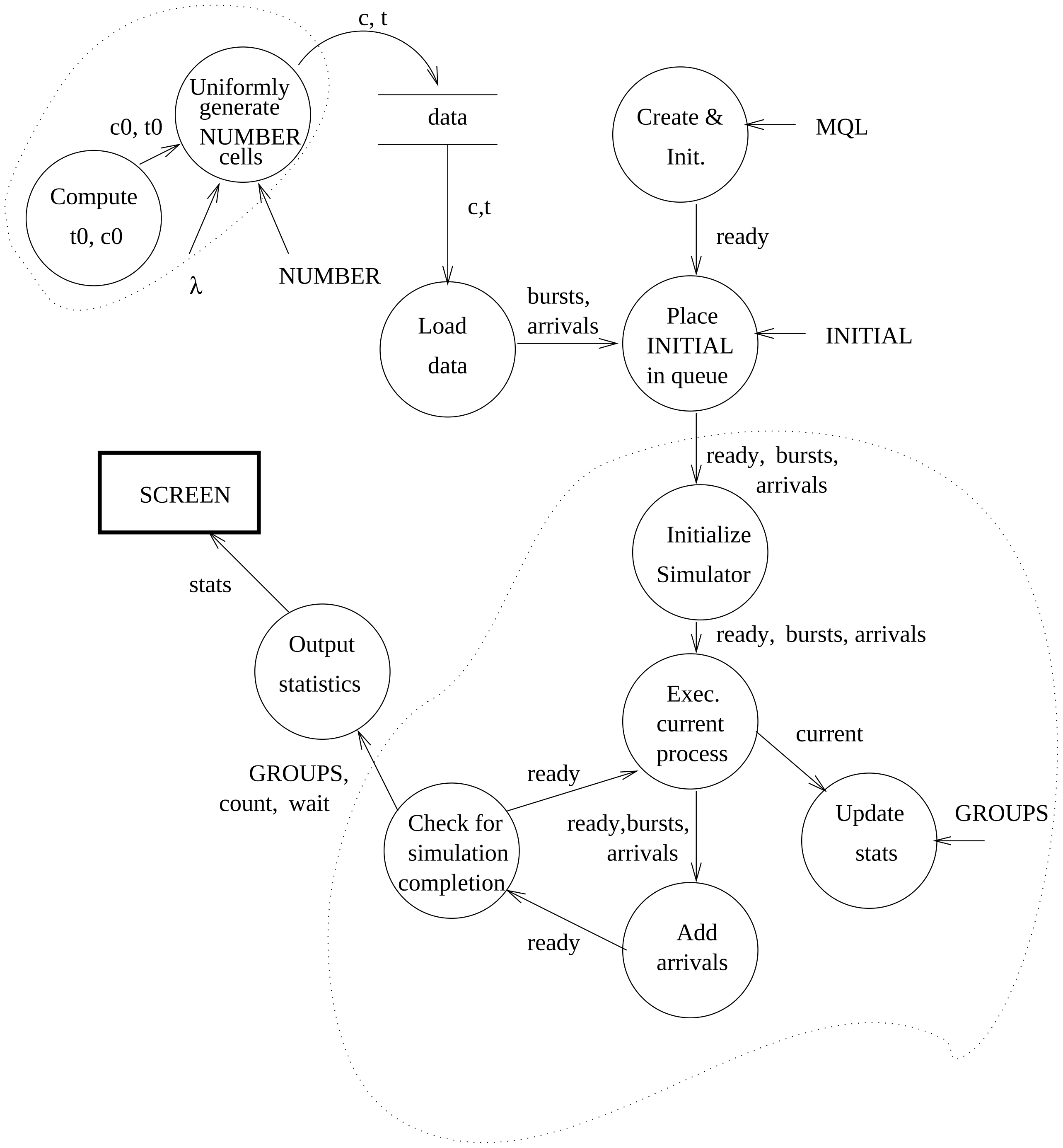

Figure 7: Recursively unformalized data flow diagram

Formalization

A second path we might choose to follow at this point in the development is a rigorous, formal one, in which precise specifications are constructed and are used–either in a subsidiary support or a central construction rôle–to develop a final implementation. We might view this choice of development process as being the *ideal*: in the best of all possible situations, we should like to be able to perform all development steps rigorously, via some initial formal specification. Of course, this will not be possible in most development situations, due to both functional and nonfunctional constraints. This is the main reason why we believe that development path 1. is the most realistic (for this situation) among the given alternatives. Even so, there may be situations where a completely rigorous development is desirous or necessary, so it behooves us to show how we may switch to one–in midstream, so to speak–during a previously semiformal heterogeneous development.

The way that a transition to a formal development from a heterogeneous development can occur is by formalizing the (heterogeneous) semiformal specification at hand. In our example, the most detailed semiformal specification we are given is a heterogeneous data flow diagram with predicate parts (see Figure 4). To formalize this, we express the diagram in Z, with one schema per DFD process (this includes the predicate parts). A formalization (or, re-expression in Z), might look like the following (we have, implicitly, added extra data stores representing state to the DFD of Figure 4 in order to simplify the formalization process.)

$$\begin{array}{rcl}
data & ::= & [\, c, t : seq\ \mathbb{R} \,] \\
Generator & ::= & [\, \Delta data ; \lambda? : \mathbb{R} ; NUMBER? : \mathbb{N} \mid P \,] \\
Load & ::= & [\, \Xi data ; \Delta Cells \mid Q' \,] \\
Cells & ::= & [\, bursts, arrivals : seq_{NUMBER}\ cell \,] \\
Create\_and\_Init & ::= & [\, MQL? : \mathbb{N} ; \Delta Ready \,] \\
Place\_INITIAL & ::= & [\, \Delta Ready ; \Xi Cells ; INITIAL? : \mathbb{N} \,] \\
Ready & ::= & [\, ready : seq_{MQL}\ cell \,] \\
Simulate & ::= & [\, GROUPS? : \mathbb{N} ; \Delta Ready ; \Delta Stats ; \Xi Cells \mid R \,] \\
Stats & ::= & [\, count, wait : seq_{GROUPS}\ cell \,] \\
Output & ::= & [\, \Xi Stats ; stats! : T \,]
\end{array}$$

(The process for translating a DFD into Z schemas is given elsewhere, but it is straightforward, and in certain cases automatable. We have omitted type and data declaration details, but these are straightforward as well, and resolve to creating a data dictionary. Unfortunately, this latter process is not automatable.)

In the Z formalization, P, Q', and R are all formal specifications, but are expressed as predicates. P is the predicate for the generator specification of Figure 4. Q' represents the predicate $\forall i : 0, ..NUMBER \bullet Q$ (with Q as before), and R is as specified in Figure 4. The resulting set of schemas and predicates is a formal heterogeneous specification, except with predicative programming specifications as the *invariants* of Z schemas. We haven't yet described how to deal with these kinds of specifications. To work with them, we have two options:

1. *Translate the schemas Generator*, *Load*, and *Simulate* into predicates (this amounts to adding appropriate variable declarations and possibly frame conjuncts to a predicate), and leave P, Q' and R untouched. The resulting formal specification will be a heterogeneous combination of predicate specifications and Z schemas. Formal development of an implemen-

tation can then proceed using theorems and rules like those developed in Appendix A, along with other standard formal techniques.

2. *Translate P, Q', and R to Z notation* and leave them as the bodies of the schemas in question. In this particular example, this amounts to a syntactic rewrite of P, Q' and R (since all three specifications are feasible). The resulting formal specification will be homogeneous, and implementation may proceed using any approach to development with Z.

The decision as to which development path to follow–formal, semiformal, or heterogeneous–may occur at any time during an initially heterogeneous development. For example, we could have performed the unformalization of the heterogeneous specification after the construction of a heterogeneous structure chart, or after the development of heterogeneous structure text. We might have performed formalization, too, in similar places. The choice of when to perform such formalizations or unformalizations will depend on the nonfunctional and functional constraints on the development *at the time the decision is made.* The heterogeneous approach to development allows designers to take into account *changes in constraints* throughout the construction process, which will hopefully provide them with the flexibility that they need for difficult developments.

END OF ASIDE.

In attempting to construct an implementation of our solution, we choose to follow the path of developing heterogeneous C code, which will mean we carry out all the final refinements of formal parts at the text level. The resulting heterogeneous specification is as follows, with P and R as above, with $Q' = \forall i : 0, ..NUMBER \bullet Q$, and with the definitions of the functions $enqueue$, $dequeue$, and $front$ as described earlier.

```
#include <stdio.h>
#include <stdlib.h>
#define NUMBER 2000
#define MQL 100
#define GROUPS 10
#define INITIAL 20

main(){
  double bursts[NUMBER], arrivals[NUMBER];
  double boundary, time, wait[GROUPS];
  int i;
  long count[GROUPS];
  struct cell current;
  FILE *fp;
  char *fname="data";

  /* Generator */

  fp=fopen(fname,"w");
  P;
  fclose(fp);

  /* Create & Initialize Queue */
  head=tail=length=0;

  for(i=0; i<GROUPS; i++){
    count[i]=0;
    wait[i]=0.0;
  }

  /* Load data */

  fp=fopen(fname,"r");
  for(i=0;i<NUMBER;i++)
    fscanf(fp,"%lf %lf\n",&c[i],&t[i]);
  Q';

  /* Copy into C data structures (this will
     be eliminated later) */

  for(i=0; i<NUMBER;i++){
    bursts[i]=c[i];
    arrivals[i]=t[i];
  }

  current.burstlength=bursts[0];
  current.arrivaltime=arrivals[0];

  /* Place INITIAL in the queue */

  for(i=0; i<INITIAL; i++)
    enqueue(bursts[i],0.00);

  i=INITIAL;
  time=0.0;
  boundary=arrivals[length];

  /* FCFS Simulation */

  R;

  printf("\nGroup ");
  for(i=0;i<GROUPS;i++) printf("%7d",i);

  printf("\nCount ");
  for(i=0;i<GROUPS;i++) printf("%7d",count[i]);

  printf("\nWait  ");
  for(i=0;i<GROUPS;i++) printf(" %6.1f",wait[i]);

  printf("\nAverage");
  for(i=0;i<GROUPS;i++) printf("%7.1f",(wait[i]/count[i]));

  printf("\n\n");
}
```

We consider this specification to be formal: we treat the C code portions as predicates (even if we are not willing to provide all of the C parts with predicate definitions. For example, the statement `boundary=arrivals[length]` has predicate definition `boundary:=arrivals(length)`. The declaration `double bursts[NUMBER]` has predicate semantics `bursts:[NUMBER*real]`.

> *ASIDE.* There are several details we have not gone into in the above heterogeneous specification. Foremost amongst these ideas is that of data refinement. We will have to provide implementations (in C) for the data objects contained in the (formal) data dictionary described elsewhere, and for the extra declarations introduced in the heterogeneous specification. We must be able to prove that these implementations are reasonable for the specification as given. In this particular example, the data refinements to be proven are relatively trivial (we do not get into issues of precision here, since we treat C as a mathematical notation. The notions of overflow, etcetera, while important, are implementation concerns, and are beyond our interest here.), and resolve to syntactic translation. In other situations, the proof obligations will be more complicated, and will require more effort to discharge. *END OF ASIDE.*

The next stage is to refine the specifications P, Q', and R to code. This is a straightforward process. We describe the refinement to P here:

```
t0:=0;
n:=random;
c0:=if (n<=0.8) then .9*random+.1 else 9*random+1;
i:=1;
while (i<NUMBER) do (
  n:=random;
  ci:=if (n<=0.8) then .9*random+.1 else 9*random+1;
  ti:=-lambda * ln(random) + t(i-1);
  i:=i+1
);
```

The final refinement to the simulator specification R might look something like the following:

```
time:=0; i:=INITIAL;
while (stable) do(
  updatecount;
  while((time+boundary>arrivals(i))∧(i<NUMBER)∧ (length≠MQL))do(
          enqueue(bursts(i),arrivals(i)-boundary);
          i:=i+1);
  current:=front;
  dequeue)
```

The refinement for Q' is easy, and we omit it.

We can now translate these implementations into C code. When we finally do this, we choose to eliminate the arrays c and t, and write the computed burst and arrival times directly to the file `data` as they are computed, for the sake of saving storage. We omit the code here, as it is a straightforward translation from the developed specifications above.

The heterogeneous specifications we have used in this example allowed us to introduce the predicate specifications[13] at the most convenient time for us as designers. It appears likely that any semantic gaps introduced by using such specifications (e.g., in the construction of the DFDs) are

[13] And, more generally, *any* useful notation.

small, since the formal notations are used and manipulated without making reference to any semi-formal specifications at a different level of development[14]. We also believe that the development presented herein may be simpler (from this developer's point of view) and possibly shorter than one that would arise through the use of standard structured analysis and design. One reason for this latter suggestion is that a lot of structure text (i.e., pseudocode) does not have to be generated because of the early introduction of the formal notations. Furthermore, the early introduction of such notations may simplify the generation of "code" (be it pseudo- or concrete), simply because a great deal of the code specifications are already present in the data diagram. Certainly, the development presented above is more rigorous than the standard SA/D process; in fact, the development can be extended to make it as rigorous as desired (as we discussed earlier).

We should emphasize once more that this development is not the only one possible. It should also be clear that the development is not necessarily the best: it just happens to be an approach that made sense to the author, and used the notations that best suited the problem *and* author at the time of development. Other developers will have other opinions (and other methods of development). That is to be expected; no development process or notation can be all things to all people. The heterogeneous approach has the ability to take *all* development methods and choices of notation into account and generalize them in such a way so as to make the most flexible method and notation for any particular problem.

B.2 A formal derivation

We now consider an example of a formal heterogeneous development: the process of rigorously developing code from a formal heterogeneous specification, using–for example–the techniques, theorems, and concepts developed in Appendix A. The particular example we consider is one of computing the *natural square root* of a natural number s (see [Morg94] for a detailed examination of this problem and its solution developed with the refinement calculus). For our notations, we use specification statements and predicates. Our development will use the refinement relations $\sqsubseteq$ and $\Leftarrow$ as necessary, and we will use the theorems of Appendix A to justify refinement steps where needed. We do not present a detailed description of our algorithmic approach to solving the problem; it will be similar to that of [Morg94], so we refer the reader there for further details.

Our initial specification is the specification statement

$$r : [\, r'^2 \leq s < (r'+1)^2 \,],$$

and as in Appendix A we use the primed-unprimed variable conventions of predicative programming for the sake of simplicity. Our first refinement will be identical to the initial step in [Morg94]:

$$\begin{array}{ll} \sqsubseteq & \mathbf{var}\ q : \mathit{nat} \bullet \\ & q, r : [\, r'^2 \leq s < q'^2 \wedge r'+1 = q' \,]. \end{array}$$

Next, setting I (an invariant) to $r^2 \leq s < q^2$ (and, accordingly, $I' = r'^2 \leq s < q'^2$), we introduce an initialization through a sequential composition, using the law *Leading Assignment* of the refinement calculus.

$$\begin{array}{lll} \sqsubseteq & \mathbf{frame}\ q, r \bullet r'^2 \leq s < q'^2; & (i) \\ & q, r : [\, I, I' \wedge r'+1 = q' \,] & (ii) \end{array}$$

[14] In fact, the predicate parts can be viewed–if it is useful–as processes of the DFD.

Note that (i) is a predicate; its introduction is due to a refinement by parts law akin to **Theorem 4** in Appendix A. Switching to the predicative programming notation, we can refine (i) (inside the frame) as follows:

$$\begin{array}{rcl} (i) & \Leftarrow & q' = s+1 \land r' = 0 \\ & \Leftarrow & q := s+1; r := 0 \end{array}$$

Next, we refine (ii), and, using the law for introducing a loop (on invariant I', guard $q = r+1$ and variant $q' - r'$) obtain:

$$\begin{array}{rcl} (ii) & \sqsubseteq & \textbf{do } q \neq r+1 \rightarrow \\ & & \quad q, r : [\, r+1 \neq q \land I, I' \land q' - r' < q - r \,] \quad (iii) \\ & & \textbf{od} \end{array}$$

Once again, the refinement of (ii) is performed in the refinement calculus. To refine (iii), we introduce an intermediate natural variable, p, and refine to a sequential composition on $mid = r < p' < q$ (again, justified by our ability to perform partwise refinement heterogeneously):

$$\begin{array}{rl} \sqsubseteq & \textbf{var } p : nat \bullet \\ & \quad \textbf{frame } p \bullet (r+1 < q \Rightarrow r < p' < q); \quad (iv) \\ & \quad q, r : [\, r < p < q \land I, I' \land q' - r' < q - r \,] \quad (v) \end{array}$$

The first half of the sequential composition is easy to refine:

$$(iv) \quad \Leftarrow \quad p := (q+r) \div 2$$

(where $\div$ is integer division). To refine (v), we introduce a selection using the appropriate refinement calculus law, with guard $s < p^2$.

$$\begin{array}{rcl} (v) & \sqsubseteq & \textbf{if } s < p^2 \rightarrow \textbf{frame } q \bullet (s < p^2 \land p < q \Rightarrow I' \land q' < q) \quad (vi) \\ & & [] \; s \geq p^2 \rightarrow \textbf{frame } r \bullet (s \geq p^2 \land r < p \Rightarrow I' \land r < r') \quad (vii) \\ & & \textbf{fi} \end{array}$$

The final two refinements are easy (in fact, in our opinion it is easier to see and prove the refinements in this stage of the development than when using the refinement calculus):

$$\begin{array}{rcl} (vi) & \Leftarrow & q := p \\ (vii) & \Leftarrow & r := p \end{array}$$

and thus code has been generated.

It is not our claim that the previous example is *shorter* than that obtained through using the refinement calculus by itself. Indeed, the number of refinement steps in both derivations is almost the same[15]. It is our claim that in several cases the actual refinements in each step are easier to see (in particular, the refinements (iv), (vi) and (vii), amongst others). This is because we have introduced specifications in the predicative notation, and in this notation the specifications in question have a clear and obvious refinement.

[15] Also, it is not clear if it is even a reasonable metric for comparison.

This is not the only way the refinement of this specification could have been presented; indeed, there are a large number of different approaches that could have been taken. In particular, it might be beneficial to consider introducing the **do**-loop using the predicative notation, which has a very simple way of handling loops (specifically, the invariant and variant are hidden in the recursive refinement, and do not have to be explicitly dealt with in the refinement law). This would of course lead to a different–and possibly shorter–development.

B.3 An Applied Example

We now consider an example of applying heterogeneous specification and development in a more applied setting. Specifically, we are interested in constructing a program for use in modelling an *ionic equilibrium*. The equilibrium in question involves Fe^{3+} (iron(III) ion), arsenate (AsO_4^{3-}), nitrate (NO_3^-), and sodium (Na^+) ion. We desire to use a particular method–*modelling the system as a polynomial*–to find the concentration of hydrogen ion in the equilibrium. Once this concentration has been determined, we can then calculate the concentrations and activity coefficients of the species and the ionic strength of the solution. The exact system in question (along with the names of the hydrolysis constants we use for each reaction) are as follows. For the iron species:

$$\begin{aligned}
\mathbf{k1} &: Fe^{3+} + H_2O &\rightarrow\quad & H^+ + FeOH^{2+} \\
\mathbf{k2} &: FeOH^{2+} + H_2O &\rightarrow\quad & H^+ + Fe(OH)_2^+ \\
\mathbf{k3} &: Fe(OH)_2^+ + H_2O &\rightarrow\quad & H^+ + Fe(OH)_3 \\
\mathbf{k4} &: Fe(OH)_3 + H_2O &\rightarrow\quad & H^+ + Fe(OH)_4^-
\end{aligned}$$

and for the arsenate species:

$$\begin{aligned}
\mathbf{k1p} &: H_3AsO_4 &\rightarrow\quad & H^+ + H_2AsO_4^- \\
\mathbf{k2p} &: H_2AsO_4^- &\rightarrow\quad & H^+ + HAsO_4^{2-} \\
\mathbf{k3p} &: HAsO_4^{2-} &\rightarrow\quad & H^+ + AsO_4^{3-}
\end{aligned}$$

To calculate the ionic strength, U, of the solution, we use the following formula:

$$U = \frac{1}{2}\sum_i m_i z_i^2,$$

where m_i is the molality of ion i and z_i is the charge on the ion. To calculate the activity coefficient of each species, we use the Davies equation (for each ion i):

$$\frac{-At \times z_i \times \sqrt{U}}{1 + 0.7797 \times \sqrt{U}}.$$

(The *At* factor will in general depend on the relative permittivity and the temperature.)

The general modelling process will be approximately as follows. The reader should take note of its iterative nature.

- Initialize the hydrolysis and system constants for the given equilibrium from a data file. The file will be formatted for use with other programs, so its structure is fixed at development time; thus, it may contain data which is irrelevant for our purposes.

- For each system observation (which consists of a *temperature*, a *pH*, a *concentration of total iron* and a *concentration of total arsenic*), calculate the coefficients of the polynomial model (set up so that the roots of the polynomial give the concentration of H^+). Compute the roots of the polynomial (to within a given precision). From this, determine the concentrations and activity coefficients of the species, and the ionic strength of the equilibrium.

Since in our model we will be dealing with polynomials with real coefficients, we realize that the roots of the polynomial may be complex. By placing an interpretation on a root as a concentration of H^+, a complex value for a root is meaningless. Thus, we will have to ensure–somehow–that the root-finding process produces at least one real, positive root. In general this is impossible, but shortly we will describe how and why we can avoid the problem.

We commence our specification by describing a few of the data objects to be used in design.

$$\begin{array}{rcl}
N & : & nat \\
a & : & [(N+1) * real] \\
root & : & [(N+1) * complex] \\
nposroots, nob & : & nat \\
p1, q1, cconc, eps, pH, kw & : & real \\
\mathbf{k1}, \mathbf{k2}, \mathbf{k3}, \mathbf{k4}, \mathbf{k1p}, \mathbf{k2p}, \mathbf{k3p} & : & real \\
cFe, cAs & : & real
\end{array}$$

In this data dictionary, N is the degree of the polynomial (in general, we may want to allow the user to change N when executing the program so that experimentation with the model can occur). a is a list of (real) polynomial coefficients (the polynomial has the form $x^N + a_1x^{N-1} + .. + a_N$). $root$ will contain the N (complex) roots of the polynomial ($root(j)$ has the form $x + iy$, where $i^2 = -1$ and x and y are real). We declare the arrays to be of length $N+1$, since we envision eventual implementation of the system in FORTRAN, where arrays are indexed from 1.

We choose to use *Bairstow's method* in finding the complex roots of the polynomial model. Previous experience has shown that Bairstow's method is reasonably well-behaved for the polynomials we will encounter in this application (there are many other good approaches, though they are typically more complicated to implement). There are several reasonable ways in which we might choose to specify Bairstow's method in this situation:

1. A direct implementation, using code from an appropriate numerical library (e.g., IMSL). Such an implementation will typically be in FORTRAN. The benefit of this approach is that a likely well-tested implementation can be reused. This will not be an option in all development situations, since we cannot always be assured of having a good reuse library.

2. A formal specification. This might then be refined into an implementation. There are several formal specifications we can give. One might be (in Hehner's notation):

   $$x^N + a_1x^{N-1} + ... + a_{N-1}x + a_N = \prod j : 1, ..N+1 \bullet (x - r'(j)) \qquad (2)$$

   where r is an N-element list of complex numbers. This can be data refined quite easily to a specification which uses a list that is twice as long and implements the complexes as pairs of consecutive reals, which is one way in which complexes can be denoted in FORTRAN.

The specification (2) is suitably definitional to allow implementation using *any* complex root-finding algorithm. Since we *know* that we want to use Bairstow's method, it seems reasonable to alter our formal specification to take this information into account.

Bairstow's method can be described (semiformally) as follows:

Algorithm: Bairstow's method.
Given: initial factors p_0, q_0, real coefficients $a_1, .., a_N$ (where a_i is the coefficient of x^{N-i}, and the coefficient of x^N is 1.)
Method:

1. Calculate sequences b and c from the equations:

$$b_{-1} = 0 \wedge b_0 = 1 \wedge \forall j : 1, ..N+1 \bullet b_j = a_j - b_{j-1} \times p_0 - b_{j-2} \times q_o,$$
$$c_{-1} = 0 \wedge c_0 = 1 \wedge \forall j : 1, ..N+1 \bullet c_j = b_j - p_0 \times c_{j-1} - q_0 \times c_{j-2}.$$

2. Calculate the δ factors from the equations:

$$\begin{aligned} c_{N-2}\delta_p + c_{N-3}\delta_q &= -b_{N-1} \\ (-b_{N-1} + c_{N-1})\delta_p + c_{N-2}\delta_q &= -b_N \end{aligned}$$

   Then, $p = p_0 + \delta_p$ and $q = q_0 + \delta_q$ (providing the system has a solution).

3. Repeat steps 1. and 2. until δ_p and δ_q are suitably small.

4. The values of p and q give us a quadratic factor, $r(x) = x^2 + px + q$. Compute the zeros of $r(x)$, giving us two roots of the initial polynomial. Copy the values of b_i into a_i for all i in order to perform synthetic division of the initial polynomial by $r(x)$ (i.e., $(x^2 + px + q) \times (x^{N-2} + b_1 x^{N-3} + .. + b_{N-2}) = x^N + ... + a_N$). Reapply the process, starting at step 1., to the divided polynomial, substituting the values of p and q for p_0 and q_0 respectively. Stop the repetition when the divided polynomial has been reduced to a constant factor.

A formal specification of Bairstow's method might look like the following. Note that it is much more detailed than our initial formal specification; further, notice that the lists a, b, c, and *root* are all declared elsewhere. We also assume that the coefficients have been normalized by $a(1)$ (i.e., the coefficient of x^N is 1.0).

```
BAIRSTOW =$\lambda N : nat; p0, q0, eps : real; itag : nat \bullet$ (
  var $m, it : nat; d, e, f, p, q, sum, os : real \bullet$ (
  $it' = itag \wedge itag' = 0$;
  while ($N > 0$) do (
    if($N = 1$) then $root'(itag) = -a(1) \wedge N' = N - 1$
    else if($N = 2$) then  $x^2 + px + q = (x - root'(itag+1))(x - root'(itag)) \wedge N' = N - 2$
    else (
      $p' = p0 \wedge q' = q0 \wedge m' = 1$;
      while($m < it$) do (
        $b'1 = a1 - p \wedge b'2 = a2 - p \times b1 - q \wedge c'1 = b'1 - p \wedge c'2 = b'2 - p \times c'1 - q \wedge$
        $\forall j : 3, ..N+1 \bullet b'j = aj - p \times b'(j-1) - q \times b'(j-2) \wedge c'j = b'j - p \times c'(j-1) - q \times c'(j-2)$;
        $c(N-2)^2 \neq c(N-3) \times (c(N-1) - b(N-1)) \Rightarrow c(N-2) \times \delta'_p + c(N-3) \times \delta'_q =$
    $-b(N-1) \wedge c(N-1) \times \delta'_p + c(N-2) \times \delta'_q = -bN;$
$p' = p + \delta_p \ \wedge q' = q + \delta_q \ \wedge sum' = \mid \delta_p \mid + \mid \delta_q \mid;$
if $(m = 1)$ then $os := sum$ else ok;
$m \neq 5 \ \vee sum \leq os \Rightarrow ($
    $sum \leq eps \Rightarrow ($
        $x^2 + px + q = (x - root'(itag+1))(x - root'(itag));$
        $N > 0 \Rightarrow (\forall i : 1, ..N+1 \bullet a'i = bi; itag' = itag + 1 \wedge N' = N - 2 \wedge m' = it + 1));$
    $m < it \Rightarrow m := m + 1);$
$m = 5 \wedge sum > os \Rightarrow N' = 0 \wedge m' = it + 1)))))$
```

When we come to implement this specification, we may have to reify away the use of complex numbers into either a record structure or pairs of consecutive reals in an array. We will choose the latter approach here, but will not present the (formal) data refinement here.

In order to demonstrate that this specification determines the roots of the polynomial, we need to show that it refines (2), given earlier. This entails: (a) showing that BAIRSTOW terminates; and (b) that it satisfies (2). (a) is straightforward. We sketch a proof (by strong induction on N) of (b) here, by showing that BAIRSTOW establishes, for a natural $itag$ and assuming a mapping $a_i = a(i)$,

$$x^N + a_1 x^{N-1} + ... + a_N = \prod i : itag, ..itag + N \bullet (x - root'(i)).$$

Base case: $N = 1$. The polynomial is $x + a_1$, and BAIRSTOW sets $root'(itag) = -a(1)$. The result follows.

Inductive hypothesis: Assume, for all $N \leq k$ and coefficients $a_1, .., a_k$,

$$x^k + a_1 x^{k-1} + .. + a_k = \prod i : itag, ..itag + k \bullet (x - root'(i))$$

Inductive step (prove for $N = k + 1$): From BAIRSTOW, we know that $x^2 + px + q = (x - root'(itag))(x - root'(itag + 1))$. Furthermore, from the inductive hypothesis,

$$x^{k-1} + b_1 x^{k-2} + .. + b_{k-1} = \prod i : (itag + 2), ..(itag + k + 1) \bullet (x - root'(i)).$$

But, from the definition of the b_i's, we see that

$$(x^2 + px + q)(x^{k-1} + b_1 x^{k-2} + .. + b_{k-1} = x^{k+1} + a_1 x^k + .. + a_{k+1}.$$

Therefore,

$$(x - root'(itag))(x - root'(itag + 1)) \prod i : (itag + 2), ..(itag + k + 1) \bullet (x - root'(i)) \\ = x^{k+1} + a_1 x^k + .. + a_{k+1},$$

which implies that

$$\prod i : itag, ..itag + (k + 1) \bullet (x - root'(i)) = x^{k+1} + a_1 x^k + .. + a_{k+1},$$

giving us our result.

We have not taken precision into account here; that is an implementation issue, and must be dealt with at that time.

We now consider the problem as a whole. On first examination, a solution to the problem at hand will likely best present itself in an iterative manner (since, among other parts, the root-finding mechanism is iterative). This is typical of many numerical methods–since the method itself is invariably iterative, an iterative specification is invariably the best. Therefore, we choose to specify a solution using heterogeneous Jackson process diagrams (PSDs), because we feel that the PSDs are a good graphical formalism for expressing sequential programs[16]. The heterogeneality will arise through compositions of process boxes with higraph predicates, and through compositions of structure text (developed from the PSDs) with formal specifications.

Our initial PSD specification is quite trivial, and is shown in Figure 8.

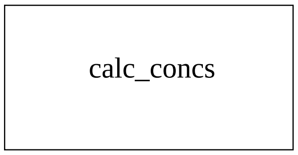

Figure 8: Initial Jackson specification

We refine this specification by adding initialization, data access, and a loop to iterate over the number of observations. This is shown in Figure 9.

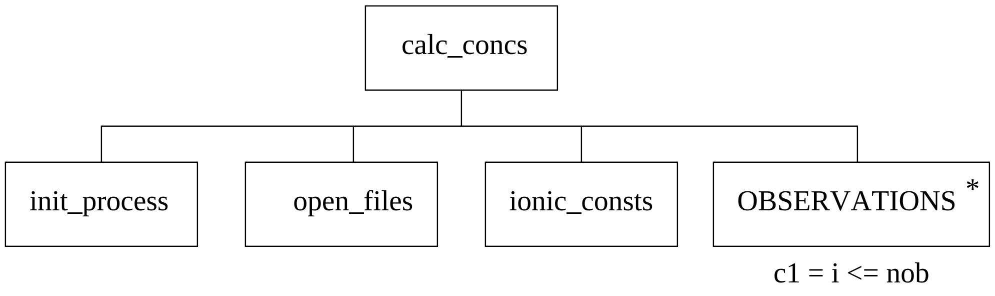

Figure 9: Adding initialization and observation loop

The loop *OBSERVATIONS* is the main part of the specification, and since it is fairly large, we refine its details in a separate diagram. This elucidation is described in Figure 10. Here, we have added heterogeneous details: a formal specification describing (the structure of) our initialization of the polynomial model; initialization of the quadratic factors needed for Bairstow's method; description of Bairstow's iterative method (the box *BAIRSTOW* in Figure 10 refers to the specification `BAIRSTOW` given above); calculation of the number of positive real roots (specification P in Figure 10); and a specification choosing the "likeliest" root out of a list of solutions (the "likeliest" root is the one that gives an H^+ concentration closest to 10^{-pH}. If there are no real positive roots, then we use 10^{-pH} as the concentration of hydrogen ion, which typically gives good results in this situation). Specification Q in Figure 10 corresponds to choosing the "likeliest" root.

From our perspective, this completes a PSD description of the system[17]. There are now several ways in which we could continue this development. We choose to take the following path:

1. As in standard JSP, annotate the PSD blocks with process details, conditions, etcetera. Further elucidation of the PSD blocks might occur here, if we so desire.

[16]Other developers will of course prefer different notations.
[17]Other developers might choose to add more detail, which is perfectly reasonable.

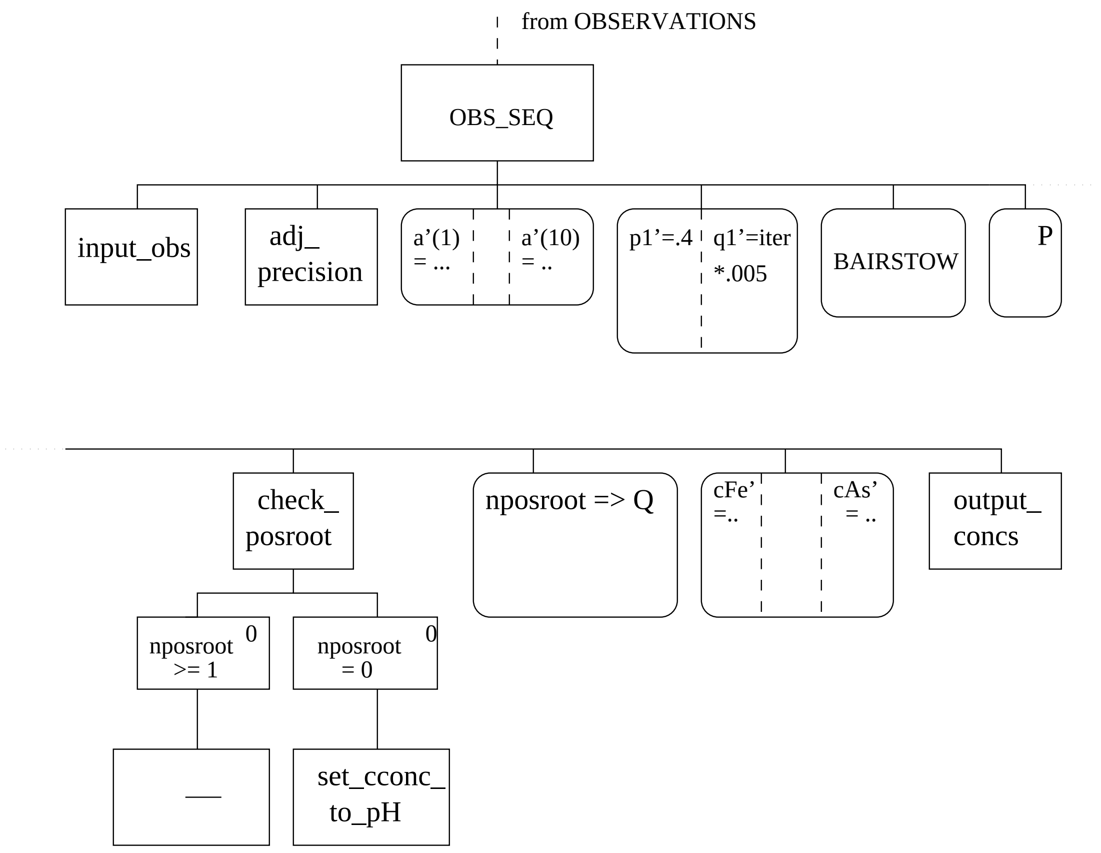

Figure 10: OBSERVATIONS loop

2. Refine the formal parts (e.g., Bairstow's method, etc.) either visually or textually, in order to acquire a better understanding of system structure. These refinements may be added to the heterogeneous PSD if it is useful or convenient to do so.

3. Generate structure text as per JSP usual, but the text for the higraph parts should be their text equivalents. This latter generation is a form of syntactic translation, mapping from higraph predicates to predicates. The mapping is described in [Paig95].

4. Further refinement and code generation may proceed, as per heterogeneous specification usual. The JSP implementation process is applied to the JSP structure text parts. Implementation of the formal parts proceeds as per predicative programming usual.

5. Syntactic translation of the (implemented) structure text and predicate combination into an implementation language occurs.

A translation of the heterogeneous PSD into heterogeneous structure text is as follows. We omit certain details (e.g., the actual values for the polynomial coefficients, a full specification of Bairstow's method) in order to keep the specification manageable. The specification BAIRSTOW is as described earlier (in our example, $N = 9$).

```
calc_concs SEQ
  init_process;
  open_files;
  ionic_consts;
  OBSERVATIONS ITER WHILE i≤nob
    obs_seq SEQ
      input_obs;
      adj_precision;
      $a'(1) = .. \wedge .. \wedge a'(N+1) = 1.0$;
      $p1' = 0.40 \wedge q1' = iter \times 0.005$;
      BAIRSTOW(N,p1,q1,eps,itag);
      $nposroots := ¢(\S i : 1, ..N+1 \bullet \mathbf{Re}\ root(i) > 0 \wedge \mathbf{Im}\ root(i) = 0)$;
      check_posroot SEL nposroot=0
        set_cconc_to_pH;
      check_posroot ALT nposroot≥1
        ok;
      check_posroot END
      $nposroot > 0 \Rightarrow (N = 0 \Rightarrow cconc := 10^{-pH}) \wedge (N > 0 \Rightarrow ($
         $cconc := root(i')$;
         $\mathbf{ensure}\ \forall j : (\S k : 1, ..N+1 \bullet \mathbf{Re}\ root(k) > 0 \wedge \mathbf{Im}\ root(k) = 0) \bullet$
                $\mid root(i) - 10^{-pH} \mid \leq \mid root(j) - 10^{-pH} \mid))$;
      $cFe' = .. \wedge .. \wedge cAs' = .. \wedge U' = .. \wedge gAs' = ..$;
      output_concs;
    obs_seq END
  OBSERVATIONS END
calc_concs END
```

There are at least two ways in which development from this heterogeneous text specification may now proceed: implementing the structure text (the semiformal parts); or refining the predicates into code. Of course, both processes must occur in order to obtain an implementation. The implementation of structure text can proceed as per JSP normal. Refinement of predicates can occur as is standard for predicative programming. After code is reached (i.e., after the heterogeneous specification has been refined to an implementation), transliteration to a programming language may occur. This latter process is mainly an issue of practicality: we are quite satisfied with a *heterogeneous program* as our final implementation, but since we will want to execute it, translation is a necessity. Of course, transliteration to an implementation language is not always trivial (issues to be dealt with include variable declaration and implementation, precision, etcetera), but it is a concern that is beyond our interest here. Our requirements specified that FORTRAN was to be the implementation language; we omit the actual implementation.

There are other development paths which are possible once a heterogeneous PSD has been created:

- Formalize the structure of the heterogeneous PSD and translate it into, for example, predicates. This will give us a structurally-operational predicate specification. Development may then proceed as per predicate normal, although a certain amount of detail (e.g., conditions and extra formalization) will have to be added in order for translation to be complete.

- Unformalize the visual predicates in Figure 10 and place PSD process boxes in their place. Development may then proceed as per JSP normal. Of course, any number of the visual predicates (from zero to all) may be unformalized, in order to meet development constraints. This process will result in a loss of information, but it may be reduced by using techniques described elsewhere.

With a heterogeneous specification and development, we may adapt our development process so that it best fits the situation at hand.

B.4 Conclusions

The main reason for us presenting such heterogeneous specifications is to demonstrate that notations (both formal and semiformal) can be used together–both formally and semiformally!–to construct programs. In many cases, the decision to use multiple notations will be based on convenience: programmers may feel more comfortable using a specific notation for a specific task; a heterogeneous notation may fit the problem or process better than a homogeneous one; and it may be easier to take nonfunctional constraints into account through the use of heterogeneous specifications. Thus, one of the main benefits of heterogeneous specifications is their flexibility and ability to adapt to the situation at hand. Even so, whether heterogeneous developments are shorter, simpler, or easier to understand is for the most part dependent upon the problem at hand. Indeed, there will be many examples where *homogeneous* specifications and developments will prove more convenient than going to the trouble and expense of heterogeneous specifications[18]. Still, it is useful–and likely even necessary–to know that notations can be used combinationally when necessary, for it seems likely that development situations *will* arise where homogeneous notations will prove to be less than adequate for the task at hand.

References

[AbLa93] M. Abadi and L. Lamport. Composing Specifications, *ACM Trans. on Programming Languages and Systems*, 15(1), January 1993.

[18]Of course, homogeneous specifications are just a special case of heterogeneous specifications!

[AsCe93] E. Astesiano and M. Cerioli. Multiparadigm Specification Languages: a first attempt at foundations. In *Proc. Semantics of Specification Languages*, Springer-Verlag, 1993.

[Back78] R.J.R. Back. *On the correctness of refinement steps in program development*, PhD thesis, Dept. of Computer Science, University of Helsinki, 1978.

[Back90] R.J.R. Back. Refinement calculus II: parallel and reactive programs. In *Stepwise Refinement of Distributed Systems*, LNCS 430, Springer-Verlag, 1990.

[BaVo89] 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.

[DeMa79] T. DeMarco. *Structured Analysis and System Specification*, Yourdon Press, 1979.

[Dijk76] E.W. Dijkstra. *A Discipline of Programming*, Prentice-Hall, 1976.

[Dijk93] E.W. Dijkstra. The Unification of Three Calculi. In *Program Design Calculi*, Proceedings of the NATO ASI on Program Design Calculi, Springer-Verlag, 1993.

[Grie81] D. Gries. *The Science of Programming*, Springer-Verlag, 1981.

[Gutt93] J.V. Guttag and J.J. Horning. *Larch: Languages and Tools for Formal Specification*, Springer-Verlag, 1993.

[Hail86] B. Hailpern. Multiparadigm languages and environments (guest editor's introduction to a special issue), *IEEE Software*, 3(1), January 1986.

[HarD88] D. Harel. On Visual Formalisms, *Comm. ACM*, 31(5), May 1988.

[HarJ90] J.S. Hares. *SSADM for the Advanced Practitioner*, Wiley, 1990.

[HeMa88] E.C.R. Hehner and A.J. Malton. Termination Conventions and Comparative Semantics, *Acta Informatica*, 25 (1988).

[Hehn93] E.C.R. Hehner. *A Practical Theory of Programming*, Springer-Verlag, 1993.

[Hoar69] C.A.R. Hoare. An Axiomatic Basis for Computer Programming, *Comm. ACM*, 12, Oct. 1969.

[Hoar85] C.A.R. Hoare. *Communicating Sequential Processes*, Prentice-Hall, 1985.

[Hoar94] C.A.R. Hoare. Unified Theories of Programming, Technical Paper, Oxford Computing Laboratory, July 1994.

[Jack81] M.A. Jackson. *System Development*, Prentice-Hall, 1981.

[Jone90] C.B. Jones. *Systematic Software Development using VDM*, Prentice-Hall, Second Edition, 1990.

[King90] 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.

[Kron93] K. Kronlöf, ed. *Method Integration: Concepts and Case Studies*, Wiley, 1993.

[Morg94] C.C. Morgan. *Programming from Specifications*, Second Edition, Prentice-Hall, 1994.

[Paig94] R.F. Paige. Formal specifications and theories of programming. Depth Paper, Department of Computer Science, University of Toronto, October 1994.

[Paig95] R.F. Paige. Higraph-based Predicate and Heterogeneous Specification, March 1995.

[Part90] H.A. Partsch. *Specification and Transformation of Programs,* Springer-Verlag, 1990.

[PHG91] D.A. Penny, R.C. Holt, and M.W. Godfrey. Formal Specifications in Metamorphic Programming. In *VDM '91: Formal Software Development Methods*, Fourth International Symposium of VDM Europe, LNCS 551, Springer-Verlag, 1992.

[ScRo77] 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.

[Spiv89] J.M. Spivey. *The Z Notation: A Reference Manual*, Prentice-Hall, 1989.

[Ward93] N. Ward. Adding Specification Constructors to the Refinement Calculus. In *Proc. FME '93: Industrial-strength Formal Methods*, Springer-Verlag, 1993.

[Wing90] J.M. Wing. A specifier's introduction to formal methods. *IEEE Computer*, 23(9), September 1990.

[WiZa92] 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.

[ZaJa93] P. Zave and M. Jackson. Conjunction as Composition, *ACM Trans. on Software Engineering and Methodology*, 2(4), October 1993.