Formalizing and Integrating SA/SD with a Program Design Calculus

Richard F. Paige

Technical Report CS-98-06

July 8, 1998

Department of Computer Science

4700 Keele Street North York, Ontario M3J 1P3 Canada

Formalizing and Integrating SA/SD with a Program Design Calculus

Richard F. Paige
Department of Computer Science, York University,
Toronto, Ontario, Canada, M3J 1P3. paige@cs.yorku.ca

Abstract

The notations and parts of a process for Structured Analysis and Structured Design [2, 15] are formalized within a program design calculus [5]. The formalizations are used in an application of a meta-method for formal method integration, combining SA/SD and the program design calculus of [5]. The methods are integrated so as to provide more rigor to the informal method, and to make the program design calculus more applicable for use in large-scale systems development. Two integrations are proposed, and the two approaches are applied in some detail to an example.

1 Introduction

Methods can be integrated by defining relationships between them, so that they can be cooperatively and productively used together. Method integration in a software context has seen recent research, on techniques to combine particular sets of methods [12, 13], and on systematic approaches to semiformal method integration [7]. An aim of this paper is to use a method for formal method integration [11] to combine a program design calculus and a structured method. We carry out this integration for two reasons: so as to provide a rigorous basis to the informal method; and to make the calculus more useful for large-scale software development.

We commence with a brief overview of method integration and the general approach we take to accomplish it. We then summarize a meta-method for formal method integration, discussed in [11]. The approach is applied to combining predicative programming [5] with SA/SD [2, 15]. Finally, we use the integrated method in an example.

Contributions of this paper include: a new, detailed application of the method of [11], combining an informal method and a program design calculus, as well as a new example of using an integrated method. We also provide new formalizations of SA/SD notations, using a design calculus, and explain some of the repercussions of using such an approach to formalize parts of a structured method.

1.1 Methods and method integration

A *method* for software development consists of notations and processes. When integrating methods, incompatibilities between methods are resolved so that the approaches can be safely and effectively used together [7]. *Method integration* is the process of combining two or more methods to form a new method. *Formal* method integration is method integration involving at least one formal method [11].

Method integration in a software engineering context is a problem of growing research interest for several reasons. One rationale is that it is unlikely that one method will suffice for use in the development of complex systems [6]; method integration can provide techniques for dealing with this complexity. Furthermore, method integration has been used successfully in industrial practice, for example, at BT [13], Philips [7], Praxis [3], and elsewhere.

1.2 Heterogeneous notations and specifications

Notations are important parts of any method, and will play a key role in the approach we use to integrate methods. In particular, we integrate notations as a first step in combining methods.

A *heterogeneous notation* is a combination of notations. It is used to write *heterogeneous specifications*, which are compositions of partial specifications written in two or more notations. Heterogeneous specifications are claimed to be useful for a number of reasons: for expressiveness; for producing simpler specification languages [16]; for writing simpler specifications than might be produced using one language [16]; for ease of expression [3]; and because they have proven to be successful in practice [13, 16]. We use them as a first step towards formal method integration, and discuss this process starting in Section 1.3.

The meaning of a heterogeneous specification is given by formally defining the meaning of a composition of partial specifications. If we combine exactly two notations, where one is formal and the other informal, we can formally define the composition by formalizing the informal notation in the formal language being used. More generally, we may want to build a heterogeneous notation from more than two formal or informal notations. In this case, to give a semantics to heterogeneous specifications written using the new notation, we construct a *heterogeneous basis* [11], a set of notations and translations between notations that is used to provide a formal semantics to heterogeneous specifications.

In this paper, we are interested in combining the use of predicative programming and SA/SD; thus, we will eventually formalize SA/SD notations in predicative notation. For more on heterogeneous bases, their construction, and use, see [11]. By formalizing SA/SD notations in predicative notation, we effectively construct a very simple heterogeneous basis consisting of only SA/SD notations and predicative notation.

1.3 Integrating methods with heterogeneous notations

We use the process of constructing heterogeneous notations as a first step in formal method integration. Integration occurs by formalizing the meaning of compositions of partial specifications written in the notations of interest; such a process also requires us to ensure that the heterogeneous notation can be parsed. Once this is done, integration continues by generalizing processes of methods so as to use heterogeneous notations, and by defining relationships between the (generalized) processes of the methods that are being combined. More details are given in Section 2.

We suggest that the process of formal method integration can be partially systematized by using heterogeneous notations. Their construction allows the notation-related complications associated with method integration to be dealt with first. The heterogeneous notation can therefore provide a basis for defining relationships between the processes of the methods that are being combined.

2 A Meta-Method for Formal Method Integration

A meta-method for formal method integration was introduced in [11]. It described an informal strategy for combining formal methods with other methods, where all methods are to be used for system specification and design. The meta-method is intended to be used as a thinking tool, to assist method engineers in determining the roles that methods can play and the relationships that can be defined between processes in an integration. We summarize the meta-method here, and then apply it in Section 3. The meta-method can be summarized as follows.

1. **Fix a base method.** Fixing a base method is a step aimed at assisting the method engineer in helping to determine possible relationships between the methods being combined. A base method may support more of the software development cycle than other methods.

2. **Choose invasive methods.** The invasive methods are to be embedded in the base method. Selection could be done on the basis of notational or methodological convenience, e.g., for adding new techniques to a base method—such as refinement to a non-refinement based method.

3. **Combine the notations of the base and invasive methods.** In general, this is accomplished by constructing a heterogeneous basis to include new notations of the methods we want to integrate (and also extra notations, e.g., those we could use in supplying a formal semantics to specifications). A single formal notation that is to be used to provide a formal semantics to system specifications can be chosen at this point.

4. **Describe informal relationships between the processes.** It is defined how the processes for the base and invasive methods will work together. The process of the base method may first be generalized

to use notations from the invasive methods; this is similar to the integration in [14]. Once generalization has occurred, informal descriptions (using annotated box-and-arrow diagrams, see Fig. 2) of relationships between the process of the base method and the processes of the invasive methods are constructed. This description explains how the processes are to be used together in the new method. Some patterns of relationships are described in [11].

5. **Guidance to the user.** Hints, examples, and suggestions on how the integrated method can be used are provided, as well as descriptions on how the methods will change when used in the combination.

As mentioned, the meta-method is intended as a thinking tool, to support method engineers in determining roles for methods to play, and to support determining relationships that can be defined between processes. When more rigor is needed in integrating methods, more powerful techniques such as meta-modeling [10] will need to be applied.

3 An Integrated SA/SD–Predicative Programming Method

We present the construction of a method integrated from SA/SD and predicative programming. It is produced by applying the meta-method presented in Section 2. With respect to the meta-method, we make the following decisions.

- SA/SD will be the base method, since we intend to use SA/SD to guide the production of system specifications. Predicative programming will be the invasive method.
- The predicative notation will serve as a "basis notation" in defining the meaning of heterogeneous specifications. To this end, we will formalize SA/SD notations in predicative notation.
- Predicative programming will *supplement* an SA/SD process. Techniques for constructing DFDs, process specifications (PSPECs), the data dictionary, and structure charts will be generalized to use predicative notation, and will be supplemented by predicative proof rules.
- In the integration, predicative programming will be used to specify and develop system components with complex functionality and to specify selected data components, e.g., those that are complex. Predicative programming will also be used to *refine* specifications through to the level of code.

Predicative programming and SA/SD are complementary methods: SA/SD offers notations (DFDs, structure charts) better suited to description of software architecture and system component relationships than does predicative programming. And predicative programming provides refinement rules that allow formal development of code from specifications, whereas SA/SD does not. The integration will help resolve a noted weakness with using program design calculi: in lengthy refinements, it is easy to lose track of refinement context. By combining the design calculus with a method like SA/SD, we allow developers to apply the calculus to refining small parts of the system, where context can be maintained.

We call the result of combining the two methods *heterogeneous SA/SD* (H-SA/SD, for short). As we shall see, we can restrict the use of predicative programming with respect to the overall size of the system specification. We do this by explicitly using the predicative notation—and thereby its refinement rules—only where we have reason to believe it will be useful in writing system specifications.

The meta-method requires formalizing SA/SD notations—i.e., creating a heterogeneous basis—so we now present two new formalizations of these languages.

3.1 Formalizations of SA/SD notations

To formalize SA/SD notations—specifically, DFDs and structure charts—we must fix an interpretation for each notation, and provide a formal expression of the notation (based on this interpretation) using predicative programming. More generally, we can view formalization of informal notations as part of the process of constructing a heterogeneous basis so that it includes the informal notations. Building a heterogeneous basis of some form is required if we are integrating more than two notations.

There are many interpretations we might take for SA/SD notations, If the interpretation or formalization that we demonstrate is not appropriate for a particular development setting, it should be changed.

3.1.1 Formalizing data flow diagrams

A data flow diagram (DFD) describes a system in terms of processes and information flow among them. We interpret DFDs as concurrent operations on a global state, with interprocess communication through channels that provide the mechanisms for data flow. The interpretation is expressed using predicative notation, which is convenient to use for describing communication, channels, and concurrency [5]. Alternative formalizations of data flow diagrams are presented in [9, 13]. The formalization given here is new. The meta-method in this paper can accommodate use of other formalizations.

In DFDs, state is represented by data stores, operations are DFD bubbles, and system input and output is represented by external entities. Thus, flow from a process to a data store depicts change of state. Internal data flows are from process to process, and represent communication through channels that may be local to the processes in question, or globally addressable but used only by specific processes.

Channels in predicative programming are statically declared as follows. A channel declaration c of type T applies to the predicative specification P that follows it.

$$\textbf{chan}\ c : T \cdot P = \exists\, M_c : [\infty^* T] \cdot \exists\, T_c : [\infty^* xnat] \cdot \textbf{var}\ r_c, w_c : xnat := 0 \cdot P$$

M_c and T_c are the *message* and *time* scripts, respectively—infinite lists of messages, and the times the messages were sent along c. r_c and w_c are *read* and *write* cursors, of extended natural type, while P is a predicative specification. To use c in P, a specifier can write $c?$ to describe a computation that reads one input on channel c; $c?$ advances the read cursor r_c by one. An occurrence of c in P describes the most recently read message (formally, an occurrence of c is a list indexing, $M(r_c - 1)$). Output of an expression e on channel c is written $c!e$.

Formalizing DFDs is as follows. The process is syntax-driven, but requires formalizers to choose how individual processes use channels, and to determine how scoping of channels is to be described.

1. *Formalize data stores and external entities:* for each data store, generate a state definition with an appropriate type and the same name as the store. For each external entity, provide a type for the data flow from or to the external entity, and formalize the flow as a state definition.
2. *Formalize processes and data flow:* for each data flow bubble A:
   (a) *Formalize each process:* generate a predicative specification A which includes any appropriate processing details.
   (b) *Formalize flow:* for each data flow d, determine a type for the data flow, then for each flow:
   - *Process-to-Store Flow:* from A to a data store, add *storename* to the frame of specification A.
   - *Process-to-Process Flow:* from A to a bubble B, declare a channel, $\textbf{chan}\ d : T$ that is accessible to both processes A and B. There are many alternatives to describing how A and B use channel d. For example, the processes could use the channel concurrently, and so the flow could be formalized as the parallel composition $\textbf{chan}\ d \cdot (A \parallel B)$. The processes could also use d as a queue or buffer, resulting in $\textbf{chan}\ d \cdot (A.\ B)$ (where . is predicative sequencing). Alternatively, the channel could be declared when the state (associated with data stores) is declared. Then, only processes connected to d in the DFD can use d in the formalization. This approach to formalization is necessary to deal with channel scoping situations that cannot be specified statically. For example, consider Fig. 1. We must declare channels u, v, and w with appropriate scopes. But predicative programming, as described in [5], has static scoping for channels, and the scoping implied by Fig. 1 cannot be described statically (though it can be described dynamically). One formalization is:

     $$\textbf{chan}\ u : T_1 \cdot \textbf{chan}\ v : T_2 \cdot \textbf{chan}\ w : T_3 \cdot (A \parallel B \parallel C)$$

     where A does not use v, B does not use w, and C does not use u. Another alternative is to declare u local to A and B, as in $\textbf{chan}\ v : T_2 \cdot \textbf{chan}\ w : T_3 \cdot (\textbf{chan}\ u : T_1 \cdot A \parallel B) \parallel C$, where w is not used by B, nor v by A. Channel use is formalized in the processes A, B, C.

By formalizing data flow as messages on channels between processes, we are underspecifying: a channel is typically bi-directional, readable, and writable by the connected processes. Data flow in DFDs is usually

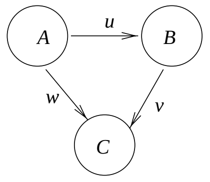


Fig. 1: DFD with scoping problem for channels

unidirectional. Thus, we require the specifiers of processes to use the channel appropriately, as dictated by the DFD. If this level of flexibility is inappropriate, the formalization of channels (e.g., to include read and write permissions) will have to be changed.

An interesting effect of using predicative notation to formalize data flow diagrams is that a designer can use predicative refinement as a formal basis for DFD refinement. And since refinement in predicative programming is just boolean implication, carrying out DFD refinement proofs in theorem proving tools, such as PVS, seems to be possible. This could allow automated formal support for refining large-scale DFDs with a rigorous semantics.

3.1.2 Structure charts

Structure charts are used to describe the modular structure of a system, and to describe the information flow across module interfaces. An example of a structure chart is in Fig. 6. The interpretation we place on structure charts is similar to data flow diagrams: we interpret a structure chart as a (possibly concurrent) set of operations on a global state which communicate through (local or global) channels. Each structure chart unit represents a particular operation or set of operations. Again, we only give this formalization as an example.

1. *Formalizing State:* formalize each data generator (an ellipse attached to a leaf node) as we did for the data store in the formalization of DFDs.
2. *Formalizing Units:* for each structure chart unit A, create a predicative specification with the same name. Provide process descriptions appropriate for the unit.
3. *Formalizing State Interactions:* for flow from a bubble to a unit (or vice versa), formalize this link as was done for flow from and to data stores in the DFD formalization, i.e., by adding state to the frame of the specification representing the receiving unit.
4. *Formalizing the Mechanisms for Information Flow:* for data d passed from a unit A to a unit B, declare a channel d of type T. d will be declared so that it is in the scope for both A and B. The channel will have a type T appropriate for the data being passed.
5. *Formalizing Information Flow from A to B:* the channels declared in the previous step will be used by processes to exchange information. The way processes use the channels must be defined by the formalizer, as was done for DFDs in the previous section. Alternatives include having A and B use

channel d concurrently; interpreting the channel as a buffer or queue; or interleaving the use of d by A or B with uses of other (local) channels. Because of the modular model provided by structure charts, the formalization of channel use may differ from that produced for the DFD model.

By providing a predicative formalization of structure charts, and with the formalization of DFDs given in the preceding section, we have a method for proving that a structure chart is an implementation of a DFD. If D is a predicative formalization of a DFD and S is a predicative formalization of a structure chart, then we need prove $\mathit{DesignOfDFD} = \forall\, \sigma, \sigma' \cdot (D \Leftarrow S)$ (where σ, σ' are the pre- and poststate of the data stores), in order to show that S implements D. For large specifications, this will be time consuming; since refinement in predicative programming is boolean implication, this suggests that it will be useful and appropriate to carry out the proofs using automatic tools, like PVS or the Refinement Calculator [1].

The proof rule *DesignOfDFD* is a formalization of the process of transforming DFDs to structure charts; such a process is sometimes known as “transaction” or “transform flow” analysis [2, 15]. Predicative refinement can therefore provide a mathematical basis for such a task.

SA/SD notations have been formalized in predicative notation because we want to integrate predicative programming and SA/SD. In general, we may want to combine several different (formal or semiformal) notations, or may want to use a formalization notation different from those used for specification. In such a case, a heterogeneous basis will have to be built. An example of a heterogeneous basis can be found in [11]; it instead uses Z for formalization of SA/SD notations.

3.2 An H-SA/SD Process

We now continue with applying the meta-method, moving to Step 4: generalization of processes, and description of informal relationships between processes. We depict an example of an H-SA/SD process in Fig. 2. In this diagram, ellipses represent steps or phases in the process, and boxes represent (heterogeneous) method products. Thick lines between ellipses represent ordering of process steps. Arrows represent “production” (out of an ellipse) or “use” (into an ellipse).

The relationships between the methods are described in the diagram: predicative programming supplements PSPEC production, the construction of a data dictionary, and the process of designing an implementation from specifications. This latter relationship occurs via use of predicative refinement. In more detail, a process for H-SA/SD is as follows (different processes can be derived to meet more complex needs).

1. *Analyze problem.* A context-level data flow diagram of the system is produced.

2. *Decomposition.* The context DFD is decomposed into more detailed process components. The processes of such a diagram can be written as standard data flow diagram processes. Some processes may later be given predicative PSPECs. If, at the decomposition stage, we decide that a process will later be given a predicative PSPEC, then we draw this process as a *rounded rectangle*. This allows a reader

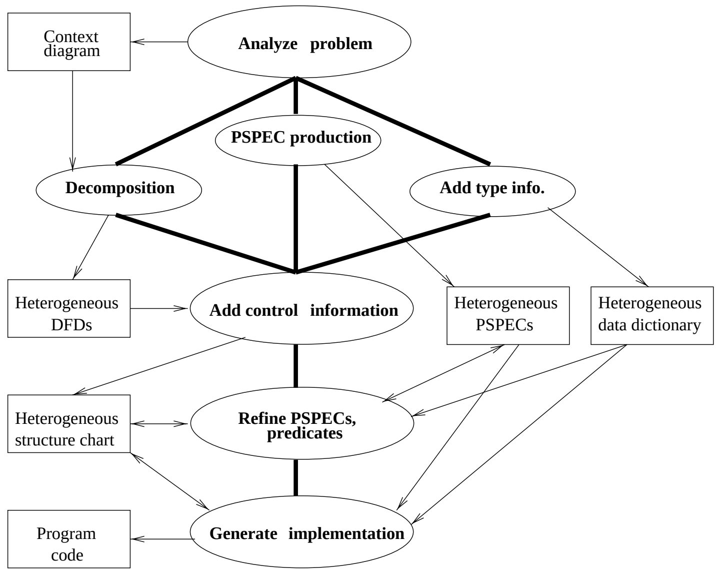


Fig. 2: H-SA/SD process

of the diagram to quickly and efficiently determine the scope of use of the predicative programming method. We can also add predicative PSPECs later, in Step (4).

We obtain restrictability of the predicative notation, by allowing the user of the integrated method to control the extent of use of the notation in the heterogeneous DFD specification.

3. *Add type information.* Extended BNF notation can be used, as can predicative data type notations (e.g., bunches, lists) for augmenting the formal partial specifications.

4. *PSPEC Production.* Process specifications are constructed. PSPECs may be specified using predicative notation, or informal pseudocode notation as is standard in SA/SD; predicative notation would be used where formality is needed, or where a refinement-based development is planned. The pseudocode would be used everywhere else. We therefore treat pseudocode PSPECS as *rough sketches* [6]: we consider them to be formal (or formally expressible), but with no provided rigorous meaning.

   By giving the user control over where and to what extent the predicative notation is used for writing PSPECs, we obtain restrictability of the notation.

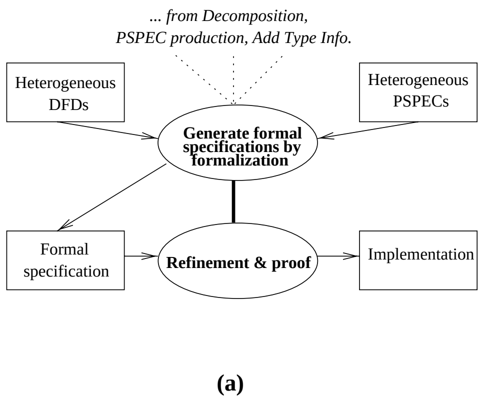
5. *Formulate modular structure.* The heterogeneous DFD is transformed into a structure chart, either by hand or by transform or transaction analysis. The resulting structure chart may include predicative partial specifications. These parts will be depicted as rounded rectangles, while standard structure chart units will be depicted as rectangles. The formal parts can be used in a subsidiary support or a central construction role as required.

6. *Refinement of PSPECs and predicates.* PSPECs are elucidated and can be semiformally refined by adding more detail. Predicative parts are formally refined, using algorithm refinement rules from [5]. We therefore obtain restrictability of predicative programming by specification construction.

7. *Implementation.* The design is implemented. The predicative parts are (possibly data) transformed to code. The non-predicate parts can be transformed to a program design language level, which can then be implemented as is standard in Structured Design. Translation into a programming language may have to occur.

3.3 An alternate integration

Different integrations of SA/SD and predicative programming can be constructed using the meta-method of Section 2, relying on the predicative formalization of SA/SD notations. One useful integration is a *formalization*. In this alternative, full formalization occurs after Step (4) of the H-SA/SD method; thus, a heterogeneous DFD, PSPECs, and a data dictionary have been constructed. Using the formalizations, the system specification is mapped into a predicative programming specification. Development of an implementation from this form can occur using predicative programming. This alternative is depicted in Fig. 3(a). It is similar to the approaches suggested in [12, 13]. Akin to these approaches, we can also formalize so as to evaluate what is missing in our system specifications, by deriving a formal specification which, in turn, could be used to derive new (heterogeneous) specifications related to the original specification. The derived specifications could then be fed back into the original H-SA/SD method.



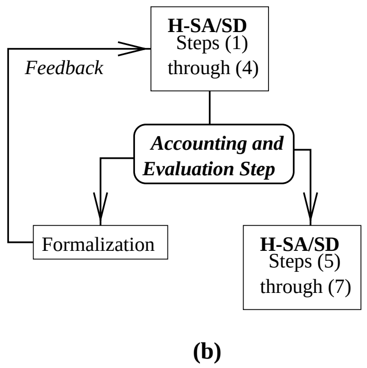

H-SA/SD Steps (1) through (4)
Feedback
Accounting and Evaluation Step
Formalization
H-SA/SD Steps (5) through (7)
(b)

Fig. 3: Alternative integration: (a) formalization; (b) relationship to H-SA/SD

The generation of a formal specification is shown occurring immediately after the steps of PSPEC production, decomposition, and type information addition. A formal specification could also be constructed elsewhere in the development process, e.g., after producing structure charts.

Fig. 3(b) shows a relationship between formalization alternative and the H-SA/SD method. As shown in Fig. 3(b), after Steps (1)–(4) of the H-SA/SD method, there can be an evaluation step, where a decision on how to proceed with the development is made. This evaluation should be done with reference to the current status of the development, and with consideration of the functional and non-functional requirements on the development. The evaluation process may suggest that the development can proceed according to the H-SA/SD method. It may also suggest that the formalization path (possibly with feedback to the original H-SA/SD specifications) be taken.

With the construction of a heterogeneous basis, as discussed in Section 1.2, we can integrate SA/SD with methods other than predicative programming, using the meta-method. By defining translations between formal notations, like Z and predicative programming, we effectively gain the ability to use new formal notations, within the H-SA/SD method as it has been defined. Such a heterogeneous basis would transitively give us a version of H-SA/SD that uses Z, and not predicative programming. One can therefore think of H-SA/SD as a meta-method itself, in which one or more formal notations and methods can be used in cooperation with SA/SD.

3.4 Comparison with other approaches

The SAZ method [12] links SSADM with Z. There is a direct transformation from an SSADM development to a pure Z development. Z specifications produced using SAZ can be studied, and new information from them can then be fed back in to the SSADM development. In our approach, predicative programming and SA/SD can be linked in a similar fashion, as suggested in Fig. 3(b).

A different approach is offered in [8]. Therein, SA is combined with VDM; VDM specifications supplement and can be automatically derived from the products of SA. The SA and VDM techniques are used in parallel; heterogeneity arises implicitly, for example, when VDM specifications are used as DFD process specifications. An advantage of this approach is that there is tool support for extracting partial VDM specifications from the products of SA.

The integration of Larch with SA [14] uses Larch interface language specifications for writing PSPECs, and LSL specifications for defining a data dictionary. The relationship between Larch and SA specifications is left informal. The intent with this integration matches ours with respect to the goal of restrictability: in the method of [14], Larch specifications are used only where helpful; in H-SA/SD, predicative programming is used only where helpful. A difference in the approaches (besides the methods involved) is that H-SA/SD offers a formal semantics for the composition of partial specifications.

4 Example

In this section, we briefly demonstrate the use of the integrated H-SA/SD method. Our goal here is to demonstrate the properties of the integrated method, and to describe the use of the alternatives to the H-SA/SD method presented in Section 3. To this end, we consider only a small example.

The problem we wish to solve is that of constructing a simulator for a scheduler that can provide service either in a first-come first-served or a round-robin fashion. The initial requirements are as follows.

A system is needed to simulate two short-term schedulers. The system must generate test data and simulate either a first-come first-served (FCFS) or a round-robin (RR) short-term scheduler, depending on user choice. The system will have two parts: the first will generate data. The second part will load the data into a "ready" queue, and simulate the scheduler operation on the data.

The generator part produces two vectors of data, one holding *NUMBER* random CPU burst lengths, and the other holding *NUMBER* 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, with parameter *LAMBDA*.

The second system component is the simulator, which simulates the appropriate algorithm on the test data. Total wait time and average wait time should be output upon completion. For the most efficiency, a circular ready queue of fixed length 100 should be used.

4.1 Analyze the problem

The first step is to construct a heterogeneous DFD. A context-level DFD appears as shown in Fig. 4(a). Recall that we use rounded rectangles to specify processes that will have predicative specifications.

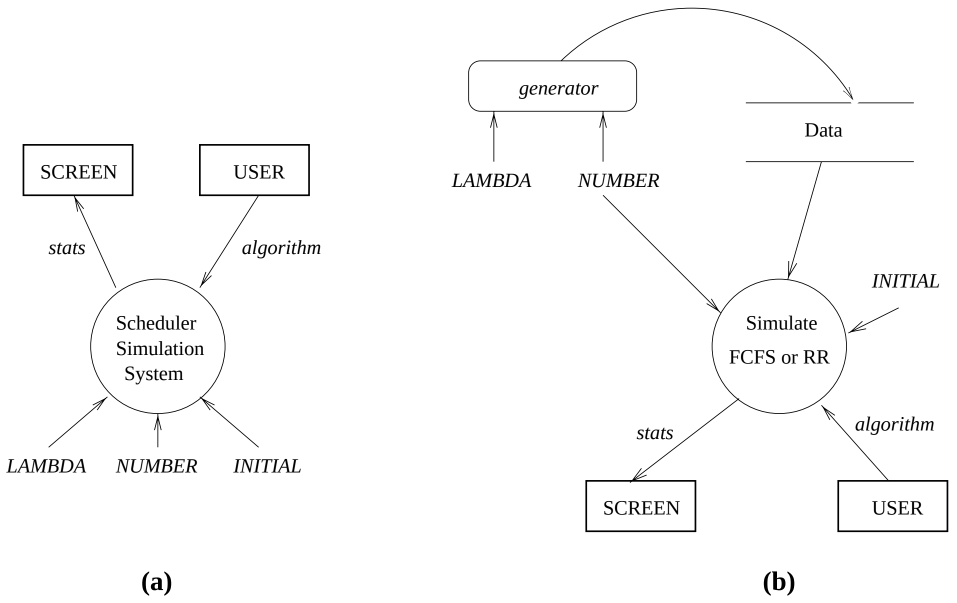


Fig. 4: (a) Context diagram; (b) Level-1 heterogeneous DFD

4.2 Decomposition

We now refine the context DFD into its two parts: the data generator, and the simulators. We eventually want to use the predicative notation to develop the data generator, so we describe the generator process as a rounded rectangle. In the decomposition, we also add store *Data*. The refinement is shown in Fig. 4(b). We come back to the decomposition stage later, after constructing part of a data dictionary.

4.3 Data dictionary creation

In Fig. 4(b), we made use of several data objects. These must be added to the data dictionary, and can then be used in other parts of the solution. First, we give several constants and their types.

$$INITIAL, NUMBER : nat \qquad LAMBDA : real \qquad algorithm : rr, fcfs$$

INITIAL is the number of processes to be in the ready queue at the start of simulation. *algorithm* is the simulation algorithm that is to be run (either *fcfs* or *rr*). The remaining data definitions are as follows. The data store *Data* consists of two *NUMBER*-length lists of generated *real*s, *bursts* and *arrivals*. The other items are used to maintain the ready queue.

$$bursts, arrivals : [NUMBER * real] \qquad\qquad boundary, head, tail, length : nat$$

The queue *ready* will be a list of records. Each record in the list, a *cell*, consists of a burst length, an arrival time, and a group field, used in calculating distribution statistics.

$$\begin{aligned} cell &= \text{“}burstlength\text{”} \to real \mid \text{“}arrivaltime\text{”} \to real \mid \text{“}group\text{”} \to int \\ ready &: [100 * cell] \end{aligned}$$

4.4 PSPEC production and further decomposition

The next step is to provide PSPECs (e.g., for *generator*) and to refine the *Simulate FCFS or RR* bubble. We show the refinement of the relevant bubble in Fig. 5.

The predicate corresponding to the rounded rectangle *generator* of Fig. 4(b) is as follows.

$$\begin{aligned} generator = \; & arrivals'(0) = 0 \wedge (bursts'(0) = 0.9 \times rand + 0.1 \vee bursts'(0) = 9.0 \times rand + 1.0) \wedge \\ & \forall\, i : 1, ..NUMBER \cdot (bursts'(i) = 0.9 \times rand + 0.1 \vee bursts'(i) = 9.0 \times rand + 1.0) \wedge \\ & \qquad\qquad arrivals'(i) - arrivals'(i-1) = -LAMBDA \times log_e rand \end{aligned}$$

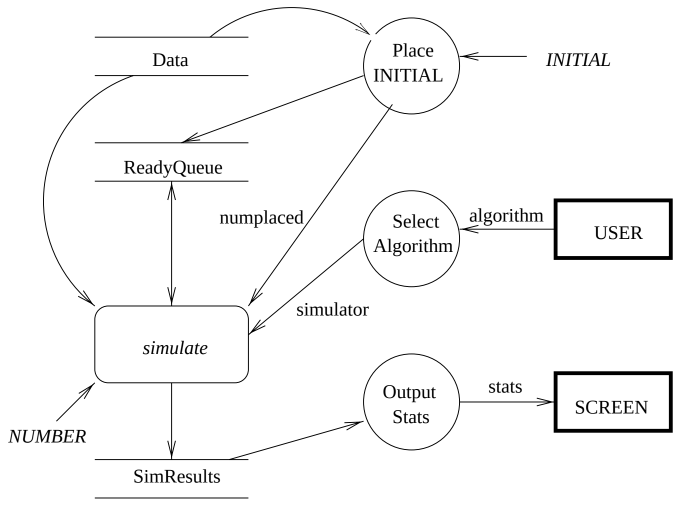


Fig. 5: Refinement of *Simulate FCFS or RR*

generator produces two lists of simulation data. The list *bursts* holds random burst lengths in the range $0.1, ..10$, and the list *arrivals* holds arrival times with a Poisson distribution. The parameterless function *rand* returns a random *real* in the range $0, ..1$.

The predicate *simulate* is a specification of the FCFS and RR simulators. We use a predicate for this part of the system because it has complex functionality. Since it is often reasonable to use operational specifications to describe simulations (because of their iterative nature), we make use of some program code. Recall that the queue to hold processes will be implemented in a circular fashion, and that when the simulation begins there are already *numplaced* processes in the ready queue.

We now are confronted with a decision: should we include specifications to read the data flow from *numplaced* and *simulator*, or should this be included in the formalization process? Simulation cannot proceed until an algorithm has been selected and the number of processes already in the ready queue (*numplaced*) is known. It is debatable whether reading such data values from channels should be part of the *simulate* PSPEC; including channel interactions in the PSPEC requires the specifier to know the formalization of DFDs.

We choose not to include specification parts to read data from the channels in the PSPECs; these will be included in formalization (see Section 4.7). The *simulate* PSPEC is as follows. A simultaneous assignment initializes the simulator variables. Then, a loop specifies the simulation; it terminates when the queue empties or overflows.

$$\begin{aligned} \mathit{simulate} \quad = \quad & \mathit{time}, \mathit{arriving}, \mathit{current}, \mathit{boundary} := 0, \mathit{numplaced}, \mathit{bursts}(0), \mathit{arrivals}(\mathit{length}). \\ & \textbf{while}\ (0 < \mathit{length} < 100)\ \textbf{do}\ (\textbf{if}\ (\mathit{simulator} = \mathit{fcfs})\ \textbf{then}\ \mathit{FCFS}\ \textbf{else}\ \mathit{RR}) \end{aligned}$$

(The dot operator is predicative sequencing.) Because predicative notation formalizes channels as state components, we can use the names of channels (i.e., *numplaced* and *simulator*) as variables in our specifications. The formalizations must guarantee that *simulator* and *numplaced* have values before they are used.

The function *dequeue*, which will be used in the simulator specification, removes the element at the front of *ready*, providing the queue is not empty.

$$\mathit{dequeue} = (\mathit{head} \neq \mathit{tail}) \Rightarrow (\mathit{head}' = (\mathit{head} + 1) \bmod 100 \wedge \mathit{length}' = \mathit{length} - 1)$$

The *FCFS* specification is as follows. We first update the statistics (the group to which a to-be-simulated process belongs, and the total running time of the simulation) in predicate *updatecount*. The simulator then calculates the number of processes, *marr*, that arrive during the simulation of the currently active process. These processes are queued, and then the current process is removed from the simulator.

$$\begin{array}{l}
\mathit{updatecount}. \\
\mathit{marr} := \mathit{MAX}\ j : \mathit{arriving}, ..\mathit{NUMBER} \cdot (\mathit{time} + \mathit{boundary} > \mathit{arrivals}(j)). \\
\mathit{head} \neq (\mathit{tail} + \mathit{marr} - \mathit{arriving}) \bmod 100 \Rightarrow \forall\, i : 0, ..\mathit{marr} - \mathit{arriving} \cdot \\
\quad \mathit{ready}' = ((\mathit{tail} + i) \bmod 100;\ \text{“}\mathit{burstlength}\text{”}) \rightarrow \mathit{bursts}(\mathit{arriving} + i) \mid \\
\qquad\qquad ((\mathit{tail} + i) \bmod 100;\ \text{“}\mathit{arrivaltime}\text{”}) \rightarrow \mathit{arrivals}(\mathit{arriving} + i) - \mathit{boundary} \mid \\
\qquad\qquad ((\mathit{tail} + i) \bmod 100;\ \text{“}\mathit{group}\text{”}) \rightarrow \lfloor \mathit{bursts}(i) \rfloor \mid \mathit{ready} \wedge \\
\quad \mathit{tail}' = (\mathit{tail} + \mathit{marr} - \mathit{arriving}) \bmod 100 \wedge \mathit{length}' = \mathit{length} + (\mathit{marr} - \mathit{arriving})\ . \\
\ \ \mathit{arriving}' = \mathit{marr} \wedge \mathit{current}' = \mathit{ready}(\mathit{head}).\ \ \mathit{dequeue}
\end{array}$$

The round-robin simulator *RR* has a similar specification. The main difference between the *RR* and *FCFS* simulators is that the current active process in the *RR* simulator is “executed” for 1 time unit, and then is queued again, providing that it has not been serviced for its complete burst length.

4.5 Heterogeneous structure chart creation

The next step in the integrated method of Section 3 is to construct a heterogeneous structure chart, in order to describe the modular structure of the system. As input, we use the heterogeneous DFD and the data dictionary information. A heterogeneous structure chart is shown in Fig. 6. (Units with predicate specifications are again drawn as rounded rectangles.)

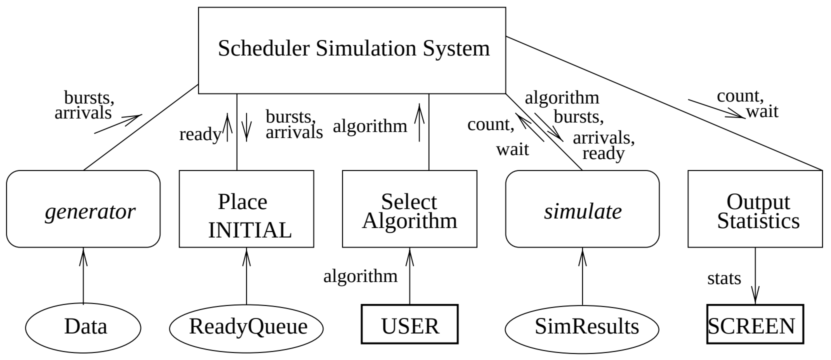


Fig. 6: Heterogeneous structure chart

4.6 Constructing an implementation

We sketch the construction of an implementation. Implementation proceeds by refining the specifications *generator* and *simulate* to code. We sketch the details, and commence with *generator*. The guard on the loop implementing *generator* is $i \neq NUMBER$, a variant is $NUMBER - i$, and an invariant is

$$1 \leq i < NUMBER \wedge \forall j : 1,..i \cdot arrivals(j) - arrivals(j-1) = -LAMBDA \times \log_e rand \wedge$$
$$(bursts(j) = 0.9 \times rand + 0.1 \vee bursts(j) = 9.0 \times rand + 1.0)$$

A *generator* implementation is as follows (omitting the refinement steps for space reasons).

```
$\textbf{var}\ n : real := rand \cdot$
$arrivals(0), i := 0, 1.$
$bursts(0) := \textbf{if}\ (n \leq 0.8)\ \textbf{then}\ 0.9 \times rand + 0.1\ \textbf{else}\ 9.0 \times rand + 1.$
$\textbf{while}\ (i \neq NUMBER)\ \textbf{do}\ ($
    $n := rand.$
    $bursts(i) := \textbf{if}\ (n \leq 0.8)\ \textbf{then}\ 0.9 \times rand + .1\ \textbf{else}\ 9.0 \times rand + 1.$
    $arrivals(i) := -LAMBDA \times \log_e(rand) + arrivals(i-1).$
    $i := i + 1)$
```

The final refinement to *simulate* might look something like the following.

```
$time, i, current, boundary := 0, numplaced, bursts(0), arrivals(length).$
$\textbf{while}\ (0 < length < 100)\ \textbf{do}\ (\textbf{if}\ (simulator = fcfs)\ \textbf{then}\ FCFS'\ \textbf{else}\ RR')$
```

FCFS′ is a refinement of the specification *FCFS*; it is as follows.

```
$updatecount.$
$\textbf{while}\ ((time + boundary > arrivals(i)) \wedge i < NUMBER \wedge length \neq 100)\ \textbf{do}\ ($
    $enqueue(bursts(i), arrivals(i) - boundary, \lfloor bursts(i) \rfloor).\ i := i + 1).$
$current := ready(head).\ dequeue$
```

(The above code is a transliteration of the recursive refinement structure obtained by refining *FCFS*.) Similarly, the specification *RR*′ is a refinement of *RR* (we omit its definition here).

After carrying out the refinements to the formal specifications above, we must produce a complete implementation by coding the remaining specification parts of Fig. 6. This encompasses the components *PlaceInitial, SelectAlgorithm, OutputStats*, and so on. We must also provide definitions that implement the data dictionary, and transform the programs generated by refinement into the implementation language. Finally, channels must be codified in an implementation language. We do not present these steps here.

4.7 Applying the formalization alternative

A formalization alternative to H-SA/SD was diagrammed in Fig. 3(a). In this relationship, complete formalization of the system specification occurred after Step (4) of the H-SA/SD method. Using the formalizations of SA/SD in predicative notation, formal specifications could be obtained from the heterogeneous specifications. These specifications could then be used, either to develop a final implementation via algorithm refinement, or to determine inconsistencies or missing parts in the informal specifications. Such an approach is offered in methods like SAZ [12] or the Yourdon variant discussed in [13].

In our example, we are given a heterogeneous data flow diagram with predicative parts (Fig. 5). By applying the formalizations given earlier, we obtain predicative specifications. System state is formalized from the data stores and external entities. We collect these specifications into a set of state declarations.

$$\begin{array}{rcl} bursts, arrivals & : & [NUMBER * cell] \\ algorithm & : & rr, fcfs \end{array} \qquad \begin{array}{rcl} count, wait & : & [10 * cell] \\ stats & : & StatType \end{array} \qquad ready : [100 * cell]$$

The processes *PlaceInitial*, *OutputStats*, and *SelectAlgorithm* all are formalized, resulting in three processes that may interact with system state. (We omit the bodies of the processes, including those parts that generate data on the channels, due to space constraints.) According to the formalization process, we obtain:

$$\begin{array}{rcl} PlaceInitial & = & \textbf{frame}\ ready \cdot \ldots \\ SelectAlgorithm & = & \ldots \end{array} \qquad OutputStats = \textbf{frame}\ stats \cdot \ldots$$

frames are used to specify the variables that can be changed by each specification. *SelectAlgorithm* changes no state, but uses name *algorithm* (see Fig. 5).

No formalization of the processes with predicative PSPECs need occur; we add frames to each process specification, renaming each specification in the process to be consistent with the convention followed for the preceding processes. The results of this process are shown below.

$$\mathit{Generator} = \mathbf{frame}\ \mathit{bursts}, \mathit{arrivals} \cdot \mathit{generator}$$

The predicate *generator* is as shown in Section 4.4. We now formalize the data flow in the specification, and express how processes use the flow. We first formalize two channels: between *PlaceInitial* and *simulate*, and between *SelectAlgorithm* and *simulate*, adding types in the process.

$$\mathbf{chan}\ \mathit{numplaced} : \mathit{nat} \cdot \mathbf{chan}\ \mathit{simulator} : \mathit{rr}, \mathit{fcfs}$$

The simulator predicate *simulate* is shown in Section 4.4. In “formalization”, it is wrapped with a frame. The process must also read values from the channels between it and *PlaceInitial* and *SelectAlgorithm*. This is shown in the specification *Simulate*, below.

$$\mathit{Simulate} \quad = \quad \mathbf{frame}\ \mathit{ready}, \mathit{count}, \mathit{wait} \cdot ((\mathit{numplaced}? \parallel \mathit{simulator}?).\ \mathit{simulate})$$

Simulate makes use of *numplaced* and *simulator* (by reading values from the channels before commencing the simulation), so these channels must be in its scope. We now describe how the system use the channels. The intent with our diagram was to state that *numplaced* data items are in the queue before the simulation starts; also, an algorithm (*fcfs* or *rr*) must be selected before simulation. We express this as follows.

$$\mathbf{chan}\ \mathit{numplaced} : \mathit{nat} \cdot \mathbf{chan}\ \mathit{simulator} : \mathit{rr}, \mathit{fcfs} \cdot \ (\mathit{PlaceInitial} \parallel \mathit{SelectAlgorithm}).\ \mathit{Simulate}$$

Data must be generated before the simulation begins. So the final formalization expresses this requirement.

$$\begin{gathered}\mathit{Generator}.\ (\mathbf{chan}\ \mathit{numplaced} : \mathit{nat} \cdot \mathbf{chan}\ \mathit{simulator} : \mathit{rr}, \mathit{fcfs} \cdot \\ (\mathit{PlaceInitial} \parallel \mathit{SelectAlgorithm}).\ \mathit{Simulate})\end{gathered}$$

Alternative formalizations of channel use and process interleaving are possible; the formalization we have given corresponds with our intuitive use of the DFD notation. We can use this formalization for algorithm refinement to code, or as documentation for developing an implementation from the informal specifications.

5 Conclusions

We have demonstrated the use of a general approach to integrating formal and semiformal methods, based on heterogeneous notations. A formal semantics for heterogeneous notations constructed from SA/SD notations and predicative notation was provided, through giving new formalizations of data flow diagrams and structure charts in terms of concurrent predicative specifications. A meta-method for formal method integration was applied in combining SA/SD and predicative programming. We applied the integrated method in solving a small problem. In the future, we plan to apply the integrated method to larger-scale problems, and explore alternative formalizations.

An issue we have not yet discussed is the effect of using heterogeneous notations on tools. Methods like SA/SD are well-supported by CASE tools, while formal methods like predicative programming can be supported by theorem provers, syntax and type checkers, etcetera. Combining notations suggests that it may be necessary to also combine tools. To provide tool support for the use of heterogeneous notations, we must also provide tool support for communication across tools, e.g., via an application framework. A benefit of using predicative notation in integration is that it is just boolean logic: any theorem prover or proof tool that uses boolean logic can be used to support the combination of methods. We envision using CASE tools associated with automated tools like PVS or STeP to support the integrated method.

The approach to method integration offered by heterogeneous notations is quite general: it can be used to combine formal methods with both formal and semiformal methods used for system specification and design. In future work, we plan to apply the approach to methods used for requirements analysis, as well as studying how to combine the use of tools for proof with CASE tools for system modeling and code generation.

References

[1] M. Butler, J. Grundy, T. Langbacka, R. Ruksenas, and J. von Wright, The Refinement Calculator: Proof Support for Program Refinement, in: *Proc. Formal Methods Pacific '97,* Springer, 1997.

[2] T. DeMarco, *Structured Analysis and System Specification* (Yourdon Press, 1979).

[3] A. Hall, Using Formal Methods to Develop an ATC Information System, *IEEE Software* **13**(2) (1996).

[4] U. Hamer and J. Peleska, The CIDS A330/340 Cabin Communication System – a Z Application, in: *Applications of Formal Methods,* (Prentice-Hall, 1995).

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

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

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

[8] P.G. Larsen, J. van Katwijk, N. Plat, K. Pronk, and H. Toetenel, Towards an integrated combination of SA and VDM, in: *Proc. Methods Integration Workshop,* (Springer-Verlag, 1991).

[9] P.G. Larsen, N. Plat, and H. Toetenel, A Formal Semantics of Data Flow Diagrams, *Formal Aspects of Computing,* **7**(5) (1995).

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

[11] R.F. Paige, A Meta-Method for Formal Method Integration, in: *Proc. Formal Methods Europe 1997,* LNCS 1313 (Springer-Verlag, 1997).

[12] F. Polack, M. Whiston, and K.C. Mander, The SAZ Project: Integrating SSADM and Z, in: *Proc. Formal Methods Europe 1993,* LNCS 670 (Springer-Verlag, 1993).

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

[14] J.M. Wing and A.M. Zaremski, Unintrusive ways to integrate formal specifications in practice, in: *Proc. VDM '91,* LNCS 551 (Springer-Verlag, 1992).

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

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