Specification and Refinement using a Heterogeneous Notation for Concurrency and Communication

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

Abstract

It is shown how to combine the Z formal specification notation with a predicative notation so as to be able to specify and reason about concurrency and communication. The integration is carried out so as to alleviate some of the limitations noted with previous integration approaches, such as the inability to use Z proof rules and tools with the integrated notation. In the process, it is demonstrated that it is not necessary to combine Z with a very different behavioural formalism in order to reason about concurrency.

1 Introduction

The Z notation [15] has proven to be useful and appropriate for specifying and reasoning about sequential software and hardware systems. The strengths of Z include its ability to construct specifications by parts, its growing tool support, and its proof system. Recent work on Z has studied its application to *concurrent systems*. In this growing body of work, there are two general classes of approaches:

1. *Extension approaches,* which apply Z, perhaps with some strengthening of specification or proof technique, to concurrent systems, e.g., [3].

2. *Integration approaches,* in which Z is combined with notations that are considered better suited to specifying and reasoning about time or concurrency, e.g., temporal logic [2], TLA [10], or CCS [7].

An advantage claimed with extension approaches is that compatibility with existing Z proof techniques and tools can be maintained. A disadvantage claimed of integration approaches is that they may have difficulty reconciling the semantics of the separate notations [3], especially when very different notations, such as Z and CCS, need to be combined. This can result in compatibility problems with the integrated notation and existing Z tools and proof rules.

This paper has several goals. First, we intend to demonstrate that it is possible to resolve some of the noted disadvantages with integration approaches and concurrency with Z. Second, we will demonstrate that it is not necessary to combine Z with a very different notation—such as CCS or TLA—in order to be able to reason about concurrency. We show that combining Z with a similar (though complementary) notation will suffice for reasoning about concurrency as well as communication. Finally, we show

that the integrated notation can support kinds of reasoning, e.g., regarding liveness and safety properties, akin to the application of Z to concurrency in [3].

We commence by constructing a heterogeneous notation, syntactically and semantically combining Z with predicative notation [8], for specifying and reasoning about concurrency and communication. We show how the semantics of the separate notations can be resolved, and how the notation can be used in specification, refinement and proof of properties. The approach is applied in several small specification, refinement, and proof case studies.

1.1 Organization of the Paper

We start with an overview of previous work, concentrating on the approach of Evans [3]; an aim of our paper is to show that an integrated notation for concurrency can be used in a manner similar to how Z is used for concurrency in [3]. Further, we will also show that there are advantages to using the integrated notation over Evans' approach. After a brief overview of predicative notation in Section 2.1, we explain our approach to integrating notations, in Section 3. Section 4 explains how to specify concurrency, and presents an example of proof of safety and liveness properties. Section 5 extends specification to communication through channels, and discusses deadlock. The approach is illustrated with several examples. Finally, we discuss the approach and its limitations, consider tool support, and summarize some further work.

2 Previous Work and Background

A number of different approaches to combining Z, concurrency, and proof have appeared in the literature. The main body of work in this area is by Duke et al [2], Fergus and Ince [4], Gotzhein [7], Evans [3], and Lamport [10].

The first three integration approaches propose the use of temporal logic in proving safety and liveness properties of Z specifications; this requires extending Z to temporal logic. With these approaches, temporal logic is used to reason about the histories of state changes that are produced by Z specifications. Operational styles of reasoning are used to prove properties by directly examining histories, an approach that has been suggested as impractical for all but the smallest of specifications [3].

Lamport [10] has suggested an approach to concurrency that integrates Z with TLA. In this integration, schemas are interpreted as actions, thus allowing use of TLA's inference rules to verify safety and liveness properties. Temporal logic operators must still be added to Z, and existing Z proof tools cannot be used directly.

Evans' work [3] has focused on the direct application of Z to specifying concurrent systems. Evans' approach augments Z specifications with an additional specification describing the system's dynamic behaviour given in terms of allowable sequences of state changes. Evans produces proof rules that can be used to verify safety and liveness properties of specifications. The standard Z rules are strengthened to ensure preservation of safety and liveness. The goal of Evans' approach is to maintain compatibility with existing Z proof techniques, and existing Z proof tools.

2.1 Predicative programming

Predicative programming is due to Hehner [8]. It is a program design calculus in which programs are specifications. Specifications are predicates on pre- and poststate (values of variables in the poststate are annotated with a prime; initial values of variables are undecorated). The weakest specification is $\top$ ("true"), and the strongest specification is $\perp$ ("false"). Refinement is boolean implication.

Definition 1. A predicative specification P on prestate σ and poststate σ' is refined by a specification Q if $\forall \sigma, \sigma' \cdot (P \Leftarrow Q)$.

The refinement relation enjoys various properties that allow specifications to be refined by parts, steps, and cases. Since refinement is just implication, carrying out a refinement is equivalent to carrying out a logical proof. Therefore, the refinement rules of predicative programming are laws of boolean logic; see [8] for a list.

Predicative specifications can be combined using the familiar operators of boolean theory, along with all the usual program combinators. The program combinators include sequencing ('.'), selection (**if-then-else**), repetition (**while-do**), and procedure call. The notation also has a **frame** construct. The specification **frame** $w \cdot P$ means that predicate P can change variables w, but no other variables; if the state consists of disjoint collections of variables w and ρ, then **frame** $w \cdot P$ is equivalent to $(P \wedge \rho' = \rho)$.

Unlike Z, predicative programming is a wide-spectrum language, and therefore is very well-suited to refinement. Further, the method is well-suited to specifying and reasoning about real-time, concurrent, and communicating systems. A variant theory, presented in [9], maintains all the useful laws and theorems of [8], but allows specification of intermediate states of a computation. This is useful in specifying concurrency, as well as for proving liveness properties.

In the modified theory, state variables are treated as functions of time. The value of variable x at time t is xt. An expression like $x + y$ is a function of time; its argument is distributed to its variable operands as follows: $(x + y)t = xt + yt$. Standard programming notations are defined as follows.

$$
\begin{aligned}
\mathbf{ok} \quad &= \quad t' = t \\
x := e \quad &= \quad t' = t + 1 \wedge xt' = et \wedge yt' = yt \wedge \ldots \\
P.\, Q \quad &= \quad \exists\, t'' : t \leq t'' \leq t' \cdot P[t''/t'] \wedge Q[t''/t] \\
\mathbf{if}\ b\ \mathbf{then}\ P\ \mathbf{else}\ Q \quad &= \quad bt \wedge P \vee \neg\, bt \wedge Q
\end{aligned}
$$

(Without loss of generality, it is assumed that an assignment takes 1 unit of time, and no other program takes time.) The notation $P[a/b]$ means "substitute a for b in P". Recursive calls in specifications are allowed: if a specification P is refined by specification S, then S can make recursive calls to P, providing that time is increased before the call.

To retain the look of the standard predicative notation, we use x for xt and x' for xt' when we do not need to talk about intermediate states.

2.1.1 Bunch notation

Bunches are used in [8] as a type system. A bunch is a collection of values, and can be written as in this example: $2,3,5$. Some bunches are worth naming, such as *null* (the empty bunch), *nat* (the natural numbers), *xnat* (the extended naturals, which include ∞), *int* (the integers), and so on. More interesting bunches can be written with the aid of the solution quantifier §, pronounced "those", as in the example $\S i : int \cdot i^2 = 4$. We use the asymmetric notation $m, ..n$ for $\S i : int \cdot m \leq i < n$.

Bunches can also be used as a type system, as in the declaration **var** $x : nat$ (perhaps with restrictions for easy implementation). More generally, $A : B$ is a boolean expression saying that A is a subbunch of B. For example,

$$2 : nat \qquad\qquad nat : int$$

We write functions in a standard way, as in the example $\lambda\, n : nat \cdot n + 1$. When the domain of a function is an initial segment of the natural numbers, we sometimes use a list notation, as in $[3;\ 5;\ 2;\ 5]$. The empty list is $[nil]$. We also use the asymmetric notation $[m; ..n]$ for a list of integers starting with m and ending before n. List length is $\#$, and list catenation is $^+$. By letting $list = \lambda\, T : \Delta list \cdot 0, ..\#(list\ T) \to T$ then $list\ T$ consists of all lists whose items are of type T.

2.1.2 Concurrency

Predicative programming includes notations for concurrent specification and for communication. Combined with the aforementioned notion of time, this allows for specification and refinement of real-time, concurrent, interactive systems.

The independent composition operator $||$ applied to specifications P and Q is defined so that $P \mathbin{||} Q$ (pronounced "P parallel Q") is satisfied by a machine that behaves according to P and at the same time, in parallel, according to Q. The formal meaning of $||$ is as follows. We first define *wait* as a specification whose execution takes an arbitrary amount of time and leaves all other variables σ unchanged during that time.

$$wait = t' \geq t \land \forall\, t'' : t \leq t'' \leq t' \cdot (\sigma t'' = \sigma t')$$

Independent composition can be defined as follows. Let v and w be bunches of variables, and P and Q predicative specifications. Then

$$(\textbf{frame}\ w \cdot P) \mathbin{||} (\textbf{frame}\ v \cdot Q) =$$
$$\textbf{frame}\ w \cdot P \land \textbf{frame}\ v \cdot (Q.\ wait) \lor \textbf{frame}\ w \cdot (P.\ wait) \land \textbf{frame}\ v \cdot Q$$

Informally, if P leaves a variable unchanged, then Q determines the final value, while if Q leaves a value unchanged, P determines its final value. The time for the independent composition is the maximum of the process times.

The $||$ operator obeys a collection of useful laws, e.g., symmetry, associativity, and distributivity. Independent compositions can also be refined by steps and by parts:

- **Refinement by Steps:** if $A \Leftarrow B \mathbin{||} C$ and $B \Leftarrow C$ and $C \Leftarrow E$ are theorems, then $A \Leftarrow D \mathbin{||} E$ is a theorem.
- **Refinement by Parts:** if $A \Leftarrow B \mathbin{||} C$ and $D \Leftarrow E \mathbin{||} F$ are theorems, then $A \land D \Leftarrow (B \land E) \mathbin{||} (C \land F)$ is a theorem.

2.1.3 Communication

Process communication is by any number of named channels. Communication on a channel c is described by two constant infinite lists M_c and T_c called the *message script* and *time script*, and two extended natural variables r_c and w_c called the *read cursor* and the *write cursor*. The message script is the list of all messages that pass along the channel, while the time script is the corresponding list of times that the messages were or are or will be sent. The read cursor is a state variable saying how many messages have been input on the channel; the write cursor is a state variable saying how many messages have been output on the channel.

Here is an example: it says that if the next input on channel c is even, then the next output on channel d will be $\top$, otherwise it will be $\bot$.

$$M_d w_d = even(M_c r_c)$$

Four programming notations are provided for communication. Let c be a channel.

$$\begin{array}{rcl} c? & = & r_c := r_c + 1 \\ c & = & M_c(r_c - 1) \\ c!e & = & M_c(w_c) = e \wedge T_c(w_c) = t \wedge (w_c := w_c + 1) \\ ?c & = & T_c(r_c) < t \end{array}$$

$c?$ specifies a computation that reads one input on channel c. The channel name c is used to denote the message that was last previously read on the channel. $c!e$ specifies a computation that writes message e on channel c. And $?c$ is a boolean expression that is true if and only if there is unread input available on channel c.

Channel declaration introduces a new channel within some local portion of a specification. A channel declaration applies to what follows it. The syntax and semantics of a channel declaration c applied to specification P is

$$\textbf{chan}\ c : T \cdot P \quad = \quad \exists M_c : list\ T \cdot \exists T_c : list\ xnat \cdot \textbf{var}\ r_c, w_c : xnat := 0 \cdot P$$

T is the type of communications on channel c. Time is of type extended natural, but could also be extended integer, rational, or real. The channel declaration also sets the read cursor r_c and write cursor w_c to initial value 0.

Here is an example of two concurrent processes communicating on channel c.

$$\textbf{chan}\ c : int \cdot c!2 \parallel (c?.\ x := c)$$

The process that puts 2 on the channel c can be executed in parallel with the process that reads an integer from channel c and assigns this value to x. Simplifying, using the definition of **chan**, we find that this specification is equivalent to $x := 2$, as expected.

By itself, predicative programming is useful and appropriate for specifying and refining concurrent, communicating systems. The issue in this paper is in terms of combining Z with predicative notation so that Z can also make use of these notions.

3 Approach to Heterogeneity

The approach to formally defining the meaning of heterogeneous notations that we use is from [11, 12]. Translations are defined between formal notations of interest. The

translations provide the mechanisms by which a heterogeneous specification can be given a formal semantics using a homogeneous specification, via mapping the original specification into a single-notation formulation. A set of notations and translations between them, which is to be used to give a formal semantics to heterogeneous specifications, is called a *heterogeneous basis*. The small heterogeneous basis that we use in this paper consists of the Z notation and the predicative notation, with translations between them. It is derived from a much larger basis given in [11]. We require only one translation in the basis, a mapping from Z to predicative notation.

To translate from a Z schema $Op == \left[\,\Delta S;\ i? : I;\ o! : O \mid P\,\right]$ to a predicative specification, we use the translation *ZToPP*, defined as follows.

$$ZToPP(Op) \mathrel{\widehat{=}} \mathbf{frame}\ w \cdot (\mathrm{pre}\, Op \Rightarrow P)$$

The frame w consists of the variables in S and the operation outputs. Two options exist for translating the inputs $i?$: they can be mapped to state variables (and *ZToPP* can be used unchanged); or they can be mapped to procedure parameters. In the latter case, Op would be translated to the predicative specification $procOp = \lambda\, i : I \cdot ZToPP(Op)$. Though *ZToPP* is written as a total function, we require that for any Op that includes a state schema by Δ convention, $P \neq true$, because predicative notation cannot describe terminating yet arbitrary computations [12].

In [14], detailed justifications for integrating notations—and, in particular, similar notations—are made.

3.1 Syntax of heterogeneous specifications

When integrating notations, both the syntax and the semantics of the separate languages must be reconciled. On the surface, semantic reconciliation seems to be the harder problem: in order to prove properties about the combined notation, we must give the combined notation a formal semantics, typically by translation. This process may be difficult, especially if the notations are very different and present radically different views and models of a system. But reconciling syntax need not be trivial either. If the notations, when combined, form a new notation with an ambiguous grammar, then changes in the syntax of one or both of the notations may be necessary.

Ambiguities in syntax arise when combining Z and predicative notation. In predicative notation, $\wedge$ and $\vee$ are applied to and produce predicates. In Z, $\wedge$ and $\vee$ are used as both predicate and schema operators. Thus, in a notation combined from predicative notation and Z, if we write the specification $S \wedge P$, we cannot tell whether $\wedge$ is an operator applied to predicates or to schemas.

We disambiguate the notations by using $\curlyvee$ and $\curlywedge$ for schema disjunction and schema conjunction, respectively.

3.2 Semantics of heterogeneous specifications

The translation *ZToPP* is the basis for formally defining the semantics of compositions of Z specifications and predicative specifications, by translating heterogeneous specifications into homogeneous specifications. In this paper, heterogeneous specifications

are given a semantics in terms of predicative notation. Therefore, we always write heterogeneous specifications under the assumption that Z partial specifications can be translated into predicative notation. This provides an *intersection semantics* to heterogeneous specifications [11]; it is so-called because the semantics of the new language is effectively the intersection of the separate languages.

To obtain the meaning of a heterogeneous specification, it must be explained how to translate the specification into predicative notation. The translation *ZToPP* is defined only on Z specifications, so we must extend it to the heterogeneous notation. The extension applies over the syntax tree of a heterogeneous specification. The reader is directed to [14] for full details. Since a heterogeneous notation has its semantics in predicative notation, predicative refinement can be strengthened to be applicable to heterogeneous specifications. See [14] for rules for refining heterogeneous specifications composed from Z and predicative specifications.

A useful implication of this semantics for heterogeneous specifications is that Z refinements are preserved under the translation. That is, if a Z specification *AOp* is refined by Z specification *COp*, using the standard definition of Z algorithm refinement [15], then *AOp* is also refined by *COp* using the standard predicative definition of refinement (applying the translation *ZToPP* behind-the-scenes).

Theorem 1. Let *AOp* and *COp* be Z specifications on state σ, and suppose that $AOp \sqsubseteq COp$, where $\sqsubseteq$ is Z operation refinement; thus, $\mathrm{pre}\, AOp \Rightarrow \mathrm{pre}\, COp$ and $\mathrm{pre}\, AOp \wedge COp \Rightarrow AOp$. Then $\forall \sigma, \sigma' \cdot (ZToPP(AOp) \Leftarrow ZToPP(COp))$.

Proof of Theorem 1. Suppose that $AOp \sqsubseteq COp$. Then

$$\begin{array}{lcl} AOp \sqsubseteq COp & = & \forall \sigma, \sigma' \cdot (\mathrm{pre}\, AOp \Rightarrow \mathrm{pre}\, COp) \wedge (\mathrm{pre}\, AOp \wedge COp \Rightarrow AOp) \\ & \Rightarrow & \forall \sigma, \sigma' \cdot (\mathrm{pre}\, AOp \Rightarrow AOp) \Leftarrow (\mathrm{pre}\, COp \Rightarrow COp) \\ & = & \forall \sigma, \sigma' \cdot ZToPP(AOp) \Leftarrow ZToPP(COp) \end{array}$$

An important corollary of Theorem 1 is that when refining a heterogeneous specification, Z refinement techniques can be applied to Z specifications, and this results in a predicative refinement of the heterogeneous specification. For example, let *AOp* and *COp* be Z specifications and *Q* a predicative specification, where $AOp \sqsubseteq COp$. Then it is a theorem that

$$AOp \parallel Q \quad \Leftarrow \quad COp \parallel Q$$

$AOp \sqsubseteq COp$ implies that $AOp \Leftarrow COp$, and by refinement by parts (Section 2.1.2), the result holds. More generally, sequencing can be replaced by any combinator over which predicative refinement is monotonic, and the result still holds.

In [6], Fischer suggests that a necessary condition for a useful combination of notations is that the combination preserves refinement, i.e., if *A* is refined by *B* in one of the original notations, then *A* is refined by *B* in the heterogeneous notation. The condition in [6] pertained to data refinement; Theorem 1 shows that notations can be combined so that the relationship holds for algorithm refinement as well.

The integration of Z and predicative notation that we have presented is simpler than integrations involving very different notations—e.g., Z and a process algebra [6]—because the notations present similar views of a system and have a similar notion

of state transition. The integrated notation also now possesses facilities for specifying and reasoning about concurrency and communication, as we now demonstrate. This demonstration aims to show that to talk about concurrency and communication with Z, it need not be necessary to integrate wildly different notations.

4 Concurrency

In this section, we discuss how to specify concurrent processes using the combination of Z and predicative notation. The process extends the definition of independent composition, $\|$, to heterogeneous specifications.

First, we show how $\|$ can be applied to Z schemas. Let $Op1$ and $Op2$, be schemas where each can be translated into predicative notation. They are as follows.

$$Op1 == [\,\Delta S;\ i? : I;\ o! : O \mid P\,] \qquad Op2 == [\,\Delta T;\ j? : J;\ k! : K \mid Q\,]$$

Then

$$\begin{aligned} Op1 \parallel Op2 &= ZToPP(Op1) \parallel ZToPP(Op2) \\ &= \textbf{frame}\ w \cdot (\text{pre}\, Op1 \Rightarrow P) \parallel \textbf{frame}\ x \cdot (\text{pre}\, Op2 \Rightarrow Q) \end{aligned}$$

Variables in state schemas S and T are considered to be translated to variables local to the independent composition, as are the schema outputs. Inputs can either be translated to local variables, or parameters of procedures.

The definition of independent composition of Z schemas has the limitations noted with the operator in [8]. In particular, it is not suited for passing values of variables via shared memory (for this, we will need communication, presented in Section 5).

Consider a simple example of a system with three natural number variables.

$$State == [\,x, y, z : \mathbb{N}\,]$$

There are two operation schemas, as follows.

$$\begin{aligned} Op1 &== [\,\Delta State \mid x' = z \wedge y' = y \wedge z' = z\,] \\ Op2 &== [\,\Delta State \mid x' = x \wedge y' = z \wedge z' = z\,] \end{aligned}$$

Then the parallel composition of these schema is:

$$\begin{aligned} Op1 \parallel Op2 &= ZToPP(Op1) \parallel ZToPP(Op2) \\ &= (x' = z \wedge y' = y \wedge z' = z) \parallel (x' = x \wedge y' = z \wedge z' = z) \\ &= x' = y' = z' = z \end{aligned}$$

Nothing has to be changed in our definition in order to be able to use time. Suppose, for example, that we changed our parallel composition to

$$(Op1 \wedge t' - t = 2) \parallel (Op2 \wedge t' - t = 1)$$

Then the semantics of this specification would be

$$\begin{aligned} &x(t+2) = zt \wedge y(t+2) = zt \wedge z(t+2) = zt \wedge \\ &x(t+1) = xt \wedge y(t+1) = zt \wedge z(t+1) = zt \end{aligned}$$

The intermediate states are shown in the semantics; the state at time $t + 1$ is due to $Op2$, while the state at time $t + 2$ is due to $Op1$.

An advantage of using a heterogeneous approach to concurrency with Z is that operands of $||$ may be written in Z or in predicative notation. This gives us the flexibility to use the most appropriate notation and refinement relation to specify and refine each process: for Z specifications, we can use Z refinement; for predicative specifications, we can use predicative refinement. It may also be advantageous to use predicative refinement on Z specifications, because predicative refinement is just boolean implication, which is simpler than Z refinement.

4.1 Example

We contrast using the combination of Z and predicative notation with using Evans' extension of Z [3], applied to a simple telecommunications protocol from [17]. We show that the heterogeneous notation can produce specifications and proofs that are comparable in size and complexity to those of Evans.

Evans' specification of the protocol is as follows. Let M be the set of messages that the protocol handles, and $State$ be the state schema for the protocol. *in* and *out* are state variables describing the incoming and outgoing messages, respectively. The invariant states that *out* is a suffix of *in*.

$$State == [\, in, out : \operatorname{seq} M \mid \exists s : \operatorname{seq} M \bullet in = s \frown out \,]$$

The valid initial states are specified by operation schema *Init*.

$$Init == [\, State \mid in = \langle\, \rangle \,]$$

Transmission and reception of messages is via the *Transmit* and *Receive* operations.

$$\begin{array}{l} \textit{Transmit} \\ \hline \Delta State \\ m? : M \\ \hline in' = \langle m? \rangle \frown in \\ out' = out \end{array}$$

$$\begin{array}{l} \textit{Receive} \\ \hline \Delta State \\ \hline in' = in \\ \#out' = \#out + 1 \lor out' = out \end{array}$$

This completes the traditional Z specification of the system. The dynamic specification augments the traditional one, to describe behaviour in terms of allowable sequences of state changes that result from execution of system operations. First, a next-state schema for the protocol is defined.

$$NextState == Transmit \curlyvee Receive$$

The dynamic behaviour of the protocol is specified by schema *ParBehaviour*.

$$\begin{array}{l} \textit{ParBehaviour} \\ \hline \sigma : \mathbb{N}_1 \to State \\ \hline \sigma \ \mathsf{validcomp}\ (\{Init \bullet \theta State\}, \{NextState \bullet \theta State \mapsto \theta State'\}) \\ \sigma \ \mathsf{wf}\ \{Receive \bullet \theta State \mapsto \theta State'\} \end{array}$$

validcomp is true for all state changes in which the first step belongs to the initial state of the system, and in which subsequent steps are related by the next-state relation. Nondeterministic selections are made on enabled state changes. Weak fairness, through wf is also specified. wf is true for any behaviour in which the set of state changes is always eventually executed. These operators are formally specified in [3].

Informally, the specification *ParBehaviour* captures the intuitive behaviour of the protocol: *Receive*s and *Transmit*s may happen in parallel. This is simulated by nondeterministic interleaving. Note that this specification does not guarantee progress.

The heterogeneous specification of the system reuses the Z specifications of *Receive* and *Transmit*. Dynamic behaviour is specified using independent composition.

$$Behaviour = (Receive \parallel Transmit).\ Behaviour$$

Informally, the system carries out *Receive* and *Transmit* in parallel indefinitely. Formally, the behaviour is specified as a fixed-point construction. Initialization can be specified by sequencing, i.e., $(Init'.\ Behaviour)$. In *Behaviour*, *Receive* has the option of doing nothing each time it is enabled. Thus, progress is not guaranteed, but it can be by constraining the *Receive* operation in exactly the same way as is done in [3]. To do this, the *Receive* operation is extended to $[Receive \mid out' \neq out]$. [13] discusses extending *Behaviour* to specify weak fairness.

4.2 Safety and liveness

In [3], Evans shows how to prove safety and liveness properties with the extension of Z, via a collection of new proof rules. We now briefly discuss how to carry out safety and liveness proofs with the heterogeneous notation, using our own formulations of UNITY-style rules.

We begin with safety properties. Let A be as follows.

$$A = (S_0 \parallel \ldots \parallel S_{k-1}).\ A$$

Each S_i may be a Z operation schema or a predicative specification. A is invariant with respect to a property P (which is a predicate on a state σ) if the independent composition preserves P, i.e.,

$$\forall\, t, t' \cdot ((S_0 \parallel \ldots \parallel S_{k-1}) \Rightarrow (P \Rightarrow P'))$$

where P' is identical to P but is in terms of the post-state σ'. In general, we cannot prove invariant properties by parts, because the processes S_i may interfere with each other (e.g., if S_i changes variable x, and S_j also changes x but to a different value). However, in the case where the **frame**s of all processes are disjoint, it will hold that

$$(S_0 \parallel \ldots \parallel S_{k-1}) \Rightarrow (S_0 \vee \ldots \vee S_{k-1}) \qquad (1)$$

and then we can prove the invariance of P by parts, by showing that

$$\forall\, i : 0,..k \cdot \forall\, t, t' \cdot S_i \Rightarrow (P \Rightarrow P')$$

If we can prove (1), then safety properties can be proven by parts. Partwise proof can also be abetted by following guidelines on the use of independent composition. In [8],

it is recommended not to write independent compositions where processes interfere with each other—i.e., with intersecting **frame**s—because it can lead to unsatisfiable specifications. Communication constructs can be used to avoid this difficulty.

In [3], proof of safety properties by parts is possible in general because parallelism is simulated by nondeterministic interleaving; $\|$ in this paper is approximately conjunction, thus requiring an extra satisfiability constraint (which is effectively (1)).

Consider the following example, showing that *Behaviour* satisfies the invariant $P = (\exists s \cdot in = s^{+}out)$. Because the processes of *Behaviour* change different variables, it suffices to show that each process maintains the invariant. Thus, we would need to show that *Receive* satisfies P. The proof obligation for this step is

$$\forall t, t' \cdot Receive \Rightarrow (P \Rightarrow P')$$

P is the state invariant. After applying *ZToPP* to *Receive*, we must prove

$$\forall t, t' \cdot (P \Rightarrow (in' = in \wedge \#out' = \#out + 1 \vee out' = out \wedge P')) \Rightarrow (P \Rightarrow P')$$

which is *true*. *Transmit* similarly satisfies the invariant P. And thus, so does *Behaviour*.

Liveness properties are more complex to prove. One useful liveness property, suggested in [3], is **leads-to**. P **leads-to** Q is informally defined as "if P is true then eventually an enabled operation will cause Q to become true". For a concurrent system like A, above, the formal meaning of P **leads-to** Q is

$$\forall t, t' \cdot P \Rightarrow (\exists t'' : t \leq t'' \leq t' \cdot (A \Rightarrow Q')[t''/t])$$

Informally, the rule expresses that if P holds at time t, then there is some time t'' in the course of steps of behaviour of the concurrent system A at which Q' is established.

Proving that a system satisfies a **leads-to** property using this definition may be complicated. If an inductive proof is not needed (i.e., P will **lead-to** Q after a single step in the computation), we must prove that

$$\forall t, t' \cdot P \Rightarrow \exists t'' : t \leq t'' \leq t' \cdot ((S_0 \parallel \ldots \parallel S_{k-1}) \Rightarrow Q')[t''/t]$$

because the processes S_i may change some of the same variables. If all processes have disjoint **frame**s, then condition (1) will hold, and the **leads-to** property can be proven by parts in much the same way as Evans [3].

There are two possible versions of the **leads-to** rule: one for weak fairness, the other strong fairness. We consider the former here. There are three main steps in a **leads-to** proof, providing that we have shown that the proof can be done by parts.

1. Show that each operation S_i in the system either leaves P invariant, or establishes the property Q.

   $$\forall i : 0, ..k \cdot \forall t, t' \cdot S_i \Rightarrow (P \Rightarrow P' \vee Q')$$

2. Show that P enables a weakly fair operation S_j, i.e., $\forall t \cdot P \Rightarrow (\text{pre}\, S_j)$.

3. Show that the weakly fair operation S_j establishes Q under assumption P.

   $$\forall t, t' \cdot P \Rightarrow (S_j \Rightarrow Q')$$

To establish the soundness of this rule, suppose that the three conditions hold (as will (1)). For any computation in which P holds initially, the first condition ensures that a valid step (where a step corresponds to the execution of an operation S_i) will either preserve P or establish Q'. By the second condition, the weakly fair operation is continuously enabled throughout the computation. As a consequence of the third rule and fairness, S_j will eventually be executed, resulting in Q' being established.

This rule will be insufficient for inductive proofs, wherein it is useful to introduce a variant that is decreased on each iteration. For such systems, we must show

$$(P \wedge N = n) \textbf{ leads}-\textbf{to } ((Q \vee N < n) \wedge P)$$

where N is a variant over a well-founded set.

For the system *Behaviour*, we might want to prove that, under a progress constraint,

$$(\#in > \#out \wedge \#in = k) \textbf{ leads}-\textbf{to } (\#out = k)$$

i.e., that if k messages are input, then eventually k messages are output. Assume that *Receive* is a weakly fair operation. First, we must constrain *Behaviour* so as to ensure eventual reception of messages. Due to the flexibility of the heterogeneous notation, this is easy to do. We simply modify *Receive* to $\left[Receive \mid out' \neq out \right]$.

We can use the inductive rule to prove liveness (because condition (1) holds, since the processes of *Behaviour* change different variables). Let a variant N be $k - \#out$, P be $\#in > \#out \wedge \#in = k$, let Q be $\#out = k$, and let I be the property of state schema *State*. First, we must show that the weakly fair operation is always enabled.

$$\forall\, t \cdot P \wedge I \wedge (k - \#out = n) \Rightarrow \operatorname{pre} Receive$$

Next, we show that each system operation either maintains P or establishes Q. The first part shows this for *Receive*; the similar obligation for *Transmit* is in [13].

$$\forall\, t, t' \cdot ZToPP(Receive) \Rightarrow ((P \wedge I \wedge k - \#out = n) \Rightarrow \\ P' \wedge k - \#out' = n \vee Q' \vee k - \#out' < n \wedge I')$$

Finally, we show that the operation establishes Q or decreases the variant.

$$\forall\, t, t' \cdot P \wedge I \wedge k - \#out = n \Rightarrow (ZToPP(Receive) \Rightarrow Q' \vee k - \#out' < n \wedge I')$$

The first obligation holds because $\#in > \#out$ implies the precondition of *Receive*. The second condition holds because *Receive* increases $\#out$, thus decreasing the variant. The last formula holds since *Receive* increases $\#out$ by 1, which either guarantees that Q' holds or that the variant is decreased.

Proving safety and liveness properties with the heterogeneous notation is more complex in general than with Evans' approach, in part because concurrency in the notation is effectively conjunction, as opposed to disjunction in [3]. In order to prove these properties by parts, an extra satisfiability proof obligation must be discharged. In general, safety and liveness properties will not be provable by parts with the heterogeneous notation. However, if we use the independent composition operator as suggested in [8] (i.e., avoid writing to shared memory) and instead make use of the communication operators presented in the next section, then a partwise approach to proof can be used. This allows us, effectively, to use the notation in much the same way as Evans' approach.

5 Communication

Now we consider input and output between processes. Input and output is by channels, through which a computation communicates with its environment. The computation may be specified in Z or in predicative notation or in a combination, perhaps via an independent composition. The channels are specified in predicative notation, as was discussed in Section 2.1. We illustrate the approach with two examples.

5.1 Example: mutual exclusion

The first example is a simple system involving mutual exclusion and synchronization. The problem is derived from one in [8]. We specify a concurrent queueing system with two processes. One process adds jobs to a shared queue, while a second process removes jobs and services them. We write a heterogeneous specification of the system (omitting details of how a job is to be serviced), and then write a specification expressing a mutual exclusive access property that the system must satisfy.

The specification commences by introducing a basic type *PROCESS*, to stand for the type of processes, as well as a *RESULT* type, to stand for an operation status output.

$$RESULT ::= SUCCESS \mid FAIL$$

The system state is as follows.

State

$queue : \mathrm{seq}_{\infty}\ PROCESS$
$numjobs : \mathbb{N}$

$numjobs = \# queue$

Operation *AddJob* places new jobs into the queue.

AddJob

$\Delta State$
$job? : PROCESS$

$numjobs' = numjobs + 1$
$queue' = queue \frown [job?]$

The operation that removes a job from the queue and services it is as follows (we ignore the details of the servicing).

ServiceJob

$\Delta State$
$result! : RESULT$

$numjobs > 0$
$queue' = tail(queue)$
$numjobs' = numjobs - 1$
$result! = SUCCESS$

RServiceJob

$\Xi State$
$result! : RESULT$

$numjobs = 0$
$result! = FAIL$

We now use the heterogeneous notation to specify the system. The specification is

$$\textbf{chan}\ a : int \cdot \textbf{chan}\ b : int \cdot P \parallel Q$$

where

$$\begin{array}{lcl} P & = & N_p.\ \ a!\top.\ \ AddJob.\ \ a!\bot.\ \ P \\ Q & = & N_q.\ \ b!\top.\ \ (ServiceJob \curlyvee RServiceJob).\ \ b!\top.\ \ Q \end{array}$$

N_p is a specification that performs some initialization for *AddJob*; N_q performs initialization for the service operations. To ensure that mutual exclusion is guaranteed, the specification must satisfy

$$\begin{array}{r} \neg\ \exists i : w_a, ..\infty \cdot \exists j : w_b, ..\infty \cdot (M_a i \wedge T_a i \leq T_b j < T_a(i+1)) \vee \\ (M_b j \wedge T_b j \leq T_a i < T_b(j+1)) \end{array}$$

The condition above states that a message does not arrive on channel a at the same time as a message on channel b (and vice versa).

The specification can be refined by parts: Z refinement can be applied to *AddJob* and *ServiceJob*; predicative refinement can be applied to the remaining portions.

5.2 Example: short-term scheduler

We now present a more detailed example that combines use of concurrency, communication, and refinement. 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 first generate test data, and second, simulate either a first-come first-served (FCFS) or a round-robin (RR) scheduler. The generator part produces two vectors of data, one holding *NUMBER* random CPU burst lengths, and the other holding *NUMBER* random arrival times. Bursts should be generated so that 80% of burst lengths are uniformly distributed between 0.1 and 1.0, with the remaining 20% between 1.0 and 10.0. The arrival times of the processes must have a Poisson distribution, with parameter *LAMBDA*. The second component simulates the selected algorithm on the test data, starting with the circular "ready" queue (length 100) holding *INITIAL* jobs. Total and average wait times should be output upon completion.

A rough sketch of the system is in Fig. 1.

In Fig. 1, circles are processes, rectangles are external entities in the environment, and parallel lines are data stores. Data flow between processes, entities, and data stores is written using arrows. To formally specify the system, we use predicative notation for its strengths: for specifying communication and concurrency, and for specifying iterative details. We use Z for everything else, i.e., individual processes. Before doing so, we specify the state of the system, and formalize the terms used in Fig. 1.

INITIAL, *LAMBDA*, and *NUMBER* were described in the informal requirements.

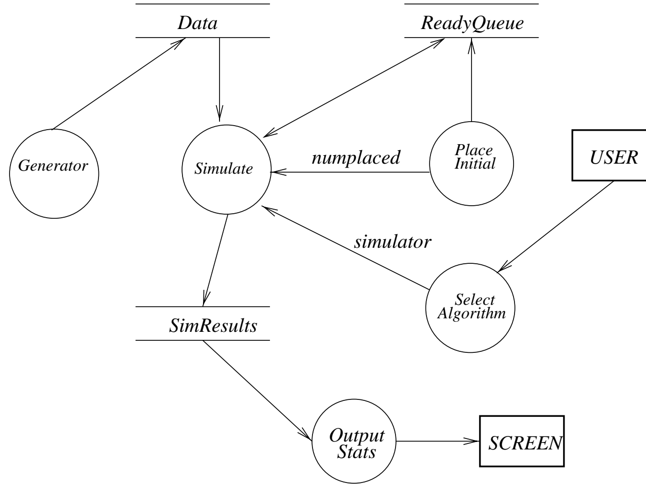

Fig. 1: Rough sketch of simulator system

$$
\begin{array}{|l}
INITIAL, NUMBER : \mathbb{N} \\
LAMBDA : \mathbb{R}
\end{array}
$$

The system's *ready queue* holds the processes that are to be serviced. The queue is modeled as a sequence of *Cell*s.

Cell

$burstlength, arrivaltime : \mathbb{R}$
$group : \mathbb{Z}$

ReadyQueue

$ready : \mathrm{seq}_{100}\, Cell$
$arriving, head, tail, length : \mathbb{N}$

The *ReadyQueue* is made up of a sequence of *Cell*s, as well as the pointers necessary to maintain and update the queue (i.e., *tail* and *head* pointers). Finally, the data store used to hold the data generated and used by the simulator is specified as a state schema.

$$Data == [\, bursts, arrivals : \mathrm{seq}_{NUMBER}\, \mathbb{R} \,]$$

Available simulation algorithms are specified as a type: $ALGTYPE ::= rr \mid fcfs$.

At least two channels appear to be necessary for this system: one between processes *PlaceInitial* and *Simulate*, and another between *SelectAlgorithm* and *Simulate*. Fig. 1 suggests names for these channels: *numplaced*, and *simulator*, respectively. These are specified in predicative notation.

$$\textbf{chan}\ numplaced : nat \cdot \textbf{chan}\ simulator : ALGTYPE$$

We now provide specifications of selected processes, concentrating on the most interesting: those for the generator and the simulator. The purpose of the *Simulate* process is to read values on its channels and then simulate a scheduler on the data generated by the remaining parts of the system. We write this as a heterogeneous specification.

$$
\begin{aligned}
Simulate \;\; = \;\; & numplaced? \parallel simulator?.\\
& time, arriving, current := arrivals(length), numplaced, bursts(0).\\
& \textbf{while}\ (0 < length < 100)\ \textbf{do}\ (\\
& \quad \textbf{if}\ (simulator = fcfs)\ \textbf{then}\ FCFS\ \textbf{else}\ RR.\\
& \quad current := ready(head).\ dequeue)
\end{aligned}
$$

FCFS and *RR* specify the behaviour of the first-come first-served and round-robin schedulers, respectively. Predicative notation is well-suited for specifying the iterative parts of the simulator, because it is a wide-spectrum language. The *FCFS* schema is as follows.

$$
\begin{array}{|l}
\hline
\textit{FCFS} \\
\Delta ReadyQueue \\
marr : \mathbb{N} \\
\hline
marr = max\{j : arriving..NUMBER - 1 \mid time > arrivals(j)\} \\
head \neq (tail + marr - arriving) \bmod 100 \\
\forall\, i : 0..marr - 1 \bullet \exists\, new : Cell \bullet \\
\quad new.burstlength = bursts(arriving + i) \\
\quad new.arrivaltime = arrivals(arriving + i) - arrivals(INITIAL) \\
\quad new.group = \lfloor bursts(i) \rfloor \\
\quad ready'((tail + i) \bmod 100) = new \\
tail' = (tail + marr - arriving) \bmod 100 \\
length' = length + (marr - arriving) \\
arriving' = marr \\
\hline
\end{array}
$$

(The schema *RR* is similar.) Informally, the operation queues all those jobs that would have arrived during the service of the current job. *marr* is the maximum (last) job to arrive during the service of the current job.

Before simulation can begin, data should be generated, and the system must be initialized. Initialization involves placing *numplaced* data items in the queue and selecting a scheduling algorithm before simulation.

$$
\begin{aligned}
Generator.\ (&\textbf{chan}\ numplaced : nat \cdot \textbf{chan}\ simulator : rr, fcfs \cdot \\
& (PlaceInitial \parallel SelectAlgorithm).\ Simulate).\ OutputStats
\end{aligned}
$$

Data is generated using a random number generator, *rand*, which returns a random *real*. The *Generator* schema is as follows.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{Generator}} \\
\hline
\Delta \textit{Data} \\
\hline
\textit{arrivals}'(0) = 0 \\
(\textit{bursts}'(0) = 0.9 \times \textit{rand} + 0.1 \lor \textit{bursts}'(0) = 9.0 \times \textit{rand} + 1.0) \\
\forall\, i : 1..\textit{NUMBER} - 1 \bullet \\
\quad (\textit{bursts}'(i) = 0.9 \times \textit{rand} + 0.1 \lor \textit{bursts}'(i) = 9.0 \times \textit{rand} + 1.0) \\
\quad (\textit{arrivals}'(i) - \textit{arrivals}'(i-1) = -\textit{LAMBDA} \times \log_e\ \textit{rand}) \\
\hline
\end{array}
$$

The operation calculates burst and arrival times for *NUMBER* jobs, where a burst time is either between 0 and 1, or between 9 and 10. Arrival times have a Poisson distribution with parameter *LAMBDA*.

OutputStats might be trivially formalized as $\textit{OutputStats} \;=\; \textit{Screen}!\textit{data}$ where *Screen* is a declared channel and *data* the simulation data. Similarly, *SelectAlgorithm* might be

$$\textit{SelectAlgorithm} \quad = \quad \textbf{while}\ \neg\ ?\textit{User}\ \textbf{do}\ \textbf{ok}.\ \textit{User}?.\ \textit{simulator}!\textit{User}$$

where *User* is a declared channel.

Now (safety) refinement can occur. We omit most of the details, since they can be found elsewhere [13]. The *Generator* specification can be implemented as a simple loop, using standard Z refinement techniques and refinement by parts (Section 2.2.1). This can occur due to Theorem 1. The guard on the loop implementing the generator is $i < \textit{NUMBER}$, a loop variant is $\textit{NUMBER} - i$, and a loop invariant is:

$$
\begin{array}{r}
1 \leq i < \textit{NUMBER} \land \\
\forall\, j : 1, ..i \cdot \textit{arrivals}(j) - \textit{arrivals}(j-1) = -\textit{LAMBDA} \times \log_e \textit{rand} \land \\
(\textit{bursts}(j) = 0.9 \times \textit{rand} + 0.1 \lor \textit{bursts}(j) = 9.0 \times \textit{rand} + 1.0)
\end{array}
$$

The first step of the refinement is to introduce a local variable, n, and to split the *Generator* specification into a leading assignment and a loop partial specification. The second step is to refine the loop to a loop body, where the body is a collection of assignment statements. The proof obligations are standard from [18]. The result of the refinement is the following program:

$$
\begin{array}{l}
n, \textit{arrivals}(0), i := \textit{rand}, 0, 1; \\
\textit{bursts}(0) := \textbf{if}\ (n \leq 0.8)\ \textbf{then}\ 0.9 \times \textit{rand} + 0.1\ \textbf{else}\ 9.0 \times \textit{rand} + 1; \\
\textbf{do}\ (i \leq \textit{NUMBER}) \rightarrow \\
\quad n := \textit{rand};\ \ \textit{bursts}(i), \textit{arrivals}(i), i := \\
\qquad \textbf{if}\ (n \leq 0.8)\ \textbf{then}\ 0.9 \times \textit{rand} + .1\ \textbf{else}\ 9.0 \times \textit{rand} + 1, \\
\qquad -\textit{LAMBDA} \times \log_e(\textit{rand}) + \textit{arrivals}(i-1), \\
\qquad i + 1 \\
\textbf{od}
\end{array}
$$

The Z schema *FCFS* can also be refined using the standard Z refinement rules. To carry out this refinement, we first define a standard queueing procedure *enqueue*. Refinement is done by first noticing that in *FCFS*, the local variable *new* as well as the

body of the universal quantifier, can be replaced by an *enqueue* operation. We also notice that the purpose of the universal quantifier is to add all arriving processes to the ready queue, providing that such an addition does not exceed the queue size. This can be refined to a loop: the loop terminates if the queue is full ($length = 100$), if there are no more waiting processes, or if all processes that arrived during servicing of the current job have been queued.

$$
\begin{array}{l}
\textbf{do } b \rightarrow \\
\quad enqueue(bursts(arriving), arrivals(arriving) - arrivals(INITIAL), \lfloor bursts(arriving) \rfloor); \\
\quad arriving := arriving + 1 \\
\textbf{od}
\end{array}
$$

where b is

$$time > arrivals(arriving) \wedge arriving < NUMBER \wedge length \neq 100$$

6 Discussion

The aim of integrating Z with predicative notation is to construct a notation that is suitable for specification and design of concurrent and communicating systems. It is not our aim to intentionally produce a notation that will be appropriate for other tasks. We therefore chose to combine Z with predicative notation, because predicative notation is a sufficient partner for Z in the tasks that we want to carry out. If we had wanted to carry out tasks beyond specification and reasoning about concurrency and communication, combining Z with a different notation would be appropriate.

In [3], Evans suggests a number of disadvantages to integrating Z with notations—like CSP, TLA, or CCS—that may be better suited to specifying concurrent behaviour: reconciling the semantics of the individual notations; using existing Z tools; and, poor use of the Z proof system. The integration of Z with predicative notation that we have presented in this paper addresses these limitations, as we now discuss.

Reconciling the semantics of separate notations can be difficult, especially for very different notations (though see [16]). But predicative notation and Z can be used to present the same view of a system. Therefore, combining these notations is simpler than, say, combining Z and CSP. Since predicative notation provides the techniques that we require—for reasoning about concurrency and communication—it does not appear to be necessary to consider alternative integrations for these purposes.

With an integration of Z with CSP, the ability to use existing Z or CSP tools with the new notation will be reduced or removed. With the integration of predicative notation with Z in this paper, the ability to use Z tools remains, at least with respect to Z partial specifications, because of the result that says a valid Z refinement implies a valid predicative refinement; Fischer [6] previously suggested the value of this capability. Some translation work may have to be done to heterogeneous specifications in order to get information regarding the system state needed to use the Z tools, but because our integration of notations is both syntactical and semantical, this is possible and relatively straightforward. With a purely semantic integration of notations, e.g., [5], this is not possible.

By using predicative notation as the semantic basis for the heterogeneous notation, we allow ourselves to use theorem provers based on typed set theory to support reasoning. So, for example, PVS can be used to support the predicative—and therefore the heterogeneous—notation. This suggests the need for a tool that will automatically translate heterogeneous specifications into PVS.

We have shown that the integrated notation can be used to carry out the tasks that are possible using Evans' extension. However, the approach has a number of advantages over Evans' work. A theoretical advantage is that miracles can be expressed in predicative programming, unlike in Z; this has been suggested as useful for simplifying data refinement. Predicative programming has built-in mechanisms for talking about time, space, and communication; we can use communication constructs and talk about real-time in the integrated notation without any further work. Evans' approach needs to be further extended in order to use such mechanisms. Further, it is easier to carry out refinement in predicative programming, due to its wide-spectrum nature. When we want to carry out refinement, we can use the predicative subset of the integrated notation. However, Evans' approach does produce some simpler proof rules (e.g., for liveness) than seem to be possible with predicative programming.

Our suggestion is not that the aforementioned problems with approaches that combine very different notations—like Z and CSP—will not arise. Rather, by carefully choosing compatible notations—like Z and predicative notation—and by understanding the roles each notation will play in the integration, the complications may prove to be less than critical.

7 Conclusions

In this paper, it has been shown how the Z notation can be used, in combination with the predicative notation of [8], to specify and reason about concurrent, real-time, communicating behaviours. A motivation for the work was to attempt to demonstrate that limitations noted with previous integrations [2, 4, 7] could be partially alleviated by integrating Z with the right notation. A second motivation was to demonstrate that it is not necessary to integrate Z with a very different notation, like a process algebra, in order to talk about concurrency. It was demonstrated that Z and predicative notation could be integrated so that the semantics of the notation allows practically full use of Z and maintains the ability to use Z proof techniques and tools on Z partial specifications. The need is created for a framework that allows combined use of existing Z tools, and prover tools like PVS.

The approach was aimed at showing how heterogeneous specifications could be used for concurrency and communication. An important aspect of this work is that it shows that it need not be necessary to extend Z in order to discuss concurrent, real-time, or communicating behaviour. Therefore, the standard Z notation can be used, augmented with predicative specifications that are well-suited to talking about such behaviours.

Acknowledgements. This research was carried out with the assistance of the National Sciences and Engineering Research Council of Canada.

References

[1] K.M. Chandy and J. Misra, *Parallel Program Design*, Addison-Wesley, 1988.

[2] R. Duke and G. Smith, Temporal logic and Z specifications. *Australian Computer Journal,* 21(2), May 1989.

[3] A.S. Evans, A case study in specifying, verifying, and refining a parallel system in Z. To appear in *Parallel Processing Letters.*

[4] E. Fergus and D. Ince, Z specifications and modal logic. *Proc. Software Engineering 90*, Cambridge, 1990.

[5] C. Fischer, CSP-OZ: a combination of Object-Z and CSP. In *Proc. FMOODS '97*, Chapman and Hall, 1997.

[6] C. Fischer, How to combine Z with a process algebra. In *Proc. ZUM '98*, LNCS 1493, Springer-Verlag, 1998.

[7] R. Gotzhein, Specifying open distributed systems with Z. In *Proc. VDM'90*, LNCS 428, Springer-Verlag, 1990.

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

[9] E.C.R. Hehner, Abstractions of time. In *A Classical Mind: Essays in Honour of C.A.R. Hoare,* Prentice-Hall, 1994.

[10] L. Lamport, TLZ. In *Proc. ZUM '94*, Springer-Verlag, 1994.

[11] R.F. Paige, A meta-method for formal method integration. In *Proc. Formal Methods Europe '97*, LNCS 1313, Springer-Verlag, 1997.

[12] R.F. Paige, Comparing extended Z with a heterogeneous notation for reasoning about time and space. In *Proc. ZUM '98*, LNCS 1493, Springer-Verlag, 1998.

[13] R.F. Paige, Specification and refinement using a heterogeneous notation for concurrency and communication. Technical Report CS-98-07, York University, October 1998.

[14] R.F. Paige, Heterogeneous notations for pure formal method integration. To appear in *Formal Aspects of Computing,* 1999.

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

[16] G. Smith, A semantic integration of Object-Z and CSP for the specification of concurrent systems. In *Proc. FME '97*, LNCS 1313, Springer-Verlag, 1997.

[17] J. Woodcock and J. Davies, *Using Z*, Prentice-Hall, 1996.

[18] J.B. Wordsworth, *Software Development with Z*, Addison-Wesley, 1992.