Specification and Refinement using a Heterogeneous Notation for Real-Time, Concurrency, and Communication

Richard Paige

Technical Report CS-1998-07

October 29, 1998

Department of Computer Science

4700 Keele Street North York, Ontario M3J 1P3 Canada

SPECIFICATION AND REFINEMENT USING A HETEROGENEOUS NOTATION FOR REAL-TIME, CONCURRENCY, AND COMMUNICATION

Richard F. Paige

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

Abstract. It is shown how to combine the Z formal specification notation [18] with a predicative notation [11], so as to be able to specify and reason about real-time, concurrency, and communication. The integration is carried out so as to alleviate some of the deficiencies noted with these approaches [5], such as the inability to use Z proof rules and tools. We demonstrate how to carry out refinement in a number of small examples of writing and refining heterogeneous specifications of concurrency and communication.

1 Introduction

The Z notation [18] 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 techniques, to concurrent systems [5].
2. *Integration approaches*, in which the Z notation is combined with notations that are considered better suited to specifying and reasoning about time or concurrency, e.g., temporal logic, TLA, or CCS [4, 8, 15]

An advantage claimed with extension approaches is that compatibility with existing Z proof techniques and proof tools can be maintained. A disadvantage claimed of integration approaches is that they may have difficulty reconciling the semantics of the separate notations [5]. This can result in problems of compatibility with the integrated notation and existing Z tools and proof rules.

A claim of this paper is that some of the limitations with integration approaches can be resolved, especially in a setting where refinement is to be carried out. Limitations can be alleviated, providing that Z is combined with an appropriate compatible notation, and the notation semantics are reconciled in a simple way. Our suggestion is that preferences of an extension of Z over an integration involving Z are not as cut-and-dried as they may seem, and that an integration of Z with a notation well-suited to specifying and reasoning about timing, concurrency, and communication can occur. With such an integration, specifications and reasoning techniques that are comparable to those produced by the method of [5] can be produced.

In this paper, we present a simple heterogeneous notation, combining Z with predicative notation [11], for specifying and reasoning about real-time, 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 and refinement case studies. Our emphasis in the integration is on indicating how *algorithm refinement* with heterogeneous specifications can occur.

1.1 Organization of the Paper

We commence with an overview of previous work, concentrating on the approach of Evans [5]. We also provide a brief overview of predicative notation. We next explain our approach to integrating notations, and describe how refinement and proof can occur on heterogeneous notations. This section includes a result which shows that in the heterogeneous notation, Z specifications can be refined using standard Z refinement laws. Section 4 recounts a technique for using Z and predicative notation together for reasoning about real-time. Section 5 explains how to specify concurrency, and presents an example of refinement. Section 6 extends concurrent 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 [4], Fergus and Ince [6], Gotzhein [8], Evans [5], and Lamport [15].

The first three integration approaches propose the use of temporal logic in proving safety and liveness properties of Z specifications, which 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, though derived inference rules for inferring temporal properties of Z schemas are typically not produced. Operational styles of reasoning are used to prove properties by directly examining histories, an approach which is suggested as impractical for all but the smallest of specifications.

Lamport [15] has suggested an alternative integration approach to concurrency, by integrating Z with TLA. In this approach, schemas are interpreted as actions, thus allowing use of TLA's assertional inference rules to verify safety and liveness properties. With this approach, temporal logic operators must still be added to Z, and existing Z proof tools cannot be used directly.

Evans' work [5] has instead focused on the direct application of Z to specifying concurrent systems. Evans' approach augments traditional Z specifications with an additional specification describing the system's dynamic behaviour (i.e., an 'external view' of the system), given in terms of allowable sequences of state changes. Evans produces proof rules which have been used to verify safety and liveness properties of specifications. The standard Z proof rules have been strengthened to ensure preservation of safety and liveness. The important goal of Evans' approach is to maintain compatibility with existing Z proof techniques, and existing Z proof tools.

In [17], it is shown how to combine Z and predicative programming in order to specify and reason about real-time and space. It is also demonstrated that the heterogeneous notation can be used to produce simpler, more concise refinements than with an extended dialect of Z.

2.1 Predicative programming

Predicative programming is due to Hehner [11]. It is a program design calculus in which programs are specifications. In this approach, programs and specifications are predicates on pre- and poststate (final values of variables are annotated with a prime; initial values of variables are undecorated). The weakest predicate specification is $\top$ ("true"), and the strongest specification is $\bot$ ("false"). Refinement is just 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 [11] 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 $(P \wedge \rho' = \rho)$.

One particular novelty with predicative programming is that *recursive* programs can be developed rather than iterative programs, using recursive refinement rules. For example, in a refinement step $S \Leftarrow P$ (where S and P are specifications), specification P can refer to S; this is a recursive call. It has been suggested that this simplifies the process of developing certain programs [11], since in particular it eliminates the need to construct invariants *before* developing loops.

Predicative programming, as defined in its standard reference [11], is well-suited to specifying and reasoning about real-time, concurrent, and communicating systems. A variant theory, presented in [12], maintains all the useful laws and theorems of [11], but allows specification of intermediate states of a computation. This is useful in talking about concurrent systems, and properties like liveness.

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.

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

(Here, it is assumed that an assignment takes 1 unit of time, and no other program takes time. This could be generalized.) Recursive calls are allowed, providing that time is accounted for before the call. So, 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-and-feel of the original predicative notation, we follow the suggestion of [9] and 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 [11] 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).

$A : B$ 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 $list = \lambda\, T : \Delta list \cdot 0, ..\#(list\ T) \rightarrow T$ 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. See [11, 12] for a number of detailed examples.

In predicative programming, the independent composition operator $\|$ applied to specifications P and Q is defined so that $P \parallel Q$ (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 unchanged during that time.

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

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

$$\begin{aligned} &(\textbf{frame } w \cdot P) \parallel (\textbf{frame } v \cdot Q) = \\ &\quad \textbf{frame } w \cdot P \wedge \textbf{frame } v \cdot (Q.\ wait) \vee \textbf{frame } w \cdot (P.\ wait) \wedge \textbf{frame } x \cdot Q \end{aligned}$$

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.

2.1.3 Communication Communication between processes 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.

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

$c?$ 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 (it is sometimes pronounced "probe 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 convenient predicative notions.

3 Approach to Heterogeneity

The approach to formally defining the meaning of heterogeneous notations that we use is from [16, 17]. 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 the notations. It is derived from a much larger basis given in [16]. We require only one translation in the basis, a mapping from Z to predicative notation.

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

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

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 [10].

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. But by changing the syntax of the notations, we may make the combined notation more unattractive to use, especially by specifiers or developers who are very familiar with one of the changed notations. Therefore, it is in our best interests to minimize syntactic changes to notations.

Ambiguity of a syntax for a heterogeneous notation appears when we combine Z and predicative notation. In predicative notation, the operators $\wedge$ and $\vee$ have the usual meaning: they are applied to predicates and produce a predicate. In Z, $\wedge$ and $\vee$ are overloaded: when used within the property of a schema, they are applied to predicates. When used within the schema calculus, they are applied to schemas. One implication of this overloading is that when, in a notation combined from predicative notation and Z, we write the specification $S \wedge P$ (where, e.g., S is a Z schema and P a predicative specification), we cannot tell whether $\wedge$ is an operator applied to predicates or to schemas. To resolve this ambiguity, we can either restrict use of the ambiguous operators in some way, or change the syntax of one of the ambiguous operators, or try to let the context indicate the meaning of the operator. Use of context, however, has negative implications with respect to building tool support for heterogeneous notations.

We disambiguate the notations by using $\curlyvee$ and $\curlywedge$ for the schema disjunction and schema conjunction operators, respectively. In this fashion, we follow the approach of [20], which showed how to schema conjoin and disjoin specification statements.

3.2 Semantics of heterogeneous specifications

The translation *ZToPP* is used to formally define 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 [16]; 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. More formally, we define the extension $ZToPP_e$ as follows. First, let $\circ$ be a combinator and z a specification, where both are in predicative notation. Let x be a Z specification. Then

$$ZToPP_e(x \circ z) \mathrel{\widehat{=}} ZToPP(x) \circ z \qquad ZToPP_e(x) \mathrel{\widehat{=}} ZToPP(x)$$

(a similar definition holds for $ZToPP_e(z \circ x)$). If z is also a Z specification, then

$$ZToPP_e(x \circ z) \mathrel{\widehat{=}} ZToPP(x) \circ ZToPP(z).$$

If the combinator $\circ$ happens to be a Z combinator, then

$$ZToPP_e(x \circ z) \mathrel{\widehat{=}} ZToPP(x \circ PPToZ(z)),$$

where *PPToZ* is a translation from predicative notation to Z (one may be found in [16]. For convenience, we will use *ZToPP* for $ZToPP_e$ from here on, and will let the context suggest how the function is to be used. When writing heterogeneous specifications, we will primarily use combinators from predicative notation, because we are interested in refinement, and because we want to retain all the refinement rules of predicative notation in the integrated notation.

To make this more concrete, consider an example where X is a Z schema and P a predicate. Applying *ZToPP* to $X \wedge P$ results in $ZToPP(X) \wedge P$. Applying *ZToPP* to $X \curlyvee P$ results in $ZToPP(X \curlyvee PPToZ(P))$.

Some rules for refining heterogeneous specifications composed from Z and predicative specifications were given in [16, 17]. The first explains how to refine Z operation schemas via predicative refinement.

Rule 0. For prestate σ and poststate σ', an operation schema with property P is refined by an operation schema with property Q if

$$\forall \sigma, \sigma' \cdot (\exists \sigma' \cdot P \Rightarrow P) \Leftarrow (\exists \sigma' \cdot Q \Rightarrow Q)$$

A useful rule is the following. Informally, the rule states the conditions to be checked in order to refine a schema by a predicative specification. Such refinements occur when algorithmically refining heterogeneous specifications that include time bounds.

Rule 1. For a prestate σ and poststate σ', a Z schema with property P is refined by a predicative specification Q if

$$\forall \sigma, \sigma' \cdot ((\exists \sigma' \cdot P) \Rightarrow P) \Leftarrow Q$$

A generalization of the substitution rule of [11] appeared in [17]. It simplifies the definition of sequencing in the case where the first operand is an assignment statement. Informally, it means that when we first assign value E to variable x, then behave like schema S, it is the same as behaving like new schema $S'[E/x]$ (where the primed schema is defined below).

Rule 2. Let x be a variable and E an expression, where '.' is predicative sequencing. If S is a schema with property P, then

$$(x := E.\ S) = S'[E/x]$$

where S' is the same as S except with property pre $S \Rightarrow P$. Read $S'[E/x]$ as "substitute E for x in the property of schema S'". σ is the state of the system.

Rule 2 means that we can apply the predicative substitution rule when using Z schemas in refinements. The substitution rule of [11] simplifies reasoning with sequences of assignments. The property of S' is changed from S under the substitution due to the translation of Z into predicative notation: Z conjoins a precondition with a postcondition, while predicative programming combines the two parts with an implication. In the special case where the implicit precondition of S is *true*, then S' can be replaced by S.

A useful implication of using predicative notation to give a semantics to heterogeneous specifications (and of refining heterogeneous specifications with predicative refinement) 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 [18], then *AOp* is also refined by *COp* using the standard predicative definition of refinement (applying the translation *ZToPP* behind-the-scenes).

Theorem. Let *AOp* and *COp* be Z specifications, and suppose that $AOp \sqsubseteq COp$, where $\sqsubseteq$ is Z operation refinement, i.e.,

$$\begin{aligned} \text{pre}\, AOp &\vdash \text{pre}\, COp \\ \text{pre}\, AOp \wedge COp &\vdash AOp \end{aligned}$$

Then $\forall \sigma, \sigma' \cdot (AOp \Leftarrow COp)$.

Proof. Suppose that $AOp \sqsubseteq COp$. Then

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

An important implication 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

$$AOp.\ Q \Leftarrow COp.\ Q$$

AOp $\sqsubseteq$ *COp* implies that *AOp* $\Leftarrow$ *COp*, and by refinement by parts, the result holds. More generally, sequential composition can be replaced by any combinator over which predicative refinement is monotonic (e.g., concurrent composition), and the result still holds. Though the theorem says that we can use Z refinement rules in our specific heterogeneous setting, it can be more useful to use the predicative notion of refinement, as we shall see in Section 4, to produce shorter proofs.

3.3 Unintrusive Integrations

The integration of Z with predicative programming that we have presented is claimed to be *unintrusive*. Unintrusive notation integrations, and unintrusive method integrations [16] should have at least the following (informal) properties.

1. *The integration does not change the specification style, syntax, or semantics unduly.* Specifiers can use most, if not all of the familiar idioms and concepts of the separate notations. They can write their specifications in a similar style. The meaning of the specifications that they write has a similar, if not identical meaning to those written in the separate notations. And the specifications that they write are of a size proportional to those of the separate notations.
2. *The integration does not change the refinement or proof techniques associated with the separate notations unduly.* Similar, if not identical, refinement laws apply to the integrated notation as the separate notations. The laws are made no more complex to use. Proof techniques that were applicable with the separate notations, e.g., refinement by parts, by steps, by cases, can be applied to the integrated notation.
3. *The integration allows use of the mechanized support provided by the separate notations.* Note that this does not require that tools of separate notations be applicable directly to specifications written in the heterogeneous notation, only that the tools be applicable to the parts of the specification written in the separate notations.

Our suggestion is that if a notation integration is unintrusive, then the cost of integrating the notations can be justified against the cost of extending a notation. Moreover, if the integration is unintrusive, then users of the separate notations may be more comfortable in using the combined notation; unintrusive integrations can therefore smooth the passage of adoption of new notations or methods.

The integration of Z and predicative programming is unintrusive because it satisfies the properties above. The specification styles of predicative notation and Z remain, for the most part, the same. One change is made to Z (with respect to the combinators $\vee$ and $\wedge$) in order to obtain syntactic compatibility. The refinement techniques for Z are applicable to the Z partial specifications that occur in a heterogeneous specification. And, therefore, Z tools, e.g., Cadiz, can be used to support the Z partial specifications.

4 Real Time

In this section, we outline how to use the combination of Z and predicative programming to talk about real time. Recall that our primary interest is specification and refinement; therefore, we will concentrate on explaining how to write real-time heterogeneous specifications, and on how to refine them algorithmically.

An integration of Z and predicative programming was presented in [17] which demonstrated how to write heterogeneous real-time specifications, and how to refine them to code. The basic approach was very simple: Z was used to write behavioural specifications, and predicative notation for writing timing specifications. The two specifications were then conjoined, and predicative refinement rules were used to refine the specification to code, following the predicative method. We summarize the approach with an example, omitting the details.

The problem is to calculate the maximum of a list of natural numbers and to do so in time proportional to the length of the list. The system state is

$$State \mathrel{\widehat=} \left[L : 0..n-1 \to \mathbb{N};\ j : \mathbb{N};\ r : \mathbb{Z}_\infty \right]$$

where L is the list, n its length, r the result of the computation, and j a counter. $\mathbb{Z}_\infty$ is all integers plus the new symbol ∞, which is larger than any integer.

The time constraints we are under are as follows. An assignment takes between 2-3 units of time, inclusive, and a tail-recursive call takes between 5-9 units of time, inclusive. No other statements take any time. The initial problem specification is $S \wedge Time$, where

$$S \mathrel{\widehat=} \left[\Delta State \mid L' = L \wedge r' = max\{j : 0..n-1 \bullet Lj\} \right]$$

and $Time \mathrel{\widehat=} t' - t : (7, ..14) \times n - (5, ..12)$. We use bunch notation [11] to express the nondeterministic timing constraints.

The timing requirement for an assignment statement indicates that $x := e$ means

$$t' - t : 2, ..4 \land x' = e \land y' = y \land \ldots$$

The refinement proceeds as follows. We first refine the heterogeneous specification into code, ignoring all timing issues. Then, once complete, we prove that the time constraints written in the initial specification are satisfied by the implementation. Then, due to monotonicity, the composition of the timing and correctness proofs will satisfy the original specification.

The refinement without time is straightforward. The first step introduces a loop index variable, j, and initializes the result r.

$$S \quad \Leftarrow \quad j, r := 1, L0.\ \ S1$$

$S1$ is a Z schema, defined as follows.

$$S1 \mathrel{\widehat=} \left[\Delta State \mid L' = L \land r' = max\{r, max\{i : j..n-1 \bullet Li\}\} \right]$$

We next refine $S1$ into a two-branch **if** statement, using the *refinement by cases* rule of [11].

$$S1 \quad \Leftarrow \quad \textbf{if } j = n \textbf{ then ok else } (j \neq n \Rightarrow S1)$$

(**ok** is the empty program that does nothing.) To prove the **then** branch, use Rule 1. To prove the **else** branch, we apply the boolean laws of specialization and discharge, which prove the obligation in two lines (see [17]).

The proof concludes by refining the **else** branch.

$$(j \neq n \Rightarrow S1) \quad \Leftarrow \quad j, r := j + 1, max\{r, Lj\}.\ \ S1$$

Now for the timing proof. We must prove that the timing constraints placed on the initial specification are maintained by the refinement, separate from the correctness proof. This requires us to show

$$\begin{array}{rcl} Time & \Leftarrow & j, r := 1, L0 \ .\ U \\ U & \Leftarrow & \textbf{if } j = n \textbf{ then ok else } Q \\ Q & \Leftarrow & j, r := j + 1, max\{r, Lj\}.\ t' - t : 5, ..10.\ U \end{array}$$

for suitable timing predicates U and Q. Note that we have inserted the time requirements for the tail-recursive call into the refinement structure; time requirements for the assignment statement are implicit in the semantics of the assignment. We conjecture that

$$\begin{array}{rcl} U & = & (j = n \Rightarrow t' = t) \land (j < n \Rightarrow t' - t : (7, ..14) \times (n - j)) \\ Q & = & j < n \Rightarrow t' - t : (7, ..14) \times (n - j) \end{array}$$

Verification is straightforward. We conclude that the timing constraint *Time* is satisfied by the implementation. By monotonicity, $S \land Time$ is implemented, and satisfies the specified time bound.

The basic approach to real-time refinement with the heterogeneous notation is to specify timing behaviour using a predicate, and then to refine a heterogeneous specification to code, using refinement techniques from [11].

5 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.

$$\begin{array}{l} Op1 \mathrel{\widehat=} \left[\Delta S;\ i? : I;\ o! : O \mid P \right] \\ Op2 \mathrel{\widehat=} \left[\Delta T;\ j? : J;\ k! : K \mid Q \right] \end{array}$$

Then

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

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. If the latter is to be done, then the translation of the schemas should be replaced by calls to a procedure, the body of which is the translation of the schema by *ZToPP*.

The definition of independent composition of Z schemas given above has the limitations noted with the operator in [11]. In particular, it is not suited for passing values of variables via shared memory (for this, we will need communication, presented in the next section).

Let's see a simple example. Consider a system with state specified by the following schema.

$$State \mathrel{\widehat=} \left[x, y, z : \mathbb{N} \right]$$

There are two operation schemas, as follows.

$$\begin{aligned} Op1 &\mathrel{\widehat=} \left[\Delta State \mid x' = z \land y' = y \land z' = z \right] \\ Op2 &\mathrel{\widehat=} \left[\Delta State \mid x' = x \land y' = z \land z' = z \right] \end{aligned}$$

Then the parallel composition of these schema is:

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

(We can omit the **frame**s generated by *ZToPP* because the state changed by both specifications is the same.)

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

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

Then the semantics of this specification would be

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

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

An advantage of using a heterogeneous approach to concurrency with Z is that with our definition of $\parallel$, the operands may be written in different notations. That is, we can write an independent composition of the form $Op \parallel P$, where Op is a Z schema and P a predicative specification. This gives us the flexibility to use the most appropriate notation to specify each process. It also gives us the flexibility to use the most appropriate refinement relation on each component. For Z specifications, we can use Z refinement, and will produce a predicative refinement of the whole. For predicative specifications, we use predicative refinement as usual.

5.1 Example

We contrast using the combination of Z and predicative notation with using Evans' extension of Z [5], applied to a simple telecommunications protocol from [21]. We show that the heterogeneous notation can produce specifications and proofs that are comparable in size and complexity to those of Evans.

Evans's 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*.

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

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.

Transmit

$\Delta State$
$m? : M$

$in' = \langle m? \rangle \frown in$
$out' = out$

Receive

$\Delta State$

$in' = in$
$\#out' = \#out + 1 \lor out' = out$

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*.

ParBehaviour

$\sigma : \mathbb{N}_1 \to State$

σ validcomp $(\{Init \bullet \theta State\}, \{NextState \bullet \theta State \mapsto \theta State'\})$
σ wf $\{Receive \bullet \theta State \mapsto \theta State'\}$

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 [5].

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. In order to bring initialization into the specification, the schema $Init'$ can be sequenced with the fixed-point construction *Behaviour*, i.e.,

$$Init'.\ Behaviour.$$

Note that in *Behaviour*, *Receive* has the option of doing nothing each time it is enabled. Thus, progress is not guaranteed. Progress can be guaranteed by constraining the *Receive* operation, in exactly the same way as is done in [5]. To do this, the *Receive* operation is extended to $[Receive \mid out' \neq out]$. Further, *Behaviour* can be extended to specify the weak fairness of the operation *Receive* by conjoining *Behaviour* with

$$\forall t \cdot \exists t'' : t \leq t'' \leq t' \cdot Receive[t''/t']$$

which says that it is always the case that eventually *Receive* will be true.

5.2 Safety and liveness

In [5], it is shown how to prove safety and liveness properties with an extension of Z. We now briefly discuss how to carry out such proofs with the heterogeneous notation, in a format that is close to that used in [5].

We begin with safety (invariant) properties. Let A be a concurrent heterogeneous specification.

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

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 one process changes variable x, and another process 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

$$\begin{aligned}(S_0 \parallel \ldots \parallel S_{k-1}) &= (S_0 \wedge \ldots \wedge S_{k-1}) \\ &\Rightarrow (S_0 \vee \ldots \vee S_{k-1})\end{aligned} \tag{1}$$

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

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

In general, if we can prove that the independent composition of S_i's implies the disjunction of the S_i's, then safety properties can be proven by parts. This is necessary in general because the operands of the independent composition can change the same variables. In [11], it is recommended not to write independent compositions in this way (i.e., with intersecting **frame**s), because it leads to unsatisfiable specifications. Communication constructs can be used to avoid the problem. In [5], proof of safety properties by parts is possible in general because parallelism is simulated by nondeterministic interleaving; $\parallel$ 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

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

Notice that P happens to be 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 [5], 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), above, will hold, and the **leads-to** property can be proven by parts, using a UNITY-style law adapted from [3], in much the same way as Evans [5].

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.

$$\forall t \cdot P \Rightarrow (\mathrm{pre}\, S_j)$$

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 must (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) in the behaviour 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. In such situations, it is useful to introduce a variant that is decreased on each iteration. For such systems, we must show

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

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

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

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

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.

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

Next, we show that each system operation either maintains P or establishes Q. The first part shows this for *Receive*.

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

A similar obligation must be discharged for *Transmit*.

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

Finally, it must be shown that the weakly fair 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*, while the second holds because *Transmit* maintains $out' = out$ and implies P'. The third 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 somewhat more complex than in Evans' approach, in part because concurrency in the notation is effectively conjunction, as opposed to disjunction in [5]. 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 [11] (i.e., avoid writing to shared memory) and instead make use of the communication operators presented in the next section, then for most practical examples, a partwise approach to proof can be used.

5.3 Data transformation

The specification of operations *Receive* and *Transmit* can be further refined, using the data transformation theory of [11].

To data transform a specification, given implementer's variables v, we must produce new *implementer's variables* w as well as an abstraction invariant D (called a data transformer in [11]). The invariant must satisfy

$$\forall w \cdot \exists v \cdot D$$

Each specification S is transformed, under the abstraction invariant, to

$$\forall v \cdot D \Rightarrow \exists v' \cdot D' \land S$$

In the example, the implementer's variables are specified by *State*; the new implementer's variables are specified by a new schema, *Section*. This new state describes the *route* that messages take through a network: a sequence of signalling point codes (SPCs) without repetition. Each section in this route may receive and send messages; those which have been received but not sent exist in the *in* section.

$$[SPC]$$

Section

$$route : \mathrm{iseq}\ SPC$$
$$rec, ins, sent : \mathrm{seq}(\mathrm{seq}\ M)$$

$$route \neq \langle\rangle$$
$$\#route = \#rec = \#ins = \#sent$$
$$rec = ins \approx sent$$
$$front\ sent = tail\ rec$$

ins represents the sequence of messages currently within the section, *rec* is the sequence of received messages, and *sent* the transmitted messages. $\approx$ denotes the pairwise concatenation of two lists. A formal definition is in [21].

A data transformer to relate the two states is

$$D \quad = \quad in = head\ rec \land out = last\ sent$$

(where *head* and *last* are predicative translations of the corresponding Z toolkit functions).

To apply the data transformer to a Z specification, we must define the effect of the transformer on a Z schema. This is straightforward. Let *Op* be a Z operation schema. Then

$$\forall v \cdot D \Rightarrow \exists v' \cdot D' \land Op \quad \widehat{=} \quad \forall v \cdot D \Rightarrow \exists v' \cdot D' \land ZToPP(Op)$$

Thus, after transformation and some simplification, the *Transmit* operation is transformed as follows. Let J be the property of schema *Section*.

$$STransmit = \mathbf{frame}\ rec \cdot J \Rightarrow (head\ rec' = [m]^+(head\ rec) \land tail\ rec' = tail\ rec \land J')$$

Receive is transformed to

$$SReceive = \mathbf{frame}\ ins, sent \cdot J \Rightarrow (front\ ins' = front\ ins \land last\ ins' = front(last\ ins) \land$$
$$front\ sent' = front\ sent \land last\ sent' = [last(last\ ins)]^+(last\ sent) \land J')$$

In the original abstract view, messages arrived at their destination nondeterministically. In the sectional view, nondeterminism is explained by the progress of messages through a sequence of sections. Thus, we should add an operation, *Daemon*, that moves messages. The effects of the *Daemon* are invisible in the abstract state, but visible in the concrete state. The *Daemon* can be specified as a Z schema.

Daemon

$\Delta Section$

$\exists\, i : 1..\#route - 1 \mid ins\ i = \langle\rangle \bullet$
$\quad (ins'\ i = front(ins\ i) \land$
$\quad ins'(i+1) = \langle last(ins(i)) \rangle \frown ins(i+1) \land$
$\quad (\forall\, j : \mathrm{dom}\, route \mid j \neq i \land j \neq i+1 \bullet ins'\ j = ins\ j))$

To specify the system's concurrent behaviour, the transformed operations and *Daemon* are composed in an independent composition, *SectionBehaviour*.

$$SectionBehaviour = (STransmit \parallel SReceive \parallel Daemon).\ SectionBehaviour$$

To guarantee eventual reception of messages, the *SReceive* and *Daemon* operations should be constrained to be weakly fair with respect to start and completion times of a computation. Define $\mathsf{wf}\ S$ as

$$\mathsf{wf}\ S \quad = \quad \forall\, t \cdot \exists\, t'' : t \leq t'' \leq t' \cdot S[t''/t']$$

Then *SectionBehaviour* must be further constrained, as follows.

$$SectionBehaviour \land\ \ \mathsf{wf}\ SReceive \land\ \ \mathsf{wf}\ Daemon$$

It must also be shown that *Daemon* is correct with respect to the abstract state. Let I be the property of *State*. To start, we must prove that

$$\forall\, in, out, rec, route, ins, sent \cdot I \land D \Rightarrow (\mathrm{pre}\, Daemon)$$
$$\forall\, in, out, in', out', route, rec, ins, sent, route', rec', ins', sent' \cdot$$
$$\quad D \land ZToPP(Daemon) \land D' \Rightarrow (\mathrm{pre}\, Daemon \Rightarrow I' \land in' = in \land out' = out)$$

The first proof obligation is to show that the *Daemon* is always enabled. The second obligation is to show that if enabled the *Daemon* does not change the abstract state.

Next, we must show two things: that whenever a weakly fair operation on the abstract state is enabled, it will remain enabled at least until the corresponding concrete operation occurs; and, whenever a weakly fair operation on the abstract state is enabled, the corresponding operation on the concrete state is eventually enabled.

Let $\mathsf{dt}\ S$ be $\forall\, v \cdot D \Rightarrow \exists\, v' \cdot D' \land S$, i.e., the transformation of specification S by data transformer D. Then the proof obligations for weakly fair abstract operation A and weakly fair concrete operation C are

$$\forall\, t_1, t_2 : t1 \leq t_2 \cdot (\mathsf{dt}\ \mathrm{pre}\, A)[t_1/t] \land C[t_2/t] \Rightarrow \forall\, t'' : t_1 \leq t'' \leq t_2 \cdot (\mathsf{dt}\ \mathrm{pre}\, A)[t''/t]$$
$$(\mathsf{dt}\ \mathrm{pre}\, A) \Rightarrow \exists\, t'' : t'' \geq t \cdot (\mathrm{pre}\, C)t''$$

These obligations must be discharged for *Receive* and *SReceive*.

A limitation of the predicative theory of data transformation is that it cannot express infinite stuttering.

6 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.

6.1 Mutual exclusion

The first example involves mutual exclusion, critical sections, and synchronization. We specify a concurrent queueing system. One process adds jobs to a queue, while a second process removes jobs from the queue and services the job.

The system will be specified using the combination of Z and predicative notation. The processes will require mutually exclusive access to the queue. We first specify the system (omitting the details of how a job is to be serviced), and then write a specification expressing mutual exclusive access to the critical section.

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.

$$[PROCESS]$$

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

The system state is as follows.

State

$$queue : \text{seq}_\infty\ PROCESS$$
$$numjobs : \mathbb{N}$$
$$numjobs = \#queue$$

AddJob places new jobs into the queue.

AddJob

$$\Delta State$$
$$job? : PROCESS$$
$$numjobs' = numjobs + 1$$
$$queue' = queue \frown [job?]$$

The operation to service a job is as follows.

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

$$P = N_p.\ \ a!\top.\ \ AddQueue.\ \ a!\bot.\ \ P$$
$$Q = N_q.\ \ b!\top.\ \ (ServiceJob \curlyvee RServiceJob).\ \ b!\top.\ \ Q$$

N_p is a specification that performs some initialization for *AddQueue*; N_q performs initialization for the service operations. Note that the schema disjunction *ServiceJob* $\curlyvee$ *ServiceJob* has an implicit *true* precondition, i.e., it is always enabled. Thus, progress is guaranteed in this concurrent system. By comparison, if we were to remove *RServiceJob* from Q, above, progress would not be guaranteed, because *ServiceJob* would only be enabled when *numjobs* was at least 1.

To ensure that mutual exclusion is guaranteed, the specification must satisfy

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

Informally, 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).

6.2 Short-term scheduler

We now present a more detailed example that brings together all the techniques we have described in the paper. The example 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 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, starting with the ready queue holding *INITIAL* jobs. Total wait time and average wait time should be output upon completion. A circular ready queue of fixed length 100 should be used.

We can provide a *rough sketch* [14] of the system as a data flow diagram in Fig. 1.

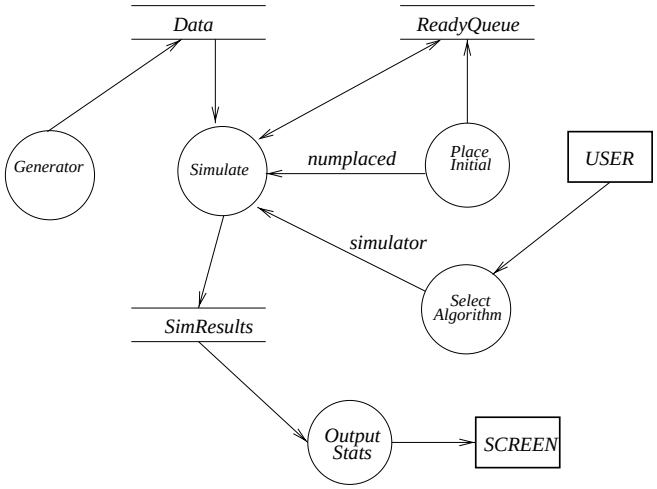

Fig.1. Rough sketch of simulator system

(The diagram in Fig. 1 is given only for illustrative purposes. We ascribe no particular semantics to it; it is drawn to help us understand the system.) In the figure, processes are written as circles, external entities in the environment as rectangles, and data stores as parallel lines. Data flow between processes, entities, and data stores is written using labelled arrows.

To formally specify the system, we express (using Z and predicative notation) the behaviour of each process and the mechanism for communication among the processes, via channels. Before doing so, we specify the state of the system, and formalize the terms used in the rough sketch of Fig. 1.

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

$$INITIAL, NUMBER : \mathbb{N}$$
$$LAMBDA : \mathbb{R}$$

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

Cell

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

ReadyQueue

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

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.

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

The type definition for the algorithms available to simulate (we specify only two) is as follows.

$$\mathit{ALGTYPE} ::= \mathit{rr}, \mathit{fcfs}$$

(*rr* and *fcfs* are names for which we provide no definition.) Two channels appear to be necessary for this system (though we might choose to make the interface between the system and the external entities *USER* and *SCREEN* channels as well): one between processes *PlaceInitial* and *Simulate*, and another between *SelectAlgorithm* and *Simulate*. The rough sketch of Fig. 1 suggests names for these channels: *numplaced*, and *simulator*, respectively. These are specified in the usual predicative notation. (We add process details later.)

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

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

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

The partial specifications *FCFS* and *RR* specify the behaviour of the first-come first-served and round-robin schedulers, respectively. Predicative notation is better suited for specifying the iterative parts of the simulator (via a **while** loop), because it is a wide-spectrum language with an embedded programming language part. Z is useful for specifying the remaining parts.

The *FCFS* schema is as follows.

FCFS

$$\Delta \mathit{ReadyQueue}$$
$$\mathit{marr} : \mathbb{N}$$

$$\mathit{marr} = \mathit{max}\{j : \mathit{arriving}..\mathit{NUMBER} - 1 \mid \mathit{time} + \mathit{boundary} > \mathit{arrivals}(j)\}$$
$$\mathit{head} \neq (\mathit{tail} + \mathit{marr} - \mathit{arriving}) \bmod 100$$
$$\forall\, i : 0..\mathit{marr} - 1 \bullet \exists\, \mathit{new} : \mathit{Cell} \bullet (\ \mathit{new}.\mathit{burstlength} = \mathit{bursts}(\mathit{arriving} + i)$$
$$\mathit{new}.\mathit{arrivaltime} = \mathit{arrivals}(\mathit{arriving} + i) - \mathit{boundary}$$
$$\mathit{new}.\mathit{group} = \lfloor \mathit{bursts}(i) \rfloor$$
$$\mathit{ready}'((\mathit{tail} + i) \bmod 100) = \mathit{newtail}' = (\mathit{tail} + \mathit{marr} - \mathit{arriving}) \bmod 100$$
$$\mathit{length}' = \mathit{length} + (\mathit{marr} - \mathit{arriving})$$
$$\mathit{arriving}' = \mathit{marr}$$

(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 must be generated, and the system must be initialized. Initialization involves placing *numplaced* data items in the queue and selecting a scheduling algorithm (*RR* or *FCFS*) before simulation.

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

Data is generated using a random number generator, *rand*, which returns a random *real*.

$$0 \leq rand < 1 \land rand : real$$

The *Generator* schema is as follows.

Generator

$\Delta Data$

$$arrivals'(0) = 0$$
$$(bursts'(0) = 0.9 \times rand + 0.1 \lor bursts'(0) = 9.0 \times rand + 1.0)$$
$$\forall i : 1..NUMBER - 1 \bullet$$
$$(bursts'(i) = 0.9 \times rand + 0.1 \lor bursts'(i) = 9.0 \times rand + 1.0)$$
$$(arrivals'(i) - arrivals'(i-1) = -LAMBDA \times \log_e\ rand)$$

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

$$OutputStats \quad = \quad Screen!data$$

where *Screen* is a declared channel and *data* the collected simulation data. Similarly, *SelectAlgorithm* might be

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

where *User* is a declared channel.

Now refinement can occur, using the results we described earlier. We omit most of the details, since they follow standard refinement practice. The *Generator* specification can be refined to a simple loop, using standard Z refinement techniques, e.g., as described in [22]. This can occur due to our result that shows that Z refinement is monotonic over all predicative combinators. The guard on the loop implementing the generator is $i < NUMBER$, a loop variant is $NUMBER - i$, and a loop invariant is:

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

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 [22]. The result of the refinement and proof obligations is the following program (in Dijkstra's guarded command language).

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

Refinement of the *FCFS* is somewhat more complex (it involves a more complex loop invariant, and the loop body is more complicated). We found it easier to refine the specification to code using predicative refinement rules, developing a recursive program instead of an iterative program. The heterogeneous setting allows us to do this. To carry out the refinement, we first define a queueing function *enqueue*, as follows.

$$\begin{aligned} \mathit{enqueue} = \lambda\, a, b : \mathit{real};\ g : \mathit{int} \cdot (\mathit{head} \neq (\mathit{tail} + 1 \bmod 100)) \Rightarrow \\ \mathit{ready}' = (\mathit{tail};\ \text{“}\mathit{burstlength}\text{”}) \to a \mid (\mathit{tail};\ \text{“}\mathit{arrivaltime}\text{”}) \to b \mid \\ (\mathit{tail};\ \text{“}\mathit{group}\text{”}) \to g \mid \mathit{ready} \wedge \\ \mathit{tail}' = (\mathit{tail} + 1) \bmod 100 \wedge \mathit{length}' = \mathit{length} + 1 \end{aligned}$$

Refinement is carried out 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 tail-recursive program with guard

$$((\mathit{time} + \mathit{boundary} > \mathit{arrivals}(\mathit{arriving})) \wedge \mathit{arriving} < \mathit{NUMBER} \wedge \mathit{length} \neq 100)$$

providing that we use *arriving* as the index variable for determining when to exit the tail recursion. The result of the refinement (a tail recursive program) can be transliterated to a **while**-loop, which looks as follows.

while $((\mathit{time} + \mathit{boundary} > \mathit{arrivals}(\mathit{arriving})) \wedge \mathit{arriving} < \mathit{NUMBER} \wedge \mathit{length} \neq 100)$ **do** (
$\quad \mathit{enqueue}(\mathit{bursts}(\mathit{arriving}), \mathit{arrivals}(\mathit{arriving}) - \mathit{boundary}, \lfloor \mathit{bursts}(\mathit{arriving}) \rfloor).$
$\quad \mathit{arriving} := \mathit{arriving} + 1$
)

7 Discussion

The aim of integrating Z with predicative notation is to construct a notation that is suitable for specification and for programming of real-time, concurrent, and communicating systems. It is therefore in our best interests to make specification and refinement (which is what we do when we program) as simple as possible. It is also in our best interests to integrate Z and predicative programming in an unintrusive manner, so that the individual notations can be used in combination in a manner that is close to how they can be used separately. These ideas led us to choose an intersection semantics for the combination of Z and predicative notation, which in turn led to simple refinement rules and specifications.

In [5], Evans suggests a number of disadvantages to integrating Z with notations — like CSP, TLA, or CCS — that are better suited to specifying concurrent behaviour.

1. Reconciling the semantics of the individual notations.
2. Using existing Z tools (e.g., for type checking and proof).
3. Poor use of the Z proof system.

The integration of Z with predicative notation that we have presented in this paper does not suffer from these disadvantages. We discuss each of these points in more detail.

Reconciling the semantics of notations can be difficult, especially for notations with very different semantics, such as Z and CSP (though see [19]). But predicative notation and Z can be used to present the same view of a system; both notations are model-oriented. Therefore, combining the notations is much simpler than combining Z and CSP, for example. And since predicative notation is well-suited to specifying concurrent behaviour, so too should the integrated notation.

As we pointed out earlier, reconciling the semantics of similar notations can be difficult, in particular, when the notations differ in expressiveness. A clear understanding of the roles of the notations in the integration will help in determining the best way to reconcile the semantics. In the integration of this paper, the role of predicative programming is for specifying communication and concurrency primitives, as well as to provide a basis for refinement. Z is used for specifying system operations, and can also be used for refinement if desired.

With an integration of Z with CSP, the ability to use existing Z tools (and existing CSP tools, like FDR) 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. Some behind-the-scenes translation may have to be done in order to get information regarding the system state needed to use the Z tools.

As well, by using predicative notation as the semantic basis for the heterogeneous notation, we allow ourselves to use theorem provers based on typed set theory. So, for example, PVS can be used to support the predicative method, and therefore the heterogeneous notation. This creates the need for a tool (e.g., based on TXL) that will automatically translate heterogeneous specifications written in Z and predicative notation, into PVS syntax. Work is underway on creating such a tool.

Finally, the integration of Z and predicative notation allows use of Z proof techniques when applied to Z partial specifications (that may, perhaps, be composed with and by predicative specifications). This is due to the particular reconciliation of semantics that we have chosen.

Our suggestion, then, is not that the problems suggested by Evans will not be apparent when combining very different notations — like Z and CSP — but by carefully choosing notations that can be unintrusively integrated, and by understanding the roles each notation will play in the integration, the disadvantages may vanish, or at least prove to be less critical.

8 Conclusions

In this paper, it has been shown how the Z notation can be used, in combination with the predicative notation of [11], to specify and reason about concurrent, real-time, communicating behaviours. The motivation for the work was to attempt to demonstrate that limitations noted with previous integrations [4, 6, 8] could be partially alleviated by integrating Z with the right notation. In order to overcome these limitations, an intersection semantics for Z and predicative notation was used, and it was demonstrated that such a semantics allows (practically) full use of Z and maintains the ability to use Z proof techniques and tools on Z partial specifications.

The approach was aimed at showing how refinement could be used on heterogeneous specifications of concurrent, real-time, or communicating behaviour. We demonstrated how both Z refinement techniques and predicative refinement techniques could be applied in such situations.

An important aspect of this work is that it shows that it need not be necessary to extend Z, or to even change the standard Z approach to specification, 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.

References

1. P. Baumann and K. Lerner. A Framework for the Specification of Reactive and Concurrent Systems in Z. In *Proc. 15th Conference on Foundations of Software Technology and Theoretical Computer Science*, LNCS 1026, Springer-Verlag, 1995.
2. J.-M. Bruel, A. Benzekri, and Y. Raymaud. Z and the Specification of Real-time Systems. In *Proc. 7th Int. Conf. on Putting into Practice Methods and Tools for Information System Design*, IRIN, 1995.
3. K.M. Chandy and J. Misra. *Parallel Program Design: A Foundation*, Addison-Wesley, 1988.
4. R. Duke and G. Smith. Temporal Logic and Z Specifications. In *Australian Computer Journal*, 21(2), May 1989.
5. A.S. Evans. A Case Study in Specifying, Verifying, and Refining a Parallel System in Z. To appear in *Parallel Processing Letters*, 1998.
6. E. Fergus and D. Ince. Z specifications and modal logic. In *Proc. Software Engineering 90*, Cambridge, 1990.
7. C.J. Fidge. Real-time Refinement. In *Proc. FME '93*, LNCS 670, Springer-Verlag, 1993.
8. R. Gotzhein. Specifying open distributed systems with Z. In *VDM and Z — Formal Methods in Software Development*, LNCS 428, Springer-Verlag, 1990.
9. I. Hayes and M. Utting. Deadlines are Termination. In *Proc. PROCOMET '98*, Chapman and Hall, 1998.
10. E.C.R. Hehner and A.J. Malton. Termination Conventions and Comparative Semantics, *Acta Informatica*, 25 (1988).
11. E.C.R. Hehner. *A Practical Theory of Programming*, Springer-Verlag, 1993.
12. E.C.R. Hehner. Abstractions of Time. In *A Classical Mind: Essays in Honour of C.A.R. Hoare*, Prentice-Hall, 1994.
13. C.A.R. Hoare. Lectures given at the NATO ASI International Summer School on Program Design Calculi, Marktoberdorf, July 1992.

14. M.A. Jackson, *Software Requirements and Specifications*, (Addison-Wesley, 1995).
15. L. Lamport. TLZ. In *Proc. ZUM '94*, Springer-Verlag, 1994.
16. R.F. Paige. A Meta-Method for Formal Method Integration. In *Proc. Formal Methods Europe '97*, LNCS 1313, Springer-Verlag, 1997.
17. R.F. Paige. Comparing Extended Z with a Heterogeneous Notation for Reasoning about Time and Space. In *Proc. 11th Int'l Conference of Z Users*, LNCS 1439, Springer-Verlag, 1998.
18. J.M. Spivey. *The Z Notation: A Reference Manual*, Prentice-Hall, 1989.
19. 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.
20. N. Ward. Adding specification constructors to the refinement calculus. In *Proc. FME '93*, LNCS 670, Springer-Verlag, 1993.
21. J. Woodcock and J. Davies. *Using Z*, Prentice-Hall, 1996.
22. J.B. Wordsworth. *Software Development with Z*, Addison-Wesley, 1992.