Electronic Notes in Theoretical Computer Science 42 (2001)
URL: http://www.elsevier.nl/locate/entcs/volume42.html 23 pages

Lifting in Z

Andrew Martin [a] and Colin Fidge [b]

[a] *Oxford University Software Engineering Centre, Wolfson Building, Parks Road, Oxford, OX1 3QD, United Kingdom*

[b] *Software Verification Research Centre, The University of Queensland, Queensland 4072, Australia*

Abstract

Formal notations such as Z provide powerful support for writing clear specifications, and for undertaking proofs of properties of those specifications. In this paper, we explore one particular style of specification, with applications in control theory and real-time specification. The notation we define permits the accurate description of concepts in these fields without significant overhead or notational clutter. This notation has long been used by practitioners in these fields; we demonstrate that it may be defined within Draft Standard Z, and that the resulting specifications are amenable to proof.

1 Introduction

Formal notations such as Z provide powerful support for writing clear specifications, and for undertaking proofs of properties of those specifications. Much of this benefit comes from having a rich language of pre-defined concepts and types to draw on, so that specifications can both be concise and precise, and also accessible. Everything we write in Z could also be written using classical first order predicate logic, but the result would be far less approachable.

In this paper, we explore one particular style of specification, with applications in (at least) control theory and real-time specification. The 'lifting' notation we define permits the accurate description of concepts in these fields without significant overhead or notational clutter. Lifted operators and notations have long been used implicitly, and often somewhat informally, by practitioners in these fields. We demonstrate that such notations can be formally defined within Draft Standard Z [19] (hereinafter, 'Standard Z') and that it is useful to do so. Keeping the definitions within Z has the value that it makes the specifications amenable to analysis using existing Z tools.

In these and other applications, we are interested in lifting our description from one involving simple data types into one which uses *functions* to those data types. Lifting allows an algebra defined for a simple type, for example

the integers, to be applied to a more complex type, for example functions from reals to integers. Concretely, in specification of real-time systems [21], we would like to write expressions such as

$$speed < 20$$

to say not that a constant value called *speed* does not exceed 20, but that the values taken by variable *speed* at various times t never exceed 20. That is,

$$\forall\, t : \mathbb{T} \bullet speed(t) < 20 \ .$$

Here and throughout this paper, $\mathbb{T}$ is the set of time values, typically the real numbers, $\mathbb{R}$.

Another example of lifting arises in the use of polynomials in various applications. Whilst polynomials are generally developed over the real field, they may usefully be employed over a wide variety of base systems. Such a treatment of polynomials is popular in control theory, where *transfer functions* are used in the solution of difference equations [4]. An example of such a use is presented in Section 4.2.

2 Previous approaches to lifting

In this section we review two previously described approaches to lifting in Z, one with an explicit lifting function, the other overloading the existing Z operators.

2.1 Explicit lifting

A number of pieces of work related to the specification of timing properties in Z are reported on by Duddy et al. [10]. In the section on lifting, they emphasise that when describing the properties of continuous timed histories, an appropriate collection of function operators is essential for writing understandable specifications. Thus it is appropriate to generalise addition of real numbers to addition of real-valued (history) functions,

$$(f +' g)(t) == f(t) + g(t) \ ,$$

permitting the expression of concise conditions such as

$$h = f +' g \ ,$$

rather than the more verbose

$$\forall\, t : \mathbb{T} \bullet h(t) = f(t) + g(t) \ .$$

They continue by describing a homomorphism (that is, a lifting function) on an algebra (S, O), for S a collection of data objects and O a collection of

operators on S. The homomorphism, for any domain type A, is the algebra $(S{\uparrow}^A, O{\uparrow}^A)$, where S is lifted to functions from domain A, and similarly for O:

$$
\begin{aligned}
&S{\uparrow}^A == A \to S \\
&O{\uparrow}^A == \{\ o : O \bullet o{\uparrow}^A\ \} \\
&o{\uparrow}^A == \lambda f_1, \ldots, f_n : (A \to S) \bullet (\lambda\, a : A \bullet o(f_1(a), \ldots, f_n(a)))\ .
\end{aligned}
$$

Thus $(\_ + \_){\uparrow}^{\mathbb{T}}$ is precisely the operator $(\_ +' \_)$ defined above. Extending the lifting operator to expressions, it is possible to prove that laws which hold in the original term algebra of (S, O) also hold in that of $(S{\uparrow}^A, O{\uparrow}^A)$ (thus it is a *covariant hom-functor* [15]).

This operator may then be used to 'lift' a variety of Z expressions and predicates. (The latter is accomplished by using boolean-valued functions in place of predicates. To handle the possibility of undefined predicates, a logic of partial functions is used.) Because there is no syntactic distinction in Z between operators and values (elements of O and S respectively) an elaborate collection of special cases was evolved to cover every eventuality, and the resulting lifting algorithm has been implemented and tested.

Thus, with the following declarations of a time-varying value and a time-invariant constant,

$$
\begin{aligned}
&speed : \mathbb{T} \to \mathbb{R} \\
&maxSpeed : \mathbb{R}
\end{aligned}
$$

the expression '$maxSpeed - speed$' is lifted as follows:

$$
\begin{aligned}
&(maxSpeed - speed){\uparrow}^{\mathbb{T}} \\
&= (\_ - \_){\uparrow}^{\mathbb{T}}(maxSpeed, speed) \\
&= \lambda\, t : \mathbb{T} \bullet (\_ - \_)(maxSpeed, speed(t)) \\
&= \lambda\, t : \mathbb{T} \bullet maxSpeed - speed(t)\ .
\end{aligned}
$$

Note that the declared type of identifiers decides whether they are indexed or not.

The lifting algorithm is further complicated by a need from the application to accomplish lifting *with respect to two values simultaneously*. In describing a time interval, we may wish to write

$$speed(\alpha) \leq speed$$

to specify that the value of *speed* throughout the interval is never less than its *initial* value. Here α is a function on *intervals* which returns the time at the beginning of the interval (the infimum of the interval). Thus the expression to the right varies according to *time*, and that on the left according to *intervals of time*.

Writing $\mathbb{TI}$ for the set of time intervals (that is, the set of contiguous subsets of $\mathbb{T}$), and ${\uparrow}^{\mathbb{T},\mathbb{TI}}$ for simultaneous lifting with respect to both times

and intervals, we have

$$(speed(\alpha) \leq speed)\uparrow^{\mathbb{T},\mathbb{TI}} = \lambda\, \Delta : \mathbb{TI};\ t : \mathbb{T} \bullet speed(\alpha(\Delta)) \leq speed(t)\ .$$

In this example we have also seen how a single variable may be used in more than one way. On the left, *speed* has been explicitly de-referenced in one subexpression and not in the other. Again, the lifting algorithm must take account of types in determining how variables are to be lifted.

2.2 Lifting without points

A less algorithmic approach is described by Brien et al. [6], achieving similar results by defining a family of lifted functions, operators, and relations. The definitions are similar to that for $(\_ +' \_)$ above, but are careful to avoid the use of *points* in the definitions wherever possible. This choice is informed by the observation that quantifiers and lambda abstractions tend to complicate proof, and that definitions which refer to functions without mentioning their points of application are often much easier to use.

In this approach, most of the operators are overloaded, as illustrated in Figure 1, adapted from the paper of Brien et al. [6]. Notice that, as with the previous approach, a boolean type is used, so relations are boolean-valued functions. The diagram illustrates how relations on reals (re-cast as boolean-valued functions) can be re-interpreted as relations on real-valued functions of time (giving boolean-valued functions of time). For example, a variable $v : \mathbb{R}$ may be lifted over the time domain to create a trace variable $v_{\mathbb{T}} : \mathbb{T} \to \mathbb{R}$. When interpreted in an expression, such as $4 \leq v_{\mathbb{T}}$, the lifted relation yields a result of type $\mathbb{T} \to \mathbb{B}$.

These functions can be lifted further into real-valued functions of *time intervals* by considering their value at the beginning and end-points (for function f these are '$b.f$' and '$e.f$'), or by integration. Moreover, constants may be compared to functions of either sort, by converting them into constant functions. Similarly, predicates (boolean-valued functions) can be lifted to time intervals using the '$\lceil \_ \rceil$' operator, and the duration for which a predicate holds in an interval can be found by using integration and defining type boolean to be the set $\{0, 1\}$.

In this approach, the homomorphism described above is defined as follows [6]. Given a set S and a function or relation h, then the *hom-functor* $(S \longrightarrow h)$ accepts a function from S to the domain of h and returns a function from S to the range of h. Since the quantified function f in the schema below may not be total, the definition requires set S to be the domain of f.

$$[X, Y, Z]$$
$$\_ \longrightarrow \_ : \mathbb{P}\, X \times (Y \nrightarrow Z) \to (X \nrightarrow Y) \nrightarrow (X \nrightarrow Z)$$
$$\forall f : X \nrightarrow Y;\ h : Y \nrightarrow Z \bullet (\mathrm{dom}\, f \longrightarrow h)\ f = h \circ f$$

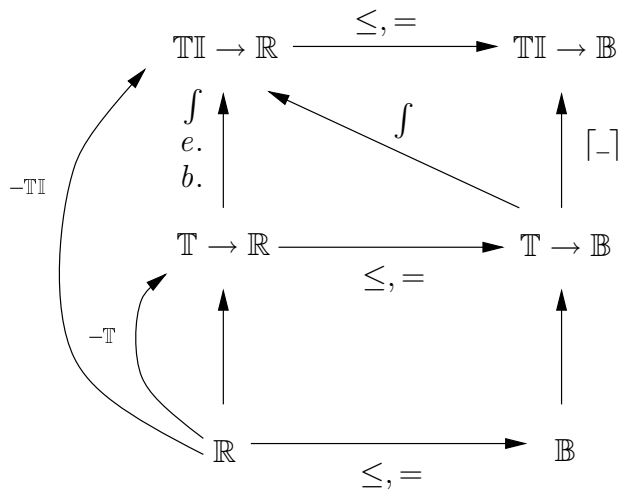

Fig. 1. Lifting by overloading.

Notice the lack of points. For example, assuming that operator $(\_ + \_)$ is of type $(\mathbb{R} \times \mathbb{R}) \to \mathbb{R}$, and given a function f of type $\mathbb{T} \to (\mathbb{R} \times \mathbb{R})$, then $(\mathbb{T} \longrightarrow (\_ + \_))\ f$ is a function of type $(\mathbb{T} \to (\mathbb{R} \times \mathbb{R})) \to (\mathbb{T} \to \mathbb{R})$.

On its own, this definition does not provide a lifting in the form we want. To overcome this, two product functors are defined (the first is used here; the second will appear below):

$$
\begin{array}{|l}
\hline
[X, Y, Z] \\
\Box : (X \to Y) \times (X \to Z) \to \\
\qquad X \to (Y \times Z) \\
\hline
\forall f : X \to Y;\ g : X \to Z;\ x : X \bullet \\
\qquad \Box(f, g)(x) = (f(x), g(x)) \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
[X, Y, Z, W] \\
\_ \times \_ : (X \to Y) \times (Z \to W) \to \\
\qquad (X \times Z) \to (Y \times W) \\
\hline
\forall f : X \to Y;\ g : Z \to W;\ x : X;\ y : Z \bullet \\
\qquad (f \times g)(x, y) = (f(x), g(y)) \\
\hline
\end{array}
$$

In this way, lifted operators may be defined without reference to points:

$$(\_ +' \_)[X] == (X \longrightarrow (\_ + \_)) \circ \Box \ .$$

For example, function $(\_ +' \_)[\mathbb{T}]$ is of type $((\mathbb{T} \to \mathbb{R}) \times (\mathbb{T} \to \mathbb{R})) \to (\mathbb{T} \to \mathbb{R})$, which gives us the desired lifting of real addition over the time domain.

In its use of boolean-valued functions, this overall approach begins to be somewhat removed from Z. The definitions in the next section will largely

follow this style, but will stay purely within Standard Z, making use of loose generics.

3 Lifting with loose generics

In this section, we construct in Z the necessary apparatus for general lifting. We incorporate features of both approaches presented above, but are largely inspired by the overloading approach of Section 2.2. (An alternative, different in form rather than structure, would be to follow the functional programming literature, for example the work of Backus [2], or indeed, in the modern style, a presentation using monads [20].)

3.1 Rings and fields

A *ring* is a mathematical structure with two operators, usually denoted as addition and multiplication [3, p. 238]. The two operators satisfy a number of properties and algebraic laws. We summarise these using a schema.

ring $[X]$

$$
\begin{array}{ll}
\_ + \_ : X \times X \to X & \text{[closure under addition]} \\
\_ * \_ : X \times X \to X & \text{[closure under multiplication]} \\
- : X \to X & \\
\mathbf{0} : X & \\
\hline
\forall\, x : X \bullet & \\
\quad \mathbf{0} + x = x \land & \text{[identity of addition]} \\
\quad (-x) + x = \mathbf{0} & \text{[inverse of addition]} \\
\forall\, x, y : X \bullet & \\
\quad x + y = y + x & \text{[commutativity of addition]} \\
\forall\, x, y, z : X \bullet & \\
\quad x + (y + z) = (x + y) + z \land & \text{[associativity of addition]} \\
\quad x * (y * z) = (x * y) * z \land & \text{[associativity of multiplication]} \\
\quad x * (y + z) = (x * y) + (x * z) & \\
 & \text{[distribution of multiplication]}
\end{array}
$$

When the multiplication operation is commutative and has an identity and inverses, the resulting structure is a *field* [3, p. 245].

field $[X]$

$ring[X]$
$\_^{-1} : X \nrightarrow X$
$\mathbf{1} : X$

$\mathbf{0} \neq \mathbf{1}$ [identity and zero distinct]
$\mathrm{dom}(\_^{-1}) = X \setminus \{\mathbf{0}\}$ [no zero divisors]
$\forall\, x : X \mid x \neq \mathbf{0} \bullet$
$\quad x * x^{-1} = \mathbf{1}$ [inverse of multiplication]
$\forall\, x : X \bullet$
$\quad x * \mathbf{1} = x$ [identity of multiplication]
$\forall\, x, y : X \bullet$
$\quad x * y = y * x$ [commutativity of multiplication]

Observe that these Z definitions depart from mathematical tradition slightly, in that we usually think of a ring as a structure $\langle X, +, * \rangle$. In these definitions, we have said what it means for some binding of operations to form a ring or field over some type X. The examples of Section 4.1 use the ring of real-valued functions, described in the example below. In Section 4.2 real-valued functions are used differently, in a structure which (almost!) forms a field. Clearly, if our application required it we could define other structures, such as a commutative ring [3, p. 239].

If we were pursuing definitions without points wherever possible, we might write these definitions somewhat differently. For example, the last axiom in *field* could be rendered as

$$(\_ * \_) = (\_ * \_) \circ \square(\mathit{second}, \mathit{first}) \ .$$

3.1.1 Examples

The most well-known rings are the reals and the complex numbers with the usual arithmetic operations. Assuming toolkit definitions of reals and real operators, the fact that the reals form a ring can be stated as:

$$\vdash \langle\!|\, \mathbf{0} == 0,$$
$$\quad - == -,$$
$$\_ + \_ == \_ + \_,$$
$$\_ * \_ == \_ * \_ \quad |\!\rangle \in ring[\mathbb{R}] \ .$$

An altogether more interesting ring is the one which lifts the operators point-wise to be operators on functions from reals to reals:

$$\begin{aligned}
\vdash \langle\!| \mathbf{0} &== (\lambda\, r : \mathbb{R} \bullet 0), \\
- &== (\lambda\, f : \mathbb{R} \to \mathbb{R} \bullet (\lambda\, r : \mathbb{R} \bullet -(f\ r))), \\
_ + _ &== (\lambda\, g, f : \mathbb{R} \to \mathbb{R} \bullet (\lambda\, r : \mathbb{R} \bullet (f\ r) + (g\ r))), \\
_ * _ &== (\lambda\, g, f : \mathbb{R} \to \mathbb{R} \bullet (\lambda\, r : \mathbb{R} \bullet (f\ r) * (g\ r)))\ \ |\!\rangle \in \mathit{ring}[\mathbb{R} \to \mathbb{R}]\ .
\end{aligned}$$

Proof of this theorem will entail demonstrating that the functions defined here satisfy the axioms given in *ring*. Thus, for example, we should prove

$$\begin{aligned}
&\forall f, g : \mathbb{R} \to \mathbb{R} \bullet \\
&\quad (\lambda\, g, f : \mathbb{R} \to \mathbb{R} \bullet (\lambda\, r : \mathbb{R} \bullet (f\ r) * (g\ r)))(f, g) \\
&\qquad = (\lambda\, g, f : \mathbb{R} \to \mathbb{R} \bullet (\lambda\, r : \mathbb{R} \bullet (f\ r) * (g\ r)))(g, f)\ .
\end{aligned}$$

The heavy use of lambda abstractions is cumbersome, which is why Brien et al. [6] use categorical notions to come up with a collection of *pointless* definitions, as mentioned above. We use similar definitions here to re-state the conjecture, also generalising to functions from some arbitrary set X to $\mathbb{R}$:

$$\begin{aligned}
[X] \vdash \langle\!| \mathbf{0} &== 0_X, \\
- &== (X \longrightarrow -), \\
_ + _ &== (X \longrightarrow (_ + _)) \circ \square, \\
_ * _ &== (X \longrightarrow (_ * _)) \circ \square\ \ |\!\rangle \in \mathit{ring}[X \to \mathbb{R}]\ .
\end{aligned}$$

The proof obligation above can now be re-stated as

$$(X \longrightarrow (\_ * \_)) \circ \square = (X \longrightarrow (\_ * \_)) \circ \square \circ \square(\mathit{second}, \mathit{first})\ .$$

Use of properties of $\square$ and distributive laws permits a direct (equational) proof of this property, without recourse to quantifiers, lambda abstractions, or points.

3.2 Lifting relations

A lifted version of the infix relational operators can be defined as well. First, we might consider a definition of a set with a partial order and an equivalence relation, in a point-free manner, using familiar notation from the Z mathematical toolkit. (So far, using operators such as '+' and '$*$' in schema definitions has been valid Z. Defining '=' as a schema component however, is not permitted. The symbol which appears here is accordingly larger and bolder than the normal Z equality. The difference is subtle, but in the usual literature there is no attempt to distinguish the symbols.)

orderedSet $[X]$

$$
\begin{array}{l}
\_ < \_ : X \leftrightarrow X \\
\_ = \_ : X \leftrightarrow X \\
\hline
\mathrm{disjoint}\langle (\_ < \_), (\_ < \_)^{\sim}, (\_ = \_) \rangle \qquad [\text{asymmetry/irreflexivity}] \\
(\_ < \_) \circ (\_ < \_) \subseteq (\_ < \_) \qquad [\text{transitivity}] \\
\mathrm{id}\, X \subseteq (\_ = \_) \qquad [\text{reflexivity}] \\
(\_ = \_) = (\_ = \_)^{\sim} \qquad [\text{symmetry}] \\
(\_ = \_) \circ (\_ = \_) \subseteq (\_ = \_) \qquad [\text{transitivity}]
\end{array}
$$

From such a definition, we might construct schemas for *orderedRing*, *orderedField*, and so on, by conjoining the schemas for *ring* (or *field*) and *orderedSet*, and adding the necessary additional axioms. Observe that we could strengthen the definition to cover total orders by replacing 'disjoint' by 'partitions X'. However, this form of lifting is not necessarily the one we want. If X takes a function type, then $e_1 < e_2$ may be expected to be true at some points of application, and false elsewhere. A relation of type $X \leftrightarrow X$ can only record whether e_1 is (or is not) *always* less than e_2.

Instead, let us declare

orderedRing $[X, Y]$

$$
\begin{array}{l}
ring[X \to Y] \\
\_ < \_ : (X \to Y) \times (X \to Y) \to \mathbb{P}\, X \\
\_ = \_ : (X \to Y) \times (X \to Y) \to \mathbb{P}\, X \\
\hline
\forall f, g : X \to Y \bullet \mathrm{disjoint}\langle (f < g), (g < f), (f = g) \rangle \\
\qquad \vdots \\
\forall f, g, h : X \to Y \bullet \\
\qquad (f < g) \subseteq (f + h < g + h) \land \\
\qquad (\mathbf{0} < h) \cap (f < g) \subseteq (f * h) < (g * h)
\end{array}
$$

Now, $e_1 < e_2$ will denote that set of points in the domain of e_1 for which e_1 is less than e_2. (The final predicate might be more naturally expressed using the lifted logical operators, below.)

The required behaviour, then, is

$$
\begin{array}{l}
\forall\, x : X;\ f, g : X \to \mathbb{R} \bullet \\
\qquad x \in (f < g) \Leftrightarrow (f\ x) < (g\ x)\ .
\end{array}
$$

Observe that this is closely related to the approach taken by the boolean-valued functions of Duddy et al. [10] and Brien et al. [6]. In our approach, however, we avoid the need to lift partiality into the predicate language, and so stay within classical Z.

We can define this using a lifting on relations akin to that for functions:

$$
\begin{array}{|l}
\hline
[X, Y] \\
\_ \rightharpoonup \_ : \mathbb{P}\, X \times \mathbb{P}\, Y \to (X \nrightarrow Y) \to \mathbb{P}\, X \\
\hline
\forall\, R : \mathbb{P}\, Y;\ f : X \nrightarrow Y \bullet \\
\quad (X \rightharpoonup R)\ f = \mathrm{dom}(f \rhd R) \\
\hline
\end{array}
$$

Then, for a particular set X, we might define:

$$(\_ < \_) = (X \rightharpoonup (\_ < \_)) \circ \square \ ,$$

and we can assure ourselves that this performs as expected using a sequence of transformational steps:

$$
\begin{array}{ll}
x \in f < g & \\
\Leftrightarrow x \in (\_ < \_)(f, g) & \text{rewriting} \\
\Leftrightarrow x \in ((X \rightharpoonup (\_ < \_)) \circ \square)(f, g) & \text{definition of } < \\
\Leftrightarrow x \in (X \rightharpoonup (\_ < \_))(\square(f, g)) & \text{function composition} \\
\Leftrightarrow x \in \mathrm{dom}(\square(f, g) \rhd (\_ < \_))) & \text{definition of } \rightharpoonup \\
\Leftrightarrow \exists\, y : \mathbb{R} \times \mathbb{R} \bullet (x, y) \in (\square(f, g) \rhd (\_ < \_)) & \text{definition of dom} \\
\Leftrightarrow \exists\, y : \mathbb{R} \times \mathbb{R} \bullet (x, y) \in \square(f, g) \land y \in (\_ < \_) & \text{definition of } \rhd \\
\Leftrightarrow \exists\, y : \mathbb{R} \times \mathbb{R} \bullet y = \square(f, g)\ x \land y \in (\_ < \_) & \text{function application} \\
\Leftrightarrow \exists\, y : \mathbb{R} \times \mathbb{R} \bullet y = (f\ x, g\ x) \land y \in (\_ < \_) & \text{definition of } \square \\
\Leftrightarrow (f\ x, g\ x) \in (\_ < \_) & \text{one-point rule} \\
\Leftrightarrow f\ x < g\ x & \text{rewriting}
\end{array}
$$

3.3 Lifting logical operators

For relations defined in this way, we may introduce some corresponding lifted logical operators.

$$
\begin{array}{|l}
\hline
\textit{liftedLogic}\ [X] \\
\_ \land \_ : \mathbb{P}\, X \times \mathbb{P}\, X \to \mathbb{P}\, X \\
\_ \lor \_ : \mathbb{P}\, X \times \mathbb{P}\, X \to \mathbb{P}\, X \\
\neg : \mathbb{P}\, X \to \mathbb{P}\, X \\
\mathrm{true}, \mathrm{false} : \mathbb{P}\, X \\
\hline
\mathrm{true} \neq \mathrm{false} \\
\forall\, x : \mathbb{P}\, X \bullet \\
\quad x \lor \mathrm{false} = x \quad \land \\
\quad x \land \mathrm{true} = x \quad \land \\
\quad x \lor \neg\, x = \mathrm{true} \quad \land \\
\quad x \land \neg\, x = \mathrm{false} \\
\end{array}
$$

$$
\begin{array}{l}
\forall\, x, y : \mathbb{P}\, X \bullet \\
\quad x \vee y = y \vee x \quad \wedge \\
\quad x \wedge y = y \wedge x \\
\forall\, x, y, z : \mathbb{P}\, X \bullet \\
\quad x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z) \quad \wedge \\
\quad x \vee (y \wedge z) = (x \vee y) \wedge (x \vee z)
\end{array}
$$

Our lifted operators, like those of classical propositional logic, form a boolean algebra. In our lifted domains, we could define a third lifting operator, like $\longrightarrow$ and $\rightharpoonup$, this time on logical operators. However, all we need are the operators of set theory.

$$
\begin{array}{l}
\mathrm{true} = X \\
\mathrm{false} = \varnothing \\
(\_ \wedge \_) = (\_ \cap \_) \\
(\_ \vee \_) = (\_ \cup \_) \\
\forall\, P : \mathbb{P}\, X \bullet \neg\, P = X \setminus P
\end{array}
$$

(That the definition of '$\neg$' cannot be as succinct as the others is perhaps an anomaly in Z.) Implication and equivalence can then be defined in the usual way.

3.4 Lifting results

So far, we have presented various definitions for lifted operators, so that functions can be combined pointwise to form new functions. We have defined lifted relations which permit the succinct expression of predicates over such expressions, and lifted logical operators to combine such relations.

Clearly, any proof of such properties which relies only on the axioms presented here will hold in *any* field, ordered field, boolean algebra, and so on. For example,

$$(a + b) - c = (a - c) + b$$

is a consequence of the ring axioms, and so will hold whether a, b and c are interpreted as real numbers, functions to reals, or members of some other ring. Likewise, the property

$$a \leq b \wedge b \leq a \Leftrightarrow a = b$$

holds on a variety of interpretations.

Though our definitions differ somewhat from those of Brien et al. [6], the following theorem from that paper holds in this context, too.

Theorem *If a function is constructed using unary and binary operations on elements, then lifting distributes fully through its structure.*

4 Applications

In this section we present two applications of the lifting capability defined above. Firstly, a way of making abstract real-time specifications more concise and readable is presented. Secondly, a formalisation of the algebraic reasoning methods traditionally used by process control engineers is given.

4.1 Real-time specification

The behaviour of time-varying systems is traditionally expressed by functions from the time domain [4]. However, stating properties 'pointwise' is often clumsy and verbose; concise specifications require lifting the usual arithmetic and logical operators so that explicit references to the time domain can be avoided. Synchronous specification languages, for instance, have been used to specify real-time systems using lifted equations on variables denoting sequences of values [13,5].

One approach to lifting using the above definitions would be to instantiate the schemas (for instance, *orderedFieldA*) in each schema where they are to be used, but this would be intrusive.

Alternatively, we may make loose global definitions. Ideally, we would like to overload the lifted '$+$' and '$*$' symbols to automatically apply to whichever field argument is appropriate from the context. In Z, we cannot make the operator definitions generic with respect to *fields*, but we can do so with respect to an arbitrary *type*. A potential pitfall is created: we must be careful that for every use of the lifted operators, we have previously defined a suitable instantiation for that type.

$$[X]$$
$$orderedRing[X, \mathbb{R}]$$
$$liftedLogic[X]$$

Because we have used subtly different symbols in the preceding definitions from those in the core language and toolkit, this declaration does not raise problems of redefinition. If we were to require that the normal arithmetic symbols be overloaded (as may be useful), then we should require a changed mathematical toolkit, which contained these declarations, together with axioms for the operators on the reals as an instance.

Spivey's Z rejects the use of loose generic definitions like this, but that of the Standard permits it. Spivey's reasons for rejecting such definitions (where the predicate part of the generic box does not uniquely define the value of the generic constants for each instantiation of the generic parameters) are explained by his concern that multiple models for such paragraphs may give rise to odd consequences [18]. Different instances of the same paragraph (with the same generic parameters) may have different values for the same generic parameter.

This turns out to be an artifact of the modelling he has chosen, and is not a problem in the Standard's semantics, where the loose generics might give rise to multiple models of the whole specification, but within any such model, each individual paragraph will have a fixed meaning. The other problem—that of there possibly not being a model at all—does not arise in the use of generics in this paper.

The hard problem is knowing what to lift, and what not to lift. This is what makes the approach of Duddy et al. [10] complex. For example, given

$$\mid \; speed : \mathbb{T} \rightarrow \mathbb{R}$$

we might write

$$speed < 40 \ .$$

Here, *speed* could be dereferenced implicitly and '$<$' and '40' left unchanged. Alternatively, '$<$' and '40' could be lifted to match the type of *speed*. (It could be argued that *everything* should be lifted with respect to time as is done in equational specification languages [13,5]. However, during formal refinements, we sometimes want the freedom to introduce temporary 'auxiliary' variables to specifications that do not necessarily obey the usual lifting conventions [14].)

One necessary step then, will be to tag all constants as such, so that they can be lifted arbitrarily.

$$\begin{array}{|l} [X, Y] \\ \hline _c : Y \rightarrow X \rightarrow Y \\ \hline \forall\, y : Y \bullet y_c = (\lambda\, x : X \bullet y) \\ \hline \end{array}$$

Most of the time, the '$_c$' annotation will be invisible. Assuming a suitable lifting of '$<$', the predicate above is covered by this lifting scheme.

For example, by introducing a type B to denote brake states, and a time-dependent variable *braking* of that type,

$$B ::= on \mid off$$

$$\mid \; braking : \mathbb{T} \rightarrow B$$

we can use the lifted logical operators to describe a potentially hazardous situation:

$$(200_c < speed) \wedge (braking = off_c) \ .$$

4.1.1 Lifting with respect to intervals

Alternatively, instead of lifting with respect to time, we might lift with respect to intervals of time. Consider the functions α and ω, which return the

endpoints (infimum and supremum) of an interval [10].

$$\alpha, \omega : \mathbb{TI} \rightarrow \mathbb{T}$$

We might wish to use these to specify intervals in which *speed* achieves a greater value at the end than it has at the beginning,

$$speed(\alpha) < speed(\omega) \ ,$$

where α and ω are implicitly dereferenced by the 'current' interval.

It is simple to make these well-typed; we simply lift function application to become function composition. Notice that some authors (though not in the Z community) use an infix dot for function application, so that the above would be written $speed.\alpha < speed.\omega$. Lifting this dot to be a function composition would be entirely natural:

$$(speed \circ \alpha) < (speed \circ \omega) \ .$$

This is now a predicate on intervals. Observe too that this approach works with other explicit dereferencing:

$$speed \circ 3_c < 10_c$$

can also be well-typed (although, as it stands, without any context, its genericity is under-determined).

4.1.2 Simultaneous lifting

Finally, we must consider how to describe simultaneous lifting (with respect to *both* times and time intervals). It is not unreasonable that we should wish to specify *open* intervals in which *speed* is always below its initial value, when braking, for instance:

$$speed < speed \circ \alpha \ .$$

As it stands, this expression is ill-typed.

To rectify this, we define a loose generic function *sel*, which takes a pair as input, and returns whichever component is required by the type. We also introduce *osel*, a function which takes a function as an argument and composes it with *sel*.

$$\begin{array}{|l} [X] \\ \hline sel : (\mathbb{TI} \times \mathbb{T}) \rightarrow X \\ osel : (X \rightarrow \mathbb{R}) \rightarrow ((\mathbb{TI} \times \mathbb{T}) \rightarrow \mathbb{R}) \\ \hline \forall f : X \rightarrow \mathbb{R} \bullet \\ \quad osel\ f = f \circ sel \\ \hline \end{array}$$

$sel[\mathbb{TI}] = first$
$sel[\mathbb{T}] = second$

Using *osel*, we may define a lifted relation which is able to be used between unlike types.

$$[X, Y]$$
$$\_ < \_ : ((X \to \mathbb{R}) \times (Y \to \mathbb{R})) \to (\mathbb{P}(\mathbb{TI} \times \mathbb{T}))$$
$$(\_ < \_) = (\_ < \_) \circ (osel \times osel)$$

This relation may now be used in the specification of a wide range of properties, such as

$$speed < speed \circ \alpha$$

and

$$speed < 40_c \ .$$

As with the corresponding example in Section 2.1, the interpretation of such expressions typically needs care. Each of these expressions denotes a set of interval–time pairs. It is probable, for example, that only those pairs in which the interval contains the time will be of interest.

With these operators we have achieved a systematic means of lifting, where the choice of operator instantiation is determined entirely by the Z type system. By use of point-free definitions, this has been accomplished without significant overhead when these expressions appear in proofs.

4.2 Control theory

In this example, we demonstrate that Z, with lifted operators as defined above, can be used in solving problems in process control theory. This is an area of study with an extensive theory, well-documented in the literature [4,12]. By rendering that theory in Z we make possible checking and analysis using existing Z tools. Directly reusing control theory principles in this way also contrasts favourably with previous work on modelling embedded systems in traditional computing formalisms such as Petri Nets [8], state-machine models [9], state-transition automata [1] or interval calculi [17], where an entirely new modelling approach is developed from scratch.

In an earlier paper [11], a similar problem has been tackled using a Z-like notation and proof tool. That analysis did not make use of the concept of lifting, and the resulting proof was intimidatingly complex.

4.2.1 The problem domain

We assume a 'digital' discrete time domain where the unit of time is one sampling interval, that is, a complete sample/process/output cycle [4]. We have a doubly-infinite time line (we include negative time) so that we are not troubled by boundary conditions when performing arithmetic. Given this time frame we wish to specify system behaviour using *timed traces*.

Definition *A* timed trace *is a function from times to values.*

$$\mathbb{TT} == \mathbb{Z} \rightarrow \mathbb{R}$$

For simplicity here, the values taken by timed trace functions are from the reals, to avoid the distractions of integer arithmetic.

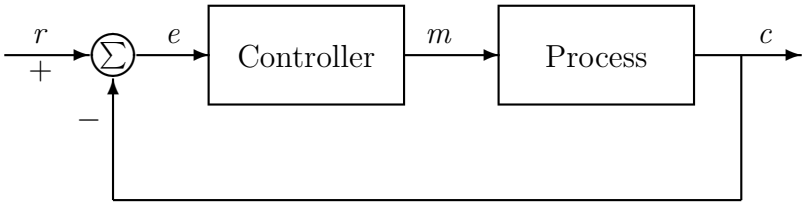

Fig. 2. Typical closed-loop feedback-driven control system.

We wish to model a typical closed-loop *feedback-driven* control system, as shown in Figure 2. The major components are an environmental process and a computerised controller that attempts to influence the behaviour of the process. Input to the system consists of a reference value r, which states the desired value of the controlled output c. The controller's task is to influence the process so that r and c coincide. To do this it monitors the error e, between the value of c and r, and constructs a manipulated control variable m. The process responds to changes in m by adjusting its output c accordingly.

The challenge in describing such a controller is that sluggishness in the process may mean that changes to m take time to influence c, and processing delays in the controller itself may mean that the value of m reflects a stale value of e.

The two components will be described here as difference equations using Z schemas. The definition of the controller will use a *gain constant*, G.

$$\begin{array}{|l} G : \mathbb{R} \\ \hline 0 \leq G \leq 1 \end{array}$$

This constant determines how much impact each unit of measured error e has on the control variable m. The controller simply responds linearly to the error, with a single-unit time delay.

Controller

$$r, c, m : \mathbb{IT}$$

$$\forall z : \mathbb{Z} \bullet$$
$$\quad \textbf{let } e == r(z-1) - c(z-1) \bullet$$
$$\quad\quad m(z) = G * e$$

The definition of the process will depend on a *decay constant*, D, which determines by how much the value of output c will decline at each step if left unchecked.

$$D : \mathbb{R}$$

$$0 \leq D \leq 1$$

The value of the output is linear in this constant, plus the influence of the manipulated control variable, again with a unit delay.

Process

$$m, c : \mathbb{IT}$$

$$\forall z : \mathbb{Z} \bullet$$
$$\quad c(z) = D * c(z-1) + m(z-1)$$

The combined behaviour of these two components is quite complex, as shown in Figure 3. Output c attempts to match the input r but tends to overshoot the desired value and is then forced to correct itself. This is due to the overall two-unit time delay in responsiveness—when trying to match the desired value, the system actually passes it before the controller can recognise that the goal has been achieved. Smaller values of G will decrease the size of the overshoot but will make the system take longer to converge on the desired value [16]. Furthermore, the constant downward influence of the decay behaviour causes the output to stabilise slightly below the desired value (a simple linear controller cannot correct this).

The challenge now is to describe the overall behaviour of the system. We claim that the schema *System* does this.

System

$$r, c : \mathbb{IT}$$

$$\forall z : \mathbb{Z} \bullet$$
$$\quad c(z) - D * c(z-1) + G * c(z-2) = G * r(z-2)$$

The claim may be stated formally:

$$\forall r, c : \mathbb{IT} \bullet (\exists m : \mathbb{IT} \bullet$$
$$\quad \mathit{Controller} \wedge \mathit{Process} \Leftrightarrow \mathit{System}) \ .$$

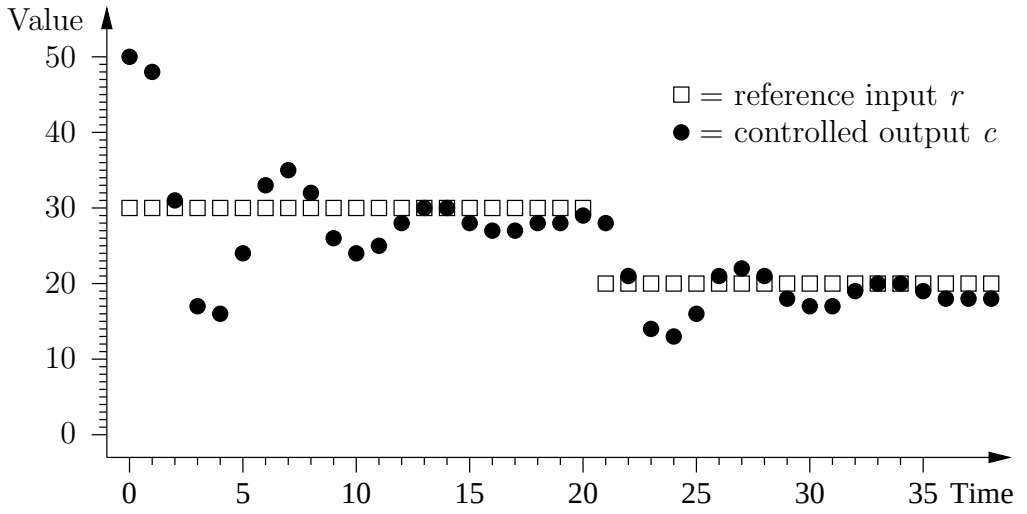

Fig. 3. Behaviour of system with $G = 0.7$, $D = 0.95$, and initial values (at time 0) $c = 50$ and $m = 0$.

4.2.2 Using lifted operations

In general the solution of such difference equations is hard. (Though in the case of the claim above, the reader may not find the algebraic manipulations too difficult.) To overcome this, the approach traditionally taken uses *transfer functions* to simplify the algebraic manipulations [4].

Definition *A* transfer function *is a function over timed traces.*

$$\mathbb{TF} : \mathbb{P}(\mathbb{TT} \to \mathbb{TT})$$

In order to perform the necessary algebraic manipulations, we shall require that additional properties hold for members of the set $\mathbb{TT}$, as explained below. Each component in a control system can be described by such a function: its input is a timed history and it produces as output another timed history.

In this example we define a single basic transfer function, the *backwards shift* operator. It represents a delay of one time unit.

$$\begin{array}{|l} \mathcal{B} : \mathbb{TF} \\ \hline \forall f : \mathbb{TT} \bullet \\ \quad \mathcal{B}\ f = \{\ z : \mathbb{Z} \bullet (z, f(z-1))\ \} \end{array}$$

Observe that, using the normal notation for function (relation) iteration, we have

$$\mathcal{B}^n f = \{\ z : \mathbb{Z} \bullet (z, f(z-n))\ \}\ .$$

This notation gives a clue as to the lifting we will undertake. We shall use an instance of the generic (lifted) operators instantiated so that addition is pointwise (doubly) lifted.

$$(\_ + \_)[\mathbb{TF}] = (\mathbb{T} \longrightarrow (\mathbb{Z} \longrightarrow (\_ + \_)) \circ \square) \circ \square$$
$$\mathbf{0}[\mathbb{TF}] = \mathbf{0}_{cc}$$
$$-[\mathbb{TF}] = (\mathbb{T} \longrightarrow (\mathbb{Z} \longrightarrow (-)))$$
$$\mathbf{1}[\mathbb{TF}] = \mathrm{id}\,\mathbb{T}$$

So far these definitions conform with the *field* axioms described in Section 3.1. Unfortunately, however, the correspondence is not perfect. Traditionally, a transfer function is defined as the 'ratio' of an arbitrary input trace to the consequent output trace [4, p. 29]. The 'product' of transfer functions [12, p. 17] is actually functional composition, rather than a lifting of the '$*$' operator, and the corresponding multiplicative inverse is a total, rather than partial, function.

$$(\_ * \_)[\mathbb{TF}] = (\_ \circ \_)$$
$$(\_^{-1})[\mathbb{TF}] = (\_^{\sim})$$

Nevertheless, these definitions have the algebraic properties needed for transfer-function arithmetic to work just like its numerical counterpart [4, §2.4].

From here we can support easy expression of properties by defining binary negation and division operators as abbreviations using the core lifted operators.

$$(\_ - \_)[X] == (\_ + \_)[X] \circ (\mathrm{id}\, X \times -)$$
$$(\_ \,/\, \_)[X] == (\_ * \_)[X] \circ (\mathrm{id}\, X \times (\_^{-1}))$$

In addition, it is often necessary to multiply a transfer function by a 'scalar' coefficient; this is defined using multiplication lifted pointwise.

$$[X, Y]$$
$$\_ \cdot \_ : Y \times (X \to Y) \to (X \to Y)$$
$$\forall\, y : Y;\ f : X \to Y;\ x : X \bullet$$
$$(y \cdot f)\ x = y * (f\ x)$$

4.2.3 Problem re-stated

We can now rewrite our schemas concisely using transfer functions (and no points). These definitions are logically identical to the previous ones. For example, using the following calculation,

$$\begin{aligned}
&(\forall\, z : \mathbb{Z} \bullet c(z) = D * c(z-1) + m(z-1)) \\
&\Leftrightarrow (\forall\, z : \mathbb{Z} \bullet c(z) = D * (\mathcal{B}\, c)z + (\mathcal{B}\, m)z) \\
&\Leftrightarrow (\forall\, z : \mathbb{Z} \bullet c(z) - D * (\mathcal{B}\, c)z = (\mathcal{B}\, m)z) \\
&\Leftrightarrow (\forall\, z : \mathbb{Z} \bullet ((\mathbf{1}_c - D_c \cdot \mathcal{B})c)z = (\mathcal{B}\, m)z) \\
&\Leftrightarrow (\mathbf{1}_c - D_c \cdot \mathcal{B})c = \mathcal{B}\, m \\
&\Leftrightarrow c = (\mathcal{B} \,/ (\mathbf{1}_c - D_c \cdot \mathcal{B}))\ m\ ,
\end{aligned}$$

the *Process* specification can be re-expressed as follows:

ProcessA

$$m, c : \mathbb{IR}$$

$$c = (\mathcal{B} \,/\, (\mathbf{1}_c - D_c \cdot \mathcal{B}))\ m$$

Similarly for the *Controller* and *System* specifications.

ControllerA

$$r, c, m : \mathbb{IR}$$

$$\textbf{let}\ e == r - c \bullet$$
$$m = (G_c \cdot \mathcal{B})\ e$$

SystemA

$$r, c : \mathbb{IR}$$

$$c = ((G_c \cdot \mathcal{B}^2) \,/\, (\mathbf{1}_c - D_c \cdot \mathcal{B} + G_c \cdot \mathcal{B}^2))\ r$$

(For some transfer function F, with input r and output c, control theory literature normally shifts argument r to the left and expresses such equalities as 'ratios', $c/r = F$ [4, p. 31]. To avoid confusing function application with the $*$ and $/$ operators, we prefer not to use this particular form of expression.)

4.2.4 Proof

We are now in a position to be able to prove the property stated at the end of Section 4.2.1. Process control theory provides a number of *block diagram laws* that characterise commonly-occurring designs as transfer functions [12, §2.3.1]. For a closed-loop system such as that in Figure 2, where the controller and process are defined by transfer functions $m = Q\ e$ and $c = R\ m$ respectively, the whole system is defined by the *closed loop transfer function*:

$$c = ((Q * R) \,/\, (\mathbf{1}_c + Q * R))\ r\ .$$

Therefore, we can reason as follows:

$$\mathit{Process} \wedge \mathit{Controller}$$
$$\Leftrightarrow \qquad \text{Closed loop law}$$
$$c = ((G_c \cdot \mathcal{B}) * (\mathcal{B} \,/\, (\mathbf{1}_c - D_c \cdot \mathcal{B})) \,/$$
$$(\mathbf{1}_c + (G_c \cdot \mathcal{B}) * (\mathcal{B} \,/\, (\mathbf{1}_c - D_c \cdot \mathcal{B}))))\ r$$
$$\Leftrightarrow \qquad \text{Multiplication}$$
$$c = (((G_c \cdot \mathcal{B}^2) \,/\, (\mathbf{1}_c - D_c \cdot \mathcal{B})) \,/$$
$$((\mathbf{1}_c - D_c \cdot \mathcal{B} + G_c \cdot \mathcal{B}^2) \,/\, (\mathbf{1}_c - D_c \cdot \mathcal{B})))\ r$$

$\Leftrightarrow$ Multiplicative inverse

$c = ((G_c \cdot \mathcal{B}^2) \,/\, (\mathbf{1}_c - D_c \cdot \mathcal{B} + G_c \cdot \mathcal{B}^2))\ r$

$\Leftrightarrow$ Definition

$System$.

This straightforward algebraic proof using lifted operators is dramatically simpler than performing such proofs without lifting. In an earlier paper [11], we undertook a machine-assisted proof of a similar property using Z, but without any lifting apparatus. Even though we there dealt with the considerably simpler case of single-unit delay, and hence no overshoot behaviour, the proof was nevertheless far more complex.

5 Conclusion

In this paper we have reviewed two previous descriptions of lifting in Z. In these, results from one algebra (typically the real field) were lifted into another (functions from some set to the reals). One aim of this paper is to bring to a wider audience material which has previously existed only in unpublished technical reports [6,10].

We have also demonstrated a version of these two lifting approaches which remains *entirely* within Standard Z, and which should therefore be amenable to analysis using the growing collection of tools available for Standard Z. For instance, this paper has been successfully checked with the *f*uzz typechecker. (Since $\mathbb{R}$ is not a type defined in *f*uzz, $\mathbb{Z}$ was used instead—this nevertheless gave us all the typing properties needed. Also, the overloading of existing Z symbols was resolved above by making them different glyphs from those used in the toolkit: in each case they are larger. Such an approach appears to be consistent with that taken by the Z Standard [19]. The *f*uzz tool parses the macro names used to generate these symbols, and thus encountered no ambiguities.) We are also developing a theorem prover-based implementation of lifted functions for reasoning about real-time systems [7].

Finally, we demonstrated the utility of this approach with a number of examples. Real-time specification uses pointwise lifting in the style described earlier; control theory uses a lifting to higher-order functions. Both are accessible using our lifted definitions.

Acknowledgements

This research was funded by the Information Technology Division of the Australian Defence Science and Technology Organisation and by Australian Research Council Large Grant A49702415, *Efficient Development of Verified Concurrent Real-Time Programs through Tool Support*. We are indebted to the authors of the unpublished manuscripts cited as Brien et al. [6] and Duddy et al. [10] for the seminal ideas which have contributed to this paper. Several

anonymous referees made detailed and helpful comments on earlier drafts of this paper.

References

[1] R. Alur, C. Courcoubetis, T. A. Henzinger, and P.-H. Ho. Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. In R. L. Grossman, A. Nerode, A. P. Ravn, and H. Rischel, editors, *Hybrid Systems*, volume 736 of *Lecture Notes in Computer Science*, pages 209–229. Springer-Verlag, 1993.

[2] J. Backus. Can functional programming be liberated from the von Neumann style? *Communications of the ACM*, 21(8):613–641, 1978.

[3] E. J. Billington, D. Donovan, B. D. Jones, S. Oates-Williams, and A. Street. *Discrete Mathematics: Logic and Structures*. Longman, 1990.

[4] J. G. Bollinger and N. A. Duffie. *Computer Control of Machines and Processes*. Addison-Wesley, 1988.

[5] M. Le Borgne, H. Marchand, E. Rutten, and M. Samaan. Formal verification of SIGNAL programs: Application to a power transformer station controller. In M. Wirsing and M. Nivat, editors, *Algebraic Methodology and Software Technology*, volume 1101 of *Lecture Notes in Computer Science*, pages 271–285. Springer-Verlag, 1996.

[6] S. M. Brien, M. Engel, He Jifeng, A. Ravn, and H. Rischel. Z description of duration calculus. Draft ProCos II Project document OU HJF 12/2, Oxford University Computing Laboratory, August 1993.

[7] A. Cerone. Axiomatisation of an interval calculus for theorem proving. In C. J. Fidge, editor, *Computing: The Australasian Theory Symposium 2001*, volume 42 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2001. To appear.

[8] J. E. Coolahan, Jr. and N. Roussopoulos. Timing requirements for time-driven systems using augmented Petri Nets. *IEEE Transactions on Software Engineering*, SE-9(5):603–616, September 1983.

[9] C. DaSilva, B. Dehbonei, and F. Mejia. Formal specification in the development of industrial applications: Subway speed control system. In M. Diaz and R. Groz, editors, *Formal Description Techniques, V*, pages 199–213. Elsevier, 1993.

[10] K. Duddy, L. Everett, C. Millerchip, B. Mahony, and I. J. Hayes. Z-based notation for the specification of timing properties. Draft, Department of Computer Science, University of Queensland, June 1995.

[11] C. J. Fidge, P. Kearney, and A. P. Martin. Applying the Cogito program development environment to real-time system design. In C. McDonald, editor, *Computer Science '98: Proc. 21st Australasian Computer Science Conference*, pages 367–378. Springer-Verlag, 1998.

[12] G. F. Franklin and J. D. Powell. *Digital Control of Dynamic Systems*. Addison-Wesley, 1980.

[13] N. Halbwachs, F. Lagnier, and C. Ratel. Programming and verifying real-time systems by means of the synchronous data-flow language LUSTRE. *IEEE Transactions on Software Engineering*, 18(9), September 1992.

[14] I. J. Hayes. Real-time program refinement using auxiliary variables. In M. Joseph, editor, *Sixth International School and Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT 2000)*, volume 1926 of *Lecture Notes in Computer Science*, pages 170–184. Springer-Verlag, September 2000.

[15] S. Mac Lane. *Categories for the Working Mathematician*, volume 5 of *Graduate Texts in Mathematics*. Springer-Verlag, New York, second edition, 1998.

[16] J. R. Leigh. *Control Theory: A Guided Tour*, volume 45 of *IEE Control Engineering Series*. Peter Peregrinus Ltd, 1992.

[17] S. Nadjm-Tehrani and J.-E. Strömberg. Formal verification of dynamic properties in an aerospace application. *Formal Methods in System Design*, 14(2):135–169, March 1999.

[18] J. M. Spivey. *Understanding Z: A Specification Language and its Formal Semantics*, volume 3 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, January 1988.

[19] I. Toyn, editor. *Z Notation*. Number 13568.2. Z Standards Panel, ISO Panel JTC1/SC22/WG19 (Rapporteur Group for Z), 1999. 'Final Committee Draft'.

[20] P. Wadler. Monads for functional programming. In Manfred Broy, editor, *Program Design Calculi*, NATO ASI Series F, pages 233–264. Springer-Verlag, 1993. Marktoberdorf International Summer School, 1992.

[21] Zhou Chaochen. Duration calculi: An overview. In D. Bjorner, M. Broy, and I. Pottosin, editors, *Formal Methods in Programming and Their Applications*, volume 735 of *Lecture Notes in Computer Science*, pages 256–266. Springer-Verlag, 1993. Extended abstract.