

**SOFTWARE VERIFICATION RESEARCH CENTRE**  
**DEPARTMENT OF COMPUTER SCIENCE**  
**THE UNIVERSITY OF QUEENSLAND**

Queensland 4072  
Australia

**TECHNICAL REPORT**

No. 94-40

**Animating Z: Interactivity,  
Transparency and Equivalence**

Mark Utting

December 1994

Phone: +61 7 365 1003

Fax: +61 7 365 1533

**Note:** Most SVRC technical reports are available via anonymous ftp, from `ftp.cs.uq.edu.au` in the directory `/pub/SVRC/techreports`.

# Animating Z: Interactivity, Transparency and Equivalence

Mark Utting  
Software Verification Research Centre  
The University of Queensland  
Email: `marku@cs.uq.edu.au`

## Abstract

The ability to animate Z specifications is useful in allowing a specifier to explore the behaviour of a specification. This paper defines three new evaluation criteria for animation systems, interactivity, transparency and operational equivalence. It also describes a simple Haskell-based animation system that satisfies these criteria.

A system that allows Z specifications to be animated can assist a specifier to understand and validate a specification. For instance, the satisfiability of an operation can be checked using some given test data. As a specification is written, the specifier may wish to experiment with alternative definitions, perhaps to check that they are computable and give the expected results. Animation can provide immediate feedback during the process of writing a specification and this may help to reduce specification errors.

Of course, not all Z specifications are directly executable [HJ89]. However, several researchers have reported success in animating subsets of Z, typically by translating schemas into Prolog or into a functional language and transforming the resulting programs to achieve acceptable efficiency [DKC89, JS89, Dil90, DN91].

Recently, Breuer and Bowen have developed an approach to establishing the correctness of a Z animation system [BB94]. In addition to correctness, they identified three other measures which can be used to evaluate and compare animation systems:

**coverage:** How many Z constructs are handled?

**efficiency:** How quickly is an animation evaluated?

**sophistication:** How many of the animations terminate?

In this paper an additional three measures are proposed: *interactivity*, *transparency* and *operational equivalence*. The underlying assumption of the paper is that the *primary* purpose of an animation system is to help a specifier *explore* the consequences

of a specification, rather than produce a final implementation of the system or even a full scale prototype that is capable of handling realistically-sized data sets.

The three additional measures are defined and discussed in the next section. The following sections outline a simple approach to animation that rates highly according to these three measures.

## 1 Three Measures of an Animation System

First, some terminology. We shall say that animation systems contain a *translator*, which translates (some)  $Z$  constructs into an *animation language* (e.g., Prolog), and an *evaluator*, which evaluates commands expressed in the animation language (e.g., a Prolog interpreter). Note that if the animation language is  $Z$  itself, then the translator may simply be the identity function.

### 1.1 Interactivity

An animation system has high *interactivity* if the specifier can enter  $Z$  terms, formulae and schema expressions for immediate evaluation. A system with high interactivity should also allow the results of evaluations to be named, so that those results can be used in following queries. High interactivity is important when animation is being used for exploratory purposes.

High interactivity may be achieved either by allowing the specifier to interact directly with the evaluator (using the animation language), or by providing a translator that is fast enough to make interaction practical and that accepts  $Z$  terms and formulae rather than just complete schemas or specifications.

### 1.2 Transparency

The *transparency* of an animation system is the degree of similarity between the input and the output of the translator. For instance, if the translator is the identity function then the animation system has high transparency, whereas if the translator transforms  $Z$  to a language with quite different syntactic structure and reorders and optimises the specification, then the animation system has low transparency.

High transparency makes it easier for the specifier to understand how some construct in the animation language corresponds to some  $Z$  construct and vice versa. This is important so that inputs and outputs of the evaluator are easily understood as  $Z$  terms and are not ambiguous. For instance, an evaluator that outputs both sets and sequences as lists is likely to lead to confusion.

Furthermore, because animation systems cannot have complete coverage of  $Z$  and still guarantee termination, it is generally useful to allow a specifier to manually modify the result of translation, to improve termination or efficiency properties. Such modifications are much easier in a highly transparent animation system.

### 1.3 Operational Equivalence

An animation system supports *operational equivalence* if it ensures that each animated operation is *equivalent* to its corresponding Z specification, rather than allowing it to be any *refinement* [Rob87, Mor90] of the specification.

The usual Z refinement relation allows an implementation to be more deterministic than the specification and to have a weaker precondition (that is, accept more inputs). However, for animation purposes, it is desirable to ensure that the animation of each specified operation is *equivalent* to its Z specification, rather than a proper refinement of it. The rationale for this is that it avoids the possibility of making inferences about the animation that are not necessarily true of the specification (and therefore, not necessarily true of *all* eventual implementations).

## 2 A Haskell-based Animation System

In this section, a simple animation system is described, based on translation into the functional programming language Haskell [HPe92]. Some of the reasons for using Haskell are that there are reasonably efficient implementations freely available, its type system is similar to that of Z (in particular, it can infer the implicit parameter types of Z generic operators such as  $\cup$ ), it provides unbounded integers and free type definitions that are similar to those of Z and it uses lazy evaluation, which helps to increase the chance of termination with some specifications.

The language being animated in this paper is Cogito-SL1 [TKK<sup>+</sup>95], which is essentially Z extended with module constructs. SL1 also includes imperative programming language constructs to support refinement and translation to Ada, but these will not be needed in this paper. In the rest of this paper, we will call our specification language Z when discussing features that are common to both Z and SL1 and call it SL1 when discussing SL1-specific constructs such as modules.

This animation system could also be applied to standard Z. However, the main advantage of using SL1 is that the typical ADT structure of a specification (state schema, initialisation and operation schemas) is stated formally within an SL1 module, whereas it is informal in Z. The translation outlined here takes advantage of the modular structure of an SL1 specification by generating one Haskell ADT for each SL1 module. In addition, SL1 allows the precondition of a schema to be separated out as a separate component (thus introducing a proof obligation) and our translation can take advantage of this.

### 2.1 Translation of Terms and Formulae

To increase transparency as much as possible, we raise the level of Haskell programming to match that of the specification language as closely as possible. This is done by using a Haskell library module that implements the (finite) set, relation

Z expression	Haskell expression
$\mathbb{F} X$	Set X
$S = T$	S == T
$S \neq T$	S /= T
$S \in T$	S 'in' T
$S \notin T$	S 'not_in' T
$\{\}$	set []
$\{a, b, c\}$	set [a,b,c]
$S \subseteq T$	S 'subset_of' T
$S \cup T$	S 'union' T
$S \cap T$	S 'inter' T
$S \setminus T$	S 'diff' T
$\bigcup S$	gen_union S
$\bigcap S$	gen_inter S
$\{x : S \mid P(x) \bullet f(x)\}$	set [f x   x <- set2list S, P x]

Table 1: Set expressions in Z and Haskell.

and function data types of Z. Tables 1 and 2 show how close the correspondence is between expressions in Z and Haskell. This makes the translation of Z terms and formulae to Haskell relatively trivial. In most cases, there is a one-to-one mapping between tokens in the specification language and the tokens in the corresponding Haskell expressions.

The Haskell implementation of Z data types defines a relation as a set of pairs. Thus `Reln A B` is simply an abbreviation for `Set (A,B)`. This means that the set operators shown in Table 1 can all be used on relations, as in Z. However, the additional structure inherent in relations makes it possible to define many more operators that are applicable only to relations. Table 2 shows the equivalent Haskell expression for various common Z operators on relations. In Table 2, `X` and `X'` may be any type that supports equality, including infinite types, such as the set of all integers. However, `S` must be a finite set, since the Haskell set operators can only manipulate finite sets. Note that this means that the Haskell implementation of relations only implements *finite* subsets of  $X \leftrightarrow X'$ . That is, it implements  $\mathbb{F}(X \times X')$  rather than the usual Z meaning of  $X \leftrightarrow X'$ , which is  $\mathbb{P}(X \times X')$ .

The restriction to finite sets is the major difference between some of the Haskell functions and the corresponding Z operators. In theory, it would be possible to handle some kinds of infinite sets using the lazy evaluation of Haskell, by encoding an infinite set as an infinite list. However the restriction to finite sets was made so that sets could be represented as sorted lists with no duplicates. This representation has the disadvantage that turning an infinite list into a set does not terminate, so we are effectively limited to finite sets. However it has the significant advantage that

Z expression	Haskell expression
$X \leftrightarrow X'$	<code>Reln X X'</code>
$\text{dom } R$	<code>dom R</code>
$\text{ran } R$	<code>ran R</code>
$S \triangleleft R$	<code>S 'domres' R</code>
$S \trianglelefteq R$	<code>S 'domsub' R</code>
$R \triangleright S$	<code>R 'ranres' S</code>
$R \trianglerighteq S$	<code>R 'ransub' S</code>
$R \circledast R'$	<code>R 'semi' R'</code>
$R \langle S \rangle$	<code>R 'image' S</code>
$R \sim$	<code>inverse R</code>
<code>id</code>	<code>identity S</code>
$R^k$	<code>R 'iter' k (for <math>k &gt; 0</math>)</code>
$R^+$	<code>transclosure R</code>
$R^*$	<code>transclosure R 'union' identity S <sup>1</sup></code>

Table 2: Relation expressions in Z and Haskell.

most set operations such as union and intersection have  $O(N)$  complexity rather than  $O(N^2)$  complexity.

Table 3 shows the correspondence between logical formulae in Z and Haskell.

This Haskell library is freely available from the author. The Haskell code for the library is extracted automatically from a technical report which describes the library [Utt94].

## 2.2 Translation of Modules and Schemas

To explain the translation of schemas and modules, we shall use the typical SL1 module  $M1$ , shown in Figure 1, as an example.  $Inv$ ,  $InitPred$ ,  $Post1$  etc. stand for arbitrary formulae over their argument variables. In SL1 all operation schemas implicitly include  $\Delta State$  in their signature, so the full signature of  $Op1$  is  $[a, b, a', b' : T; x? : Tx, y! : 0..10]$ . The *changes\_only*  $\{b\}$  notation in  $Op1$  means that all state variables other than  $b$  do not change, so it is equivalent to  $a' = a$ .  $Pre1$  is the explicit precondition for  $Op1$ .

From the  $M1$  module, the translation process generates a Haskell module of the same name. The header of the module lists the exported names and includes the Z library module mentioned above.

```
> module M1( M1_state, M1, m1_inv, m1_init, m1_op1 )
> where
> import ZTypes
```

Z expression	Haskell expression
true	True
false	False
$P \wedge Q$	$P \ \&\& \ Q$
$P \vee Q$	$P \    \ Q$
$P \Rightarrow Q$	$P \ \text{'implies'} \ Q$
$P \Leftrightarrow Q$	$P \ \text{'iff'} \ Q$
$\neg P$	not P
$(\exists x : T \mid P \bullet Q)$	or [Q   x <- set2list T; P]
$(\forall x : T \mid P \bullet Q)$	and [Q   x <- set2list T; P]
$(\forall x : Tx; y : Ty \mid P \bullet Q)$	and [Q   x<-set2list Tx; y<-set2list Ty; P]

Table 3: Formulae in Z and Haskell.

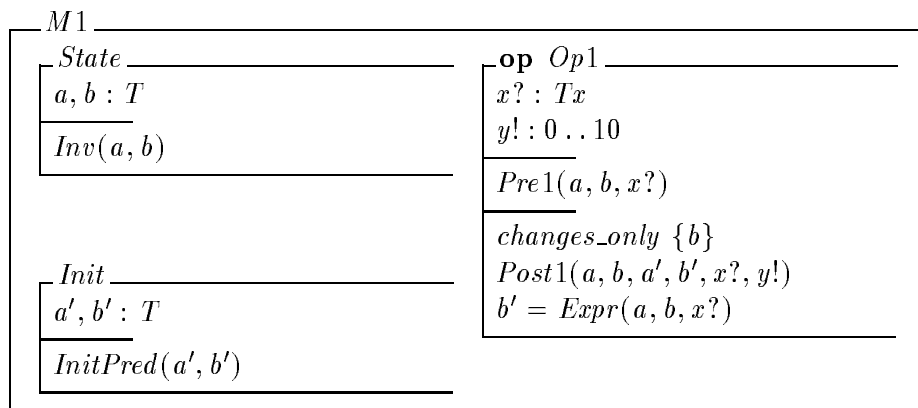


Figure 1: An SL1 module called *M1*

The *State : Exp* schema of the *M1* module is translated to a Haskell constructor called `M1_state`, plus a predicate `m1_inv` that checks the invariant *Inv*.

```
> data M1_state = M1 Ta Tb deriving (Eq,Ord,Text)
>
> m1_inv :: M1_state -> Bool
> m1_inv (M1 a b) = Inv(a,b)
```

The `data` definition defines a new product type called `M1_state` and automatically derives appropriate equality, ordering and printing and parsing functions for that type. Defining a new type in this way has two advantages: it forces the Haskell type checker to be more strict (two schemas that happen to contain identical component types will still not be type compatible) and it provides us with a constructor (`M1`) that can be used in pattern matching.

Experience has shown us that it is worth treating equalities in the invariant that have the form *Variable = Expr* in a special way. We expand them out into the definition of each operation explicitly, and define an `M1_internal_inv` function that contains the remainder of the invariant. Unlike `M1_inv`, the `M1_internal_inv` function would not be exported from the `M1` module. The case study in Section 3 illustrates this.

Next we consider the translation of operation schemas, including the *Init* operation. Our intention is to translate each operation into a Haskell function that takes the inputs of the operation (e.g., *x?*) and the initial state of the module as parameters, and returns the new module state, together with any outputs of the operation. Note that we are relying on the *Z* conventions for specifying sequential systems here, with *x?* and *y* being interpreted as inputs to the operation and *y'* and *z!* being interpreted as outputs. These conventions are enforced in SL1 whenever a schema is marked as an operation schema, using the `op` keyword.

In general, some operations may be non-deterministic. That is, the predicate that defines the operation may allow several different outputs for a given input, rather than uniquely defining the outputs. This means that non-deterministic operations cannot be directly translated into functions in the obvious way if we wish our animation system to support operational equivalence.

Four possible ways of translating a non-deterministic operation are:

1. refining it so that it is deterministic, then implementing it as a function. This would mean sacrificing operational equivalence.
2. implement it as a relation. That is, require the caller to supply the result value as an *input* parameter and define the function to return true or false according to whether that value satisfies the specification. This amounts to implementing the characteristic function of the schema's predicate.
3. implement it as a function that returns the set of all possible results. This uses the fact that  $X \leftrightarrow Y$  is isomorphic to  $X \rightarrow \mathbb{P} Y$ .

4. implement it as a backtracking function that returns a single result each time it is called, but can be re-invoked to explore alternate results, if that is desired.

The most obvious choice for a functional animation language is perhaps (1), but here we shall take the last alternative so as to preserve operational equivalence between specifications and animations.

The standard way of implementing backtracking in lazy functional languages is to return the set of all possible results, then only access (and therefore evaluate) those results that are of interest. This means that there is little difference between alternatives (3) and (4) above, except for the way in which we use the resulting set of outputs. The following definitions define a special data type called `Solutions` that encapsulates this notion of backtracking. It contains a list of alternative outputs that are generated by an operation, plus the name of the operation (the latter is to enable more meaningful messages to be produced).

```
> data Solutions a = Operation String [a] deriving (Eq,Ord,Text)
```

Using this scheme to represent backtracking, the *Init* operation translates to:<sup>2</sup>

```
> m1_init :: () -> Solutions M1_state
> m1_init ()
>   = Operation "m1.init"
>     [M1 a' b'
>       | a' <- set2list T;
>         b' <- set2list T;
>         InitPred(a',b');
>         m1_inv (M1 a' b')]
```

This works by generating all possible values for  $a'$  and  $b'$ , then testing those values to select the ones which satisfy  $InitPred(a', b')$  and the state invariant. Obviously, this naive translation is not very efficient, but we shall show how it can be improved shortly.

A naive translation of operation *Op1* is similar, except that it has input parameters and each solution is a pair, containing a final state and an output value for  $y!$ . The *changes\_only*  $\{b\}$  predicate expands to  $a' = a$ .

```
> m1_op1 :: Tx -> M1_state -> Solutions (M1_state,Ty)
> m1_op1 x_in (M1 a b)
>   = Operation "m1.op1"
>     [(M1 a' b', y_out)
>       | a' <- set2list T;
```

---

<sup>2</sup>The empty parameter to the `m1_init` function is unfortunately necessary, because the Haskell implementation being used (Yale Haskell, version Y1.2) does not allow constants to be exported from modules, only functions.

```

>      b' <- set2list T;
>      y_out <- [0..10];
>      Pre1(a,b,x_in);
>      a' = a;
>      Post1(a,b,a',b',x_in,y_out);
>      b' = Expr(a,b,x?);
>      m1_inv (M1 a' b')]

```

However, we can improve on this translation in several ways:

- We take advantage of the explicit precondition of *Op1* to move the evaluation of *Pre1* to the beginning of the list comprehension (before all the `<-` generators), so that it is only evaluated once, rather than many times.
- We add the input argument into the documentation string, so that it is more obvious how the operation was parameterised. (It would also be possible to put the input state variables in too, if desired).
- We move the  $b' = Expr(a, b, x?)$  predicate out of the list comprehension, whenever *Expr* only depends upon input variables. However, unless we can verify statically that  $Expr(a, b, x?)$  is in the type of  $b'$ , we must still check this in the list comprehension, to ensure that no solutions are generated if it is not true.
- Similarly, the  $a' = a$  that results from the *changes\_only*  $\{b\}$  predicate can be moved out into a **where** definition. No type checking is needed in this case, because  $a$  is already type checked.
- To improve readability, we give a name to the final state by defining it in the **where** clause.

These improvements mean that *Op1* translates to the following more efficient function. It is still quite easy to see how each component of this function has been derived from the original schema, so we have attained a high degree of transparency.

```

> m1_op1 :: Tx -> M1_state -> Solutions (M1_state,Ty)
> m1_op1 x_in (M1 a b)
>   = Operation
>     ("m1.op1(" ++ show x_in ++ ")")
>     [(state', y_out)
>      | Pre1(a,b,x_in);
>        b' 'in' T;
>        y_out <- [0..10];
>        Post1(a,b,a',b',x_in,y_out);
>        m1_inv state']
>   where
>     a' = a

```

```

>      b' = Expr(a,b,x?)
>      state' = M1 a' b'

```

The only source of non-determinism in this comes from the choice of  $y!$ . If  $y!$  was also defined by equality, as is often the case, then this operation would be fully deterministic, always producing either an empty list of solutions (indicating that the inputs did not meet the precondition) or a singleton list containing the only possible answer.

### 2.3 Interacting with an Animation

There are many different ways of using Haskell to interact with an animation similar to the one described in the last section. However, here we shall describe a simple scheme that we have found to be quite effective.

Firstly, note we can evaluate any operation directly, giving it some input values, and immediately see all the possible outputs that it produces. For example, the expression `m1_init ()` will return all states that satisfy the *Init* operation.

If we are only interested in seeing one output (the *first* one perhaps), then the following functions are useful. The `soln` function takes a number  $i$  ( $0 \leq i$ ) and returns the solution obtained by backtracking  $i$  times. The `>>` function is forward functional composition, which allows us to execute operations in a left-to-right order.

```

> soln :: Int -> Solutions a -> a
> soln _      (Operation name [])
>           = error ("No more solutions to " ++ name)
> soln 0      (Operation name (x:xs))
>           = x
> soln (n+1) (Operation name (x:xs))
>           = soln n (Operation name xs)

> infixl 0 >>
> f >> g = g . f

```

This allows us to enter `m1_init () >> soln 1` to obtain the first solution from the *Init* operation. Since we have followed the convention that the state is always the last parameter to each function, this allows a sequence of update operations to be composed together as higher order functions. For operations that produce a pair of outputs (the updated state and other outputs) we can use the standard `fst` and `snd` pair selection functions to choose which we want to see.

For example, if the type  $Tx$  is an enumerated type comprised of several colours, the following expression returns the  $y!$  output that results from two applications of the *Op1* operation (and checks that *Op1* is non-deterministic and has at least three solutions).

```

m1_init () >> soln 1 >>
m1_op1 Red >> soln 1 >> fst
m1_op1 Blue >> soln 3 >> snd

```

This style of interaction provides a way of testing sequences of operations by interacting directly with the Haskell interpreter. It is also possible to define more complex styles of interaction, such as printing a trace of the inputs and outputs of each operation as it is executed, or reporting which operations are deterministic and which are not.

### 3 Case Study: A Dependency Management System

To illustrate the animation system, we shall use a *dependency management system* (DMS) case study. This is an ADT which manages a graph of dependencies between various nodes and provides operations that ensure that no circularities are introduced into the graph. A typical use of such a dependency management system would be to record dependencies between theorems in a theorem prover to ensure that no circular reasoning is allowed. Figure 2 contains an SL1 specification of the key parts of the DMS.

This module has a type parameter, which determines the type of the nodes of the DMS graph. To handle this, we can either translate this module to a Haskell module that exports a `DMS` type that takes a parameter, or simply instantiate the `NS` type for the purposes of animation. For simplicity, we shall take the latter alternative here and instantiate `NS` to be integers.

The Haskell script that results from translating this DMS specification consists of all the lines in this section that begin with `>`.

```

> module DMS( Dms_state, Dms(..), dms_inv, dms_Init,
>             dms_AddNode, dms_CanAdd,
>             dms_AddDependence, dms_Dependents,
>             dms_SomeDirectDependent
>             -- and other operations...
>             )
> where
> import ZTypes

> data NS = Integer

> data Dms_state
>   = Dms (Set NS)      -- the known nodes
>         (Reln NS NS) -- dir_dep_on
>         (Reln NS NS) -- dep_on
>         deriving (Eq,Text)

```

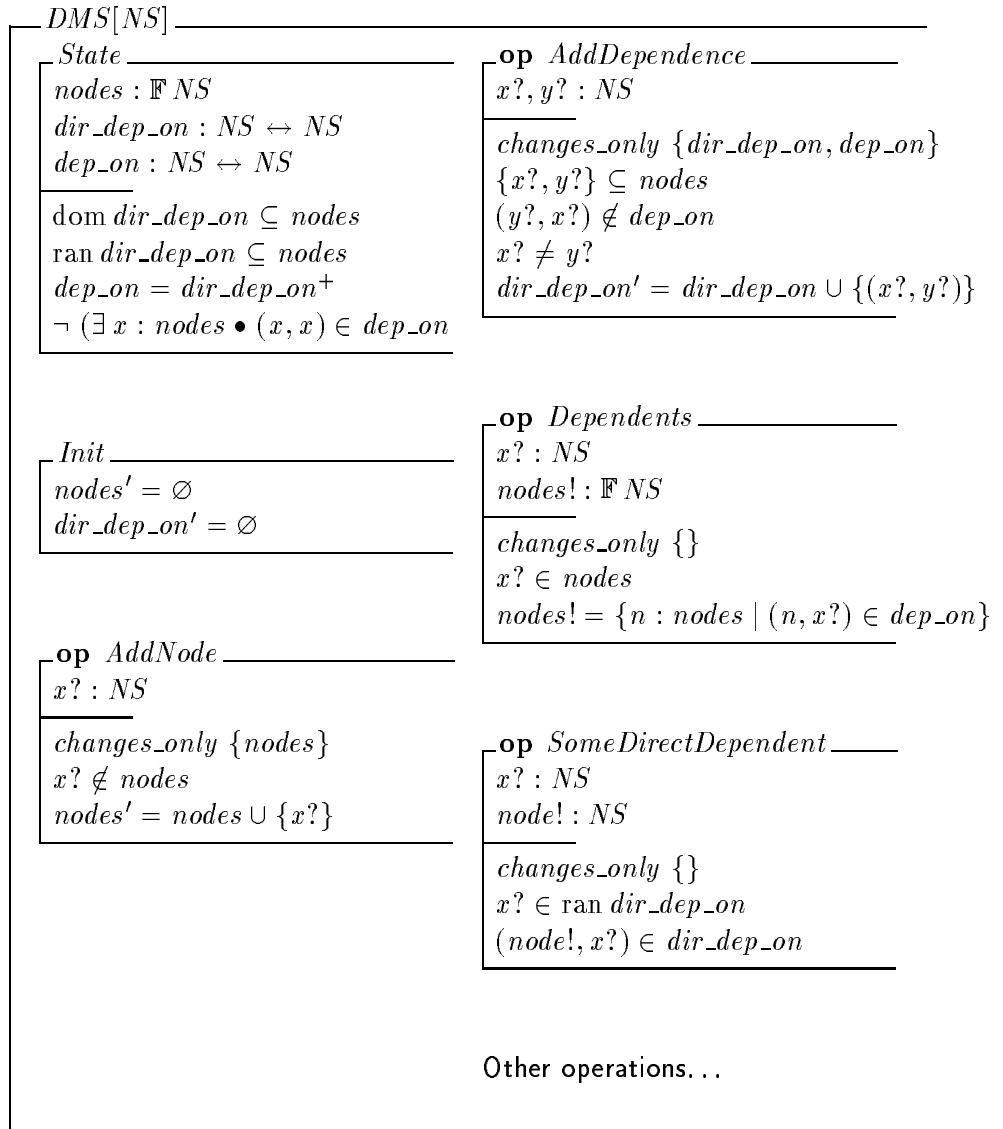


Figure 2: An SL1 specification of a DMS system

```

> dms_internal_inv :: Dms_state -> Bool
> dms_internal_inv (Dms nodes dir_dep_on dep_on)
>   = dom dir_dep_on 'subset_of' nodes &&
>     dom dep_on 'subset_of' nodes &&
>     not (or [(x,x) 'in' dep_on | x <- nodes])

> dms_inv :: Dms_state -> Bool
> dms_inv (Dms nodes dir_dep_on dep_on)
>   = dms_internal_inv (Dms nodes dir_dep_on dep_on) &&
>     dep_on = transclosure dir_dep_on

```

The *Init* operation translates to:

```

> dms_Init :: () -> Dms ns
> dms_Init ()
>   = Operation "dms_Init"
>     [state'
>       | dms_internal_inv state']
>   where
>     nodes' = set []
>     dir_dep_on' = set []
>     dep_on' = transclosure dir_dep_on'
>     state' = Dms nodes' dir_dep_on' dep_on'

```

Translating the two update (add) operations gives:

```

> dms_AddNode :: NS -> Dms -> Solutions Dms
> dms_AddNode x_in (Dms nodes dir_dep_on dep_on)
>   = Operation "dms_AddNode"
>     [state'
>       | x_in 'not_in' nodes;
>         dms_internal_inv state']
>   where
>     nodes' = nodes 'union' set [x_in]
>     dir_dep_on' = dir_dep_on
>     dep_on' = dep_on
>     state' = Dms nodes' dir_dep_on' dep_on'

> dms_AddDependence :: NS -> NS -> Dms -> Solutions Dms
> dms_AddDependence x_in y_in (Dms nodes dir_dep_on dep_on)
>   = Operation "dms_AddDependence"
>     [state'
>       | set [x_in,y_in] 'subset_of' nodes;
>         (y_in,x_in) 'not_in' dep_on;

```

```

>         x_in /= y_in;
>         dms_internal_inv state']
>     where
>         nodes' = nodes
>         dir_dep_on' = dir_dep_on 'union' set [x_in,y_in]
>         dep_on' = transclosure dir_dep_on'
>         state' = Dms nodes' dir_dep_on' dep_on'

```

The *Dependents* and *SomeDirectDependent* operations are query operations, since they do not change the state. However, their translation is similar to that of the update operations above, since we currently always output the final state, even if it is unchanged.

```

> dms_Dependents :: NS -> Dms -> Solutions (Dms, Set NS)
> dms_Dependents x_in (Dms nodes dir_dep_on dep_on)
>   = Operation "dms_Dependents"
>     [(state',nodes_out)
>      | x_in 'in' nodes;
>       dms_internal_inv state']
>   where
>     nodes' = nodes
>     dir_dep_on' = dir_dep_on
>     dep_on' = dep_on
>     nodes_out = set [ n | n <- nodes; (n,x_in) 'in' dep_on]
>     state' = Dms nodes' dir_dep_on' dep_on'

> dms_SomeDirectDependent :: NS -> Dms -> Solutions (Dms, NS)
> dms_SomeDirectDependent x_in (Dms nodes dir_dep_on dep_on)
>   = Operation "dms_SomeDirectDependent"
>     [(state',node_out)
>      | node_out <- NS;
>       x_in 'in' ran dir_dep_on;
>       (node_out,x_in) 'in' dir_dep_on;
>       dms_internal_inv state']
>   where
>     nodes' = nodes
>     dir_dep_on' = dir_dep_on
>     dep_on' = dep_on
>     state' = Dms nodes' dir_dep_on' dep_on'

```

The `dms_SomeDirectDependent` function is interesting because its list comprehension has `node_out` ranging over all integers, which means that some initial solutions will be produced (assuming that some exist), then it will search infinitely for more solutions. This could be fixed by realising that `node_out` must be in the domain of

`dir_dep_on` and generating `node_out <- dom dir_dep_on` instead. However, this is probably beyond what a naive translator would be capable of.

## 4 Conclusions

This paper has introduced three new measures for evaluating Z animation systems. The Haskell animation system described in the paper rates highly according to these three measures, since it has good interactivity (using the Haskell interpreter), has a reasonably transparent relationship between Z schemas and the resulting animation functions and it preserves operational equivalence. Returning to the original measures proposed by Breuer and Bowen [BB94], its coverage is average (it does not require schemas to be written using a special style, but it is restricted to finite types), its efficiency is average (usually better than a naive generate-and-test system, but non-obvious optimisations are left to the user) and its sophistication is average (it should terminate on all finite types). We do not have a proof of correctness, but believe that it should be easier to prove for this system (assuming that the Z library is implemented in Haskell correctly) than for more sophisticated systems which handle some infinite types.

The translations shown in this paper were performed by hand. In future work, we intend to extend our current Cogito-SL1 tools (parser, type checker, editor, theory generator and theorem prover) to include a translator to Haskell. We also intend to try extending this approach to handle schema references and schema operators directly, rather than expanding them out in advance. Finally, an alternative approach to animation, using the Cogito theorem prover, Ergo, also looks promising. It would seem to allow a mixture of evaluation using concrete data, symbolic evaluation and theorem proving.

## References

- [BB94] Peter T. Breuer and Jonathan P. Bowen. Towards correct executable semantics for Z. In J.P. Bowen and J.A. Hall, editors, *Z User Workshop, Cambridge 1994*, Workshops in Computing, pages 185–209. Springer-Verlag, 1994.
- [Dil90] Antoni Diller. *Z: An Introduction to Formal Methods*. John Wiley, 1990.
- [DKC89] A. J. J. Dick, P. J. Krause, and J. Cozens. Computer aided transformation of Z into Prolog. In Nicholls [Nic89], pages 71–85.
- [DN91] Veronika Doma and Robin Nicholl. EZ: A system for automatic prototyping of Z specifications. In S. Prehn and W. J. Toetenel, editors, *VDM '91: Formal Software Development Methods*, volume 551 of *LNCS*, pages 189–203. Springer-Verlag, 1991.

- [HJ89] I. J. Hayes and C. B. Jones. Specifications are not (necessarily) executable. *IEE/BCS Software Engineering Journal*, 4(6):330–338, November 1989.
- [HPe92] P. Hudak, S. Peyton Jones, and P. Wadler (editors). Report on the programming language Haskell, a non-strict purely functional language (version 1.2). *ACM SIGPLAN Notices*, 27(5), May 1992.
- [JS89] M. Johnson and P. Sanders. From Z specifications to functional implementations. In Nicholls [Nic89], pages 86–112.
- [Mor90] Carroll Morgan. *Programming from Specifications*. Prentice Hall, 1990.
- [Nic89] J.E. Nicholls, editor. *Z User Workshop, Oxford 1989*, Workshops in Computing, Berlin, Germany, 1989. Springer-Verlag.
- [Rob87] K.A. Robinson. Refining Z specifications to programs. In *Australian Software Engineering Conference*, pages 87–97. I. E. Aust, May 1987.
- [TKK<sup>+</sup>95] Owen Traynor, Peter Kearney, Ed Kazmierczak, Li Wang, and Einar Karlsen. Extending Z with modules. *Proceedings of Australian Computer Science Communications, Adelaide 1995*, 17(1):513–522, 1995.
- [Utt94] Mark Utting. A Haskell implementation of Z data types. Technical Report 94-39, Software Verification Research Centre, The University of Queensland, St. Lucia, QLD 4072, Australia, 1994. Available via ftp from `ftp.cs.uq.oz.au` in the file `/pub/SVRC/techreports/tr94-39.ps.Z`.