Rapid Prototyping using Formal Specifications

Michael Winikoff   Philip Dart   Ed. Kazmierczak

Department of Computer Science, The University of Melbourne.
{winikoff,philip,eka}@cs.mu.oz.au
http://www.cs.mu.oz.au/~{winikoff,philip,eka}

Abstract. There is growing interest in animating formal specifications for the purpose of better understanding the requirements and for validating the specification. Formal specifications in a non-executable language offer challenges for animation systems, for example, dealing effectively with infinite data sets, sensibly animating functions which are not computable and determining a sensible set of inputs and outputs for arbitrary relations.
In this paper we examine these issues in the context of animating Z specifications in the logic programming language Mercury. In particular we outline how information for making a non-executable Z specification executable can be derived using static analysis techniques from logic programming. We present analysis algorithms for deriving control (mode) and representation (subtype) information and show how these analyses are used in a tool for deriving Mercury programs from Z specifications. Finally we compare our approach with existing systems for animating Z specifications.
Keywords: Requirements, Rapid Prototyping, Formal Methods, Animation, Logic Programming, Z, Mercury.

1 Introduction

Many problems in software development can be traced back to poor requirements analysis [Sallis et al., 1995; Lutz, 1993; Curtis and Iscoe, 1988]. Furthermore, it is known that fixing requirements problems at later stages of the traditional software development life cycle is expensive [Davis, 1993; Boehm, 1981]. One way of alleviating the problems due to requirements is to shift more of the development effort to earlier phases of the software lifecycle. This is the motivation for rapid prototyping [Bischofberger and Pomberger, 1992; Luqi, 1992; Loucopoulos and Karakstas, 1995].

One other way of analysing requirements in more depth is to write a formal specification of the system, or at least the critical parts of it. Writing a formal specification allows us to model key aspects of the system clearly and unambiguously. There are a number of analyses that can be performed on the formal specification in order to increase our confidence that the model is complete, consistent and captures the key properties needed by the proposed system. One way of analysing formal specifications is to *prove* that the properties required of the system are also properties of the model. Another way is to create an *animation*, essentially a prototype, of the model and to evaluate the animation with the aid of the client.

If there is a mathematical correspondence between the animation and the formal specification then animating the formal specification can be a largely automatic process.

Ensuring such a correspondence has a number of advantages over current prototyping methods.

- There is significant scope for automation leading to shorter cycle times.
- At the end of the requirements analysis process a validated formal specification is available.
- Changes to the animation can easily be reflected in the specification.
- It is easier to determine if a defect is a defect of the animation or of the specification.
- There is less chance of the specification and the animation diverging as is often the case in prototyping [Hargrave, 1995].
- It is still possible to employ a theorem prover and check key properties and, provided that our mathematical correspondence preserves properties of the model in the animation, these properties should also hold of the animation.

Getting these advantages in a rapid prototyping process hinges on the ability to derive animations quickly and in such a way that preserves the properties of the original formal model.

In this paper we show how static analysis for logic programs can be used to derive information which is important for deriving animations from formal specifications. We do this in the context of deriving animations for Z specifications in the programming language Mercury.

We begin by discussing the requirements and issues (section 2). In the body of the paper (section 3) we present an animation method for the specification language Z. We discuss the transformation of Z to logic (section 3.1) and present analyses which the tool performs. In particular, we discuss control information represented as modes (section 3.2) and representation issues (section 3.3). In section 4 we outline a proof of correctness for the proposed animation method. We briefly compare our work to other animation methods which have been proposed for Z in section 5 and then conclude with the current status of our work and some future research.

2 Requirements and Issues for Animation

In this section we examine in detail the properties which are required of the process for deriving an animation from a specification. In order to realise the benefits of animation, the process has to satisfy a number of properties:

1. There must be a clean separation between the specification and any additional information required to animate a specification.
2. The derivation of the animation from the specification and the additional information must be as automated as possible.
3. It must be possible to animate specifications which contain non-executable parts.
4. Both the analysis and derivation of the animation, and its execution should be tolerably efficient.

Ideally we would like a clean separation of concerns between specification and animation. This allows the developer to concentrate on the task of understanding and analysing the problem while writing the specification, without being constrained by the need to guarantee executability. Consider, for example, the following specification:

$$\begin{array}{|l} \sqrt{\ } : \mathbb{R} \to \mathbb{R} \\ \hline \forall x \bullet (\sqrt{x})^2 = x \end{array}$$

To make the square root function executable we would need to choose a representation for the real numbers and an algorithm which computes approximations to the exact square root using the representation. As an example we might choose to represent real numbers by sequences of rational approximations which means choosing an algorithm to compute square roots using such sequences. Approximations like this do not add to the specification since they work counter to the principles of good abstraction by complicating our understanding of the problem rather than simplifying it. Further, approximations make reasoning about the specification unnecessarily difficult which in turn complicates the process of validation[1].

This example also indicates the two key issues for animating specifications: (i) determining a representation for abstract data and (ii) determining an operational semantics for the specification. Consider the following example:

$$\mathit{Divide} \mathrel{\widehat=} [x, y, q, r : \mathbb{N} \mid x = y \times q + r;\ r < y]$$

What operational semantics should we assign to the schema *Divide*? If we are given y, q and r then we can determine x. On the other hand, if we are given x, q and r then we can determine y or fail because y is not integral. In fact, we can determine any of the four variables, provided we have the other three. Thus we can execute the schema *Divide* in four different modes (at least!). Analysing the possible modes of use of predicates is one kind of analysis which assists the animation process but other analyses are also possible. The aim of such analyses is to automate as much of the animation process as possible while retaining the flexibility to choose efficient representations for data and algorithms. Further, the more that such a process is automated, the shorter the cycle time for deriving animations and evaluating them.

Another requirement that helps reduce cycle time is the ability to animate specifications which are not completely executable. In the absence of this property, the entire specification needs to be analysed and made executable (by providing additional information) before *any* testing or exploration can take place. Additionally, forcing the specification to be executable is likely to result in specifications which are less clear and which contain design biases [Hayes and Jones, 1989]. An animation tool should have a number of strategies for handling difficult parts of the specification – these can range from simply ignoring them (unsound!), executing them with a time bound, querying the user, or invoking another tool (such as a theorem prover).

Finally, both the analysis and animation process and the execution of the resulting animation ought to be reasonably efficient. It is particularly important that a generate and test execution strategy be avoided when possible.

[1] See [Bloesch and Kazmierczak, 1996] where reasoning about floating point approximations to real numbers is discussed.

We have mentioned a number of times the strong and precise relationship between the specification and the animation. The precise nature of this relationship is an open issue. One reason for this is that compromises in representation must sometimes be made. Specification notations (such as Z) use infinite data types. Representing (for example) the unbounded natural numbers in an animation might be possible but may sometimes not be desirable. Representing reals is generally difficult. These compromises blur the relationship between the specification and its animation.

A precise characterisation of the relationship between a specification and its animation is essential for the correctness proof. The notion of approximation has been proposed in [Breuer and Bowen, 1994; West, 1996], and although these sources do not address some of the issues mentioned here they do offer a good starting point.

3 An Animation Method for Z

In this section we present a concrete proof of concept – a simple (but not naive!) animation method for the specification language Z. The aim is twofold: to give a feel for what the animation process entails, and to demonstrate how the requirements of the previous section can be met.

As mentioned, we make use of the specification language Z[2] [Brien and Nicholls, 1992; Spivey, 1992]. We have chosen to use Z for a number of reasons: it is a popular notation and is being used in industry. Additionally, Z is quite rich and its animation requires addressing issues raised by constructs such as infinite data types and logically complex operations (schema operations). These constructs, respectively, offer challenges relating to the choice of data representation and to correctness arguments. It should be emphasised that such issues are not unique to Z – they are generic issues and their solutions are reusable. There are many alternatives to Z, however adopting Z as an exemplar specification language does not result in a significant loss of generality.

Our animation target is the logic programming language Mercury[3] [Somogyi et al., 1995b; Somogyi et al., 1995a]. A logic programming language is a very tempting target for a specification language which is based on first order set theory (such as Z). The conceptual gap between a logic programming language (which is effectively, an executable subset of first order logic) and a logic based specification language is significantly smaller than between, say, a logic based specification language and an imperative programming language. This reduces the amount of transformation involved and significantly simplifies the correctness arguments.

The primary reason for choosing Mercury is that (unlike Prolog) it is sound with respect to the logical semantics – any answer returned by the Mercury system is a logical consequence of the program[4]. Other reasons include its efficiency[5] and its use of extra-

[2] See also the Z Archive at `http://www.comlab.ox.ac.uk/archive/z.html`

[3] For more information on Mercury see `http://www.cs.mu.oz.au/mercury`

[4] On the other hand, there are a number of aspects where Prolog fails to satisfy this. These include unsafe negation, the lack of an occur check and a range of impure features.

[5] Mercury is twice the speed of the fastest Prolog system, and is around five to ten times the speed of commercial compiler-based Prolog systems [Somogyi et al., 1996].

logical annotations[6] to specify control information.

One other major reason for selecting Mercury is that it fits in with an important philosophical point which we adopt which is *early checking*. Mercury makes extensive use of static analyses to detect errors at compile time. This notion of *early checking* is important for animation for the following reasons. In order to be able to apply a range of strategies to difficult (non-executable) parts of the specification we need to identify the difficult parts before runtime. Additionally, even if only a single naive strategy is used, at least there is a warning that the animation is likely to run into difficulties. On the other hand, if checking is not performed at compile time then specifications containing difficult parts are more likely to be translated into nonterminating programs – this introduces a debugging aspect into the animation process and significantly lengthens the cycle time.

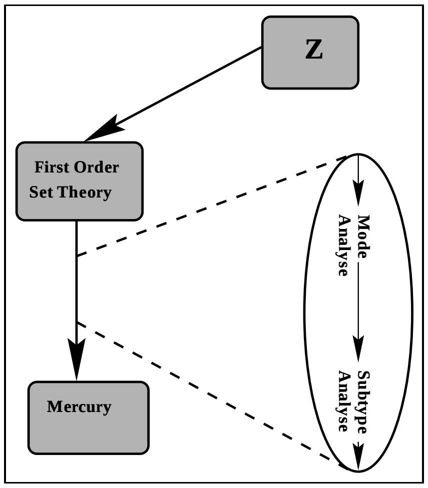

Fig. 1. The Animation Process

Early checking determines our view of executability – we want to impose as strong a restriction as possible so as to guarantee that problems with executing the animation are detected as early as possible. Hence a specification is considered to be executable if it is complete with respect to a sequential execution strategy.

We now need to look at a number of questions which determine the animation process. We have chosen our starting point (Z) and end point (Mercury). We now need to build a bridge between them. In order to do this we need to decide:

1. What additional information do we need to translate Z into Mercury?
2. How can we determine this information automatically?
3. Can we, and how do we, combine the extra information and the Z specification to yield a Mercury program?
4. What can we do with the difficult parts of the specification?

We can answer the first question by simply looking at the differences between logic and Mercury – the main missing ingredient is control information as embodied in modes. In section 3.2 we present a mode analysis algorithm.

In addition to translating the specification we also need to provide an equivalent to Z's standard library and to provide intelligent representations for the Z datatypes.

The subtype analysis algorithm outputs a modified logical formula which incorporates the derived information. This includes the mode information derived in an earlier

[6] Mercury supports mode and determinism declarations, and various pragmas[Henderson et al., 1996].

pass (see figure 1) and, with the addition of a suitable library and a change of syntax, yields a Mercury program.

For the moment, we leave open the question of dealing with the difficult parts of a specification – flexibility is the best approach and as long as the analyses are able to detect difficulties, a range of approaches can be used.

The proposed animation process is summarised in figure 1. We begin by translating Z to first order set theory and then analyse the result in order to determine control information and representation information.

3.1 Z To Logic

The first step in the animation of Z is translation to first order set theory. This is done for a number of reasons: it reduces the size (and complexity) of the language and it means that the analysis parts of the system are not Z specific – this makes it easier to later extend the system to handle other specification notations. We shall also see in section 4 how using first order set theory as an intermediate notation simplifies the correctness proof.

The main features which are present in Z but absent from first order set theory are schema operations, various forms of paragraphs (generic definitions, axiomatic definitions, etc.) and expressions. The translation takes as input a specification written in standard Z [Brien and Nicholls, 1992] and outputs a transliteration in first order set theory where the syntax is:

$$\mathcal{L} \ ::= \ \mathit{atom} \mid \mathcal{L} \wedge \mathcal{L} \mid \mathcal{L} \vee \mathcal{L} \mid \neg\, \mathcal{L} \mid \mathcal{L} \Leftrightarrow \mathcal{L} \mid \exists\, D \bullet \mathcal{L} \mid \mathit{if}\ \mathcal{L}\ \mathit{then}\ \mathcal{L}\ \mathit{else}\ \mathcal{L} \mid \{D \mid \mathcal{L}\}$$

Flattening expressions into predicates is straightforward as is the expansion of schema operations. We focus below on the interesting part of the translation – handling the various forms of Z paragraphs. The details of the translation to first order set theory can be found in [Winikoff and Dart, 1997]. In particular the presentation below omits certain details relating to the handling of generics and global names.

One form of a Z paragraph is the axiomatic definition:

$$\begin{array}{|l} x : T \\ \hline p \end{array}$$

This defines an x in the set T which satisfies p. It also asserts that at least one such x exists. The translation to first order set theory separates these two aspects. We generate the clause $\forall\, x \bullet \mathit{axiom}(x) \Leftrightarrow x \in T \wedge p$. We also modify goals by adding a *consistency check* which ensures that for each axiomatic definition there is at least one value which satisfies the axiom, ie. $\exists\, x \bullet \mathit{axiom}(x)$.

Another form of a Z paragraph is the schema: $S \mathrel{\widehat{=}} [x : T;\ y : T' \mid q]$. This defines a set of bindings, each of which maps the names x and y to values drawn from T and T' respectively which satisfy q. Unlike axiomatic definitions there is no satisfiability requirement – the set of possible bindings may be empty.

The translation of schema requires a notion of binding. For each set of names and types $n_1 : T_1, \ldots, n_n : T_n$ we define the predicates $\mathit{bind}(v_1, \ldots, v_n, b)$ which is true iff its last argument is a binding where the *i*th name (n_i) is mapped to v_i. Given this, the schema above can be translated to: $\forall\, b \bullet S(b) \Leftrightarrow \exists\, x, y \bullet x \in T \wedge y \in T' \wedge q \wedge bind(x, y, b)$.

A third type of Z paragraph is the generic definition. The generic definition

$$\begin{array}{|l}
[X] \\
\hline
C : \mathbb{P}\, X \\
\hline
\forall\, x : X \bullet x \in C \Leftrightarrow p \\
\hline
\end{array}$$

defines a family of sets. For each set Y we have $C[Y]$ which is the definition given where X is replaced with Y. Generic definitions are used extensively in the Z tool kit to provide polymorphism.

The translation of generic definitions combines the instantiation step (providing the Y above) and the actual use into a single step. This gives the following translation: $\forall\, X, C \bullet gen(X, C) \Leftrightarrow (\forall\, x \bullet x \in C \Leftrightarrow (x \in X \wedge p))$ We actually prefer to avoid the indirection involved and generate the simpler version: $\forall\, X, x \bullet C(X, x) \Leftrightarrow x \in X \wedge p$. Note that here C is the name of the predicate being defined, not a variable.

In order to be well defined a generic definition must be *uniquely* defined for any instantiation of the generic parameter [Spivey, 1992, page 40]. We could insert a consistency check for this; however proving the uniqueness of a definition will (usually) be beyond the abilities of the animation tool and so we require that these are proven separately – perhaps using an interactive theorem prover.

We conclude this section with a simple example which we shall return to in the next section. The following Z specification

$$\begin{array}{|l}
[X] \\
\hline
\_ \text{ subseq } \_ : \operatorname{seq} X \leftrightarrow \operatorname{seq} X \\
\hline
\forall\, x, y : \operatorname{seq} X \bullet x \text{ subseq } y \Leftrightarrow (\exists\, t_1, t_2 : \operatorname{seq} X \bullet x = t_1 \frown y \frown t_2) \\
\hline
\end{array}$$

is translated to the formula:

$$\begin{array}{l}
\forall\, X, x, y \bullet \ \text{subseq}(X, x, y) \Leftrightarrow \exists\, t_1, t_2, t_3 \bullet \\
\qquad t_1 \in \operatorname{seq} X \wedge t_2 \in \operatorname{seq} X \wedge t_3 \in \operatorname{seq} X \wedge cat(X, t_1, t_3, x) \wedge cat(X, y, t_2, t_3)
\end{array}$$

Where *cat* is the translation of Z's concatenation operator over sequences ($\frown$) and where x and y are required to be sequences of X. We could insert this check into the formula, however we prefer not to – as we shall see in section 3.3 it may be possible to infer that if *subseq* is used in certain contexts then x and y are guaranteed to be sequences. When this cannot be proven, the necessary type check can be inserted in the calling context.

3.2 Control: Modes

One of the main types of information which we need to add to a Z specification in order to execute it is control information. The control information specifies how to execute a formula – effectively, it defines a flow of control. In Mercury control information is derived from modes. Modes specify for each predicate the required variable state before execution and the state of variables resulting from execution. For example, the append predicate might require that (i) its first two arguments be known before execution, and (ii) the result of execution be to determine the third argument. Modes provide an abstract description of the requirements for and the effect of executing a formula.

We shall be using a fairly simple mode system which treats variables as being either *unbound*, or completely determined (*ground*). A predicate's mode is written as a mapping from a set of variables which are required to be ground, to a set of variables which are required to be unbound and which are made ground by the execution of the predicate. For example, two of the modes of the equality predicate are:

$$X = Y :: \{X\} \Rightarrow \{Y\} \qquad X = Y :: \{Y\} \Rightarrow \{X\}$$

The first of these corresponds to assigning to the unbound variable Y the value of the ground variable X. The example mode given above for append would be written as $append(X, Y, Z) :: \{X, Y\} \Rightarrow \{Z\}$. Since a variable cannot be both unbound and ground it follows that for any mode of the form $I \Rightarrow O$ we have that $I \cap O = \emptyset$. In addition, the mode of an atomic formula A is given and is denoted by m_A.

Given the modes of atomic formulae, it is possible to determine the modes of compound formulae as follows:

$\vee$: Since the subformulae need to be interchangeable (from an operational point of view) they must have the same modes. Thus, a mode holds for a disjunction if it holds for *every* disjunct.

$\neg$: Negations cannot bind variables so any mode for F which is of the form $I \Rightarrow \emptyset$ is also a mode for $\neg F$. Note that if F requires that certain variables are known then $\neg F$ will also require that these variables be known.

$\exists$: The variables being quantified over begin life as unbound and hence cannot be required as input. Since they are local, we delete them from the result. Thus if F has mode $I \mapsto O$ then the mode of $\exists x\, F$ is $I \mapsto (O \setminus \{x\})$ provided that x does not occur in I.

$\wedge$: The execution of a conjunction corresponds to sequencing. Note that at this stage we assume that the sequence of formulae is given, that is, the conjunction $F_1 \wedge F_2 \wedge F_3$ is well moded if and only if F_1 can be executed first, followed by F_2 and then F_3. We discuss the reordering of conjuncts below.
A given sequence of subformulae can be moded as follows. Suppose that we are dealing with a binary conjunction $F_1 \wedge F_2$ and that F_1 and F_2 have modes $I_1 \Rightarrow O_1$ and $I_2 \Rightarrow O_2$ respectively. The execution of the conjunction begins by executing F_1, hence I_1 must be ground. The execution grounds the variables O_1. Execution continues with F_2. Note that the variables in I_2 must be ground – hence they must either be given as ground *or* be in O_1. The execution of F_2 grounds the variables in O_2. The variables needed for execution of $F_1 \wedge F_2$ are I_1 and $I_2 \setminus O_1$. The variables which are made ground are $O_1 \cup O_2$. This gives the overall mode $F_1 \wedge F_2 :: (I_1 \cup (I_2 \setminus O_1)) \Rightarrow (O_1 \cup O_2)$.

Rules for determining whether a formula is well moded are given in figure 2. One rule which we haven't discussed yet is weakening. This allows a formula to accept more known variables than it really needs. It is used to obtain agreement between disjuncts. For example, if $F_1 :: \{X\} \Rightarrow \{Z\}$ and $F_2 :: \{X, Y\} \Rightarrow \{Z\}$ then the formula $F_1 \vee F_2$ is well moded if F_1 can also be given the mode $\{X, Y\} \Rightarrow \{Z\}$.

Note that we do *not* have a corresponding rule on output. A rule which would allow formulae to pretend that they bind fewer variables than they really do would be problematic since it would allow a variable's value to be generated by more than one predicate. This poses a problem since it introduces an implied equality check (and hence a notion of implied modes) and since we allow infinite data which cannot be tested for equality.

$$\frac{F_1 :: I \Rightarrow O \quad \ldots \quad F_n :: I \Rightarrow O}{F_1 \vee \ldots \vee F_n :: I \Rightarrow O}\ \vee \qquad \frac{}{A :: m_A}$$

$$\frac{F_1 :: I_1 \Rightarrow O_1 \quad \ldots \quad F_n :: I_n \Rightarrow O_n \quad \forall j : 2..n \bullet O_j \cap (O_1 \cup \ldots \cup O_{j-1}) = \emptyset}{F_1 \wedge \ldots \wedge F_n :: J_1 \cup \ldots \cup J_n \Rightarrow O_1 \cup \ldots \cup O_n}\ \wedge$$

Where $J_1 = I_1$ and $J_{i+1} = I_{i+1} \setminus (O_1 \cup \ldots \cup O_i)$.

$$\frac{F :: I \Rightarrow O \quad O = \emptyset}{\neg F :: I \Rightarrow \emptyset}\ \neg \qquad \frac{F :: I \Rightarrow O \quad I \cap \overline{x} = \emptyset}{\exists \overline{x} \bullet F :: I \Rightarrow (O \setminus \overline{x})}\ \exists$$

Where $\overline{x}$ is a set of variables.

$$\frac{F :: I \Rightarrow O \quad O = \overline{y} \quad I \cap \overline{y} = \emptyset}{x = \{\overline{y} \mid F\} :: I \Rightarrow x}\ \mathit{set} \qquad \frac{F :: I \Rightarrow O \quad I \subseteq I'}{F :: I' \Rightarrow O}\ \mathit{weaken}$$

Fig. 2. Rules for Mode Analysis

One important point regarding the derivation of modes concerns the rule for conjunction. In the form presented, the rule assumes that the conjunction $F_1 \wedge \ldots \wedge F_n$ is executed in order. This is highly undesirable since it implies that logical conjunction is not commutative – since the logic is being generated automatically from a specification in a notation where logical conjunction *is* assumed to be commutative this is a problem.

Thus, we need to allow for different orderings in conjunction. A conjunction of formulae $F_1 \wedge \ldots \wedge F_n$ is well moded if some permutation of the formulae is well moded according to the definition above. Note that in some cases, a conjunction will have multiple modes which involve different orderings. For example, the formula $\exists z \bullet x = z \wedge z = y$ has the two modes $\{x\} \Rightarrow \{y\}$ and $\{y\} \Rightarrow \{x\}$. The first involves the moding:

$$(\exists z \bullet (x = z :: \{x\} \Rightarrow \{z\} \wedge z = y :: \{z\} \Rightarrow \{y\}) :: \{x\} \Rightarrow \{z, y\}) :: \{x\} \Rightarrow \{y\}$$

The second involves swapping the order of the two conjuncts:

$$(\exists z \bullet (z = y :: \{y\} \Rightarrow \{z\} \wedge x = z :: \{z\} \Rightarrow \{x\}) :: \{y\} \Rightarrow \{z, x\}) :: \{y\} \Rightarrow \{x\}$$

The reordering is part of the mode analysis and is performed at analysis time. This contrasts with a common notion in logic programming languages of coroutining where goals are reordered at runtime using a delay mechanism. Performing the check at analysis time fits in with the principle of early checking.

```
1  $\forall X, x, y \bullet$
2  $\text{subseq}(X, x, y) :: \{X, x\} \Rightarrow \{y\}$
3      $\Leftrightarrow \exists t_1, t_2, t_3 \bullet ($
4          $(cat(X, t_1, t_3, x) :: \{X, x\} \Rightarrow \{t_1, t_3\}$
5           $\wedge\ t_1 \in \text{seq}\, X :: \{t_1\} \Rightarrow \emptyset$
6           $\wedge\ t_3 \in \text{seq}\, X :: \{t_3\} \Rightarrow \emptyset$
7           $\wedge\ cat(X, y, t_2, t_3) :: \{X, t_3\} \Rightarrow \{y, t_2\}$
8           $\wedge\ t_2 \in \text{seq}\, X :: \{t_2\} \Rightarrow \emptyset$
9          $) :: \{X, x\} \Rightarrow \{t_1, t_2, t_3, y\}$
10     $) :: \{X, x\} \Rightarrow \{y\}$
```

Fig. 3. Mode Analysis Example

There are a number of pleasant consequences of the mode analysis we have chosen. The execution of a mode-correct formula cannot violate the occur check (since equality is only well moded as an assignment), it cannot bind variables in the scope of a negation and we are guaranteed that a simple sequential execution is sound.

The mode analysis is implemented as a nondeterministic bottom up derivation. Modes are assigned to atoms and then combined upwards to give the moding for the compound formula. The algorithm uses a topological sort to avoid generating all permutations of conjunctions.

Returning to our running example, the definition of *subseq* can be analysed and the mode $\{X, x\} \Rightarrow \{y\}$ determined. The derivation is given in figure 3.

3.3 Subtypes and Representation

Z's notion of type is fairly coarse. In order to animate a specification more information is required. For example, in Z, a function and relation have the same (basic) type. However, function application is invalid with relations. A similar situation applies for sequences (which are just instances of $\mathbb{P}(\mathbb{Z} \times X)$).

Also, when considering whether operations can be animated, some operations can be performed in certain modes with infinite sets and in other modes only with finite sets. A simple example is equality – we can assign an infinite object to a variable, however testing for equality between two ground values can only be performed if the values are finite.

Thus subtype analysis is needed. Furthermore, it is desirable to perform the analysis at compile time since a runtime check may be expensive, or in some cases, semi-decidable or even undecidable.

The subtype analysis inserts all of the tests which are required to be in the code and then attempts to eliminate some subset of these by proving that they can never fail. This is similar to soft typing and to the work of [Dart and Zobel, 1992].

We begin with a brief motivating example. Consider the logical formula $\exists x, r \bullet x = y \frown z \wedge x = q \frown r$. Suppose that the formula has been given mode $\{y, z\} \Rightarrow \{q\}$

and that the first conjunct has mode $\{y,z\} \Rightarrow \{x\}$ and the second conjunct has mode $\{x\} \Rightarrow \{q,r\}$.

The library operation $\frown$ ("cat", which concatenates sequences) is only applicable if its arguments are sequences (that is, partial functions from $\mathbb{N}$ to X with a range of $1 \mathrel{..} n$ for some n). Thus, the first conjunct above is *really*[7]: $\mathrm{seq}\, y \wedge \mathrm{seq}\, z \wedge x = y \frown z \wedge \mathrm{seq}\, x$. Note that we place constraints on input variables before the predicate and constraints on the output after the predicate. This makes no difference logically (as conjunction is commutative) but retains the ability of $\wedge$ to be treated as sequential conjunction. Thus, the whole formula is treated as[8] $\exists\, x, r \bullet (\mathrm{seq}\, y \wedge \mathrm{seq}\, z \wedge x = y \frown z \wedge \mathrm{seq}\, x \wedge \mathrm{seq}\, x \wedge x = q \frown r \wedge \mathrm{seq}\, r \wedge \mathrm{seq}\, q)$

One useful property of $\frown$ is that for $x \frown y = z$ the constraint *seq z* $\Leftrightarrow$ *seq y* $\wedge$ *seq x* holds. This constraint can be assumed to hold *after a successful execution of* $x \frown y = z$. Thus, we cannot eliminate tests on inputs. However, if (for example) $\frown$ is used in the mode where x and y are determined from z and z is known to be a sequence (either from the context or since we have an explicit check) then after the call to $\frown$ we can conclude that x and y are guaranteed to be sequences.

We can apply this reasoning process to the first occurence of $\frown$ in the formula above and conclude that x must be a sequence. Since x is a sequence, the second occurence of $\frown$ (in mode $\{x\} \Rightarrow \{q,r\}$) does not require any checks on its input and guarantees that q and r are sequences. This yields the final formula: $\exists\, x, r \bullet (\mathrm{seq}\, y \wedge \mathrm{seq}\, z \wedge x = y \frown z \wedge x = q \frown r)$.

Note that modes are needed in order to perform the subtype analysis. The analysis may need to modify the input formula by inserting checks which could not be eliminated. Inserting these checks preserves sequential executability – if the input can be sequentially executed then so can the output.

The subtype analysis returns a four tuple consisting of (1) the required properties which need to be inserted as tests[9], (2) the constraints which hold, (3) the number of solutions of the predicate[10], and (4) the modified formula.

The basic properties which we are interested in are *subtype* (e.g. partial function, sequence, bag etc.) and *size* (e.g. finite, infinite, empty). As we have seen, subtype information is important to determining the semantics of certain Z operations. Size information is useful is selecting representations and in determining when implied modes are permitted. Due to a lack of space we are unable to present further details of the subtype analysis. We refer the interested reader to [Winikoff, 1997].

---

[7] Where we write $\mathrm{seq}\, x$ to indicate the goal that succeeds iff x is a sequence, ignoring for the moment the type of the elements of the sequence.

[8] Yes, there are two occurences of *seq x* – the first is the condition on x occuring as the output of the first $\frown$. The second is the condition on x occuring as the input to the second occurence of $\frown$.

[9] We keep these separate so that they can be eliminated if possible.

[10] This is needed since the size of a set comprehension depends on the number of solutions of its predicate.

$$\boxed{\begin{array}{l}
\forall X, x, y \bullet \\
\mathrm{subseq}(X,x,y) :: \{X,x\} \Rightarrow \{y\} \\
\quad \Leftrightarrow \exists\, t_1, t_2, t_3 \bullet (\\
\qquad\quad (x \in \mathrm{seq}\, X :: \{x\} \Rightarrow \emptyset \\
\qquad\quad \wedge\ \mathit{cat}(X, t_1, t_3, x) :: \{X,x\} \Rightarrow \{t_1, t_3\} \\
\qquad\quad \wedge\ \mathit{cat}(X, y, t_2, t_3) :: \{X, t_3\} \Rightarrow \{y, t_2\} \\
\qquad\quad) :: \{X,x\} \Rightarrow \{t_1, t_2, t_3, y\} \\
\quad) :: \{X,x\} \Rightarrow \{y\}
\end{array}}$$

Fig. 4. Subtype Analysis Example

Returning to our running example, we consider the first atomic formula (line 4 of figure 3): Since x is an input and is required to be a sequence (from the type declaration) we insert a check of this yielding: $x \in \mathrm{seq}\, X \wedge \mathit{cat}(X, t_1, t_3, x)$. We now consider lines 5 and 6. From the fact that x is a sequence and from properties of *cat* we can deduce that t_1 and t_2 are sequences of X. Hence these two lines can be eliminated since they will never fail.

We now turn to line 7 of figure 3. The input variable t_3 is required by *cat* to be a sequence of X. From the analysis of the previous formula we know that this is the case and no check needs to be inserted. Since t_3 is a sequence of X, the properties of *cat* allow us to conclude that y and t_2 are both sequences of X. This allows us to delete line 8 leaving us with the definition in figure 4. Given an appropriate definition for *cat* this can be trivially translated to the Mercury code:

```
:- pred subseq(X::in, seq(X)::in, seq(X)::out) is nondet.
subseq(_X,_x,_y) :-
  is_seq(_X,_x), cat(_X,T_1,T_3,_x), cat(_X,_y,T_2,T_3).
```

4 Correctness

In this section we outline a proof of correctness for the animation method. Until the Z standard emerges and Mercury is provided with a detailed formal semantics the proof cannot be completed in full detail. However, we can still derive a strategy for the proof.

The result we are interested in showing is a soundness result – any fact that is derived by the animation should be a consequence of the interpretation of the original specification under Z's semantics. As an aside, we note that a majority of the existing implemented systems (see section 5) are not sound.

The proof divides naturally into the following parts:

1. **Correctness of the Library:** Where the library code is not generated by the animation tool we need to prove that it is faithful to the Z toolkit. For each library operation we prove that it approximates (in the sense of [Breuer and Bowen, 1994]) the corresponding Z operation.
2. **Correctness of the Analyses:** Both analyses can be viewed as abstract executions. In order to prove them correct we need to provide a formalisation of sequential execution. We then use standard techniques to show that the domains used (modes and constraints etc.) are sound approximations.

3. **Equivalence between full Z and the logic translation:** This requires showing that the translation from Z to first order set theory preserves meaning. We can view the translation as being from full Z to a kernel subset. The translation should correspond to meaning preserving transformations of Z according to the semantics given in [Brien and Nicholls, 1992].
4. **Soundness of Mercury with respect to first order logic:** In translating from Z to Mercury we model the logical connectives of first order set theory using those of Mercury – for example $\vee$ is translated to "`;`". We need to show that the Mercury implementation of the logical connectives is sound with respect to first order set theory. Mercury has been carefully designed so as to be sound, however the Mercury group has not yet provided a detailed formal semantics for the language or a formal proof of soundness.

5 Comparison

Due to space limitations we are unable to present other proposals in details and our comparison will be rather cursory. We are aware of ten other proposals (see figure 5) for animation systems. Of these, only half have been implemented. Only one of the proposals pays any attention to the process surrounding the animation – the others treat the technical aspects of the tool only. Most of the proposals translate Z into either Prolog or Haskell and do not use any form of analysis. The properties of the various proposals are summarised in figure 5. We consider a system to be *implemented* if it can take a Z specification and produce a running animation. Some proposals (zx and Z-into-Haskell) have an implemented toolkit but require that the user manually translate the Z specification into another notation. We do not consider these to be implemented. A tool is *sound* if all answers it gives satisfy the specification. A surprising number of the tools are unsound.

System & References	**Implemented?**	**Process Issues?**	**Analysis**	**Sound?**
EZ [Doma and Nicholl, 1991]	✓	✗	✗	✗
PiZA [Hewitt et al., 1997]	✓	✗	✓	✗
West [West, 1995]	✗	✗	✗	✓
Z-into-Haskell [Goodman, 1995]	✗	✗	✗	✓
zx [Breuer and Bowen, 1994]	✗	✗	✗	✓
Utting [Utting, 1994]	✗	✗	✗	✓
ZANS [Jia, 1995]	✓	✗	✓	✓
FunZ [Sherrell and Carver, 1995]	✗	✓	✗	?
Tzc [Sterling et al., 1996]	✓	✗	✗	✗
Z$--$ [Valentine, 1995]	✓	✗	✗	?

Fig. 5. Other Animation Proposals

The animation method presented in this paper improves on previous work in a number of ways:

- It pays attention to process related issues – ie. how is the animation tool used? In particular, the importance of iterative development is recognised.
- It is sound. In particular, it does not ignore difficult parts of the Z notation (for example, generic definitions, non-base types, axiomatic definition).
- It uses analysis to guarantee that a specification can be sensibly executed. In particular, we emphasize the notion of early checking.
- The mode analysis improves on the limited notion of directionality of [Jia, 1995] and on the informal development in [Hewitt, 1991; Hewitt, 1992].
- The use of a subtype analysis in animation is novel.

6 Status & Further Work

The translation from Z to Logic [Winikoff and Dart, 1997] has been implemented as an extension to the Melbourne University Z typechecker, MUZ, which is itself implemented in Mercury. The mode and subtype analyses described have also been implemented and a number of simple specifications have been successfully analysed and translated to Mercury and Prolog.

In addition to completing the development of the animation tool as proposed, there are a number of areas for further work. One of the advantages of a formal specification over a prototype in a scripting language is that it is possible to do more with a formal specification then just execute it. Theorem provers can be used to show that a formal specification has certain properties, model checkers can guarantee that certain states will not arise. We would like to eventually build an integrated workbench containing a range of tools. In particular, there are a number of places where integration with a theorem prover would prove useful – for example proving correctness of alternative (executable) versions of a specification or attempting to animate difficult parts of a specification.

Once the tool is fully operational, a detailed systematic evaluation and comparison to existing work can be performed. Finally, there is scope for improving the tool – for example, the mode analysis used is simple. By adopting a more sophisticated mode system it is possible to extend the class of specifications which can be automatically animated.

Acknowledgements

We would like to thank Peter Breuer, Mark Hewitt, Leon Sterling, Margaret West, Mark Utting and Howard S. Goodman for discussion relating to the capabilities of their systems.

References

Bischofberger, W. and Pomberger, G. (1992). *Prototyping-Oriented Software Development: Concepts and Tools*. Texts and Monographs in Computer Science. Springer-Verlag.

Bloesch, A. and Kazmierczak, E. (1996). Formally developing the square root function: A case study in specifying and refining operations involving real numbers. In Ramamohanarao, Kotagiri, editor, *Australasian Computer Science Conference*, pages 35–44, Melbourne.

Boehm, B. (1981). *Software Engineering Economics*. Prentice Hall.

Bowen, J. P. and Hinchey, M. G., editors (1995). *ZUM'95: The Z Formal Specification Notation, 9th International Conference of Z Users, Limerick, Ireland, September 7-9, 1995, Proceedings*, volume 967 of *Lecture Notes in Computer Science*. Springer-Verlag.

Breuer, P. T. and Bowen, J. P. (1994). Towards correct executable semantics for Z. In Bowen, J. P. and Hall, J. A., editors, *Z User Workshop, Cambridge 1994*, Workshops in Computing, pages 185–209. Springer-Verlag.

Brien, S. M. and Nicholls, J. E. (1992). Z base standard. Technical Monograph PRG-107, Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford, UK. Accepted for standardization under ISO/IEC JTC1/SC22.

Curtis, B., Krasner H. and Iscoe, Neil (1988). A field study of the software design process for large systems. *Communications of The ACM*, 31(11):1268–1287.

Dart, Philip W. and Zobel, Justin (1992). Efficient run-time checking of typed logic programs. *Journal of Logic Programming*, 14(1&2):31–69.

Davis, Alan M. (1993). *Software Requirements: Objects, Functions and States*. Prentice Hall. Revised Edition.

Doma, V. and Nicholl, R. (1991). EZ: A system for automatic prototyping of Z specifications. In Prehn, S. and Toetenel, W. J., editors, *VDM'91: Formal Software Development Methods*, volume 551 of *Lecture Notes in Computer Science*, pages 189–203. Springer-Verlag.

Goodman, H. S. (1995). The Z-into-Haskell tool-kit: An illustrative case study. In [Bowen and Hinchey, 1995], pages 374–388.

Hargrave, B. (1995). When to prototype: Decision variables used in industry. *Information and Software Technology*, 37(2):113–118.

Hayes, I. J. and Jones, C. B. (1989). Specifications are not (necessarily) executable. *IEE/BCS Software Engineering Journal*, 4(6):330–338.

Henderson, Fergus James, Conway, Thomas Charles, Somogyi, Zoltan, and Jeffery, David (1996). The Mercury language reference manual, version 0.5. Technical Report 96/10, Department of Computer Science, University of Melbourne.

Hewitt, M. A. (1991). Automated animation of Z using Prolog. B.Sc. Project Report, Lancaster University, Department of Computing.

Hewitt, M. A. (1992). Optimization of Prolog generated from Z specifications. Master's thesis, Department of Computing Science, University of Aberdeen.

Hewitt, M A, O'Halloran, C M, and Sennett, C T (1997). Experiences with PiZA, an animator for Z. In Bowen, Jonathan P., Hinchey, Michael G., and Till, David, editors, *ZUM'97: TheZ Formal Specification Notation*, pages 37–51. Springer. LNCS 1212.

Jia, Xiaoping (1995). An approach to animating Z specifications. Available from `ise.cs.depaul.edu` with the `ZTC` and `ZANS` tools.

Loucopoulos, P. and Karakstas, V. (1995). *System Requirements Engineering*. Software Engineering. McGraw-Hill.

Luqi (1992). Status report: Computer-aided prototyping. *IEEE Software*, 9(6):77–81.

Lutz, Robyn R. (1993). Targeting safet-related errors during software requirements analysis. In *Proceedings of SIGSOFT '93, Foundations of Software Engineering*, pages 99–105.

Sallis, Philip, Tate, Graham, and McDonell, Stephen (1995). *Software Engineering*. Addison Wesley Publishing Co.

Sherrell, Linda B. and Carver, Doris L. (1995). FunZ: An Intermediate Specification Language. *The Computer Journal*, 38(3):193–206.

Somogyi, Zoltan, Henderson, Fergus, and Conway, Thomas (1995a). Mercury: an efficient purely declarative logic programming language. In *Proceedings of the Australian Computer Science Conference*, pages 499–512.

Somogyi, Zoltan, Henderson, Fergus, Conway, Thomas, Bromage, Andrew, Dowd, Tyson, Jeffery, David, Ross, Peter, Schachte, Peter, and Taylor, Simon (1996). Status of the Mercury system. In *Proceedings of the JICSLP'96 Workshop on Parallelism and Implementation Technology for (Constraint) Logic Programming Languages*, pages 207–218.

Somogyi, Zoltan, Henderson, Fergus, Conway, Thomas, and O'Keefe, Richard (1995b). Logic programming for the real world. In Smith, Donald A., editor, *Proceedings of the ILPS'95 Postconference Workshop on Visions for the Future of Logic Programming*, pages 83–94, Portland, Oregon.

Spivey, J. M. (1992). *The Z Notation: A Reference Manual*. Prentice Hall International Series in Computer Science, 2nd edition.

Sterling, Leon, Ciancarini, Paolo, and Turnidge, Todd (1996). On the animation of "not executable" specifications by Prolog. *International Journal of Software Engineering and Knowledge Engineering*, 6(1):63–87.

Utting, Mark (1994). Animating Z: Interactivity, transparency and equivalence. Technical Report 94-40, Software Verification Research Center.

Valentine, Samuel H (1995). The programming language $Z--$. *Information and Software Technlogy*, 37(5-6):293–301.

West, M. M. (1995). Types and sets in Gödel and Z. In [Bowen and Hinchey, 1995], pages 389–407.

West, Margaret M. (1996). Animation is approximation. Technical Report 96.03, School of Computer Studies, University of Leeds.

Winikoff, Michael (1997). Animating Z specifications. Technical report, Department of Computer Science, Melbourne University. To appear.

Winikoff, Michael and Dart, Philip (1997). Translating Z to logic. Technical Report 97/14, Department of Computer Science, Melbourne University.