Verifying Model Oriented Specifications through Animation

Ed. Kazmierczak Michael Winikoff Philip Dart

{ed,winikoff,philip}@cs.mu.oz.au
Department of Computer Science
The University of Melbourne

Abstract

In this paper we demonstrate how light weight tools can be used to increase the level of confidence in **Z** specifications. In particular we outline the Pipedream approach to exploring **Z** specifications through animation, and illustrate the range of analyses that can be performed. We argue that, while a light weight approach does not give the same levels of assurance that an automated reasoning system would, it does give levels of assurance which are adequate for most projects and with significantly less overhead. We illustrate how animation can be used to perform verification using the example of a simple dependency management system.

Keywords: Animation, Verification, Assurance, Formal Methods, **Z**.

1 Introduction

All software engineering projects should provide a high level of assurance, that is, a high degree of confidence, that the final system meets the needs for which it was engineered. An important factor that distinguishes engineering from ad hoc development is in being able to exert control over the level of assurance achieved in a project to suit the purpose of the project. Life and livelihood critical systems require the highest levels of assurance that the system meets the needs for which it was engineered. For many projects, however, lesser levels of assurance are acceptable.

In this paper our interest is in improving the outcomes of requirements analysis by using formal methods, or more precisely mathematical modelling, to determine, analyse and verify requirements. To this end, we need to create mathematical models of the systems of interest, which when written in a language like **Z** become the formal specification of the system. Unfortunately, there are few methods for developing and validating models of large complex systems and not much by way of tool support.

By building a model of a system it is possible to evaluate the quality[1] of the requirements or a design prior to constructing the system. If the model is expressed in a formal notation then the evaluation of the model can be assisted by various tools. Ideally, modelling is an iterative process [9, 15] with short cycle times providing rapid convergence to a correct model of the proposed system. An iteration consists of a modelling step, followed by an evaluation of the model, followed by an improvement of the model based

[1] The IEEE Recommended Practice for Software Requirements Specifications [10] lists eight characteristics of a quality Software Requirements Specification: that it is correct, unambiguous, complete, and consistent, and that the requirements are ranked for importance and stability, and are verifiable, modifiable and traceable.

on the evaluation results. We can evaluate a model by *exploring*[2] it and its consequences using a range of techniques and tools such as theorem proving, animation, testing, and prototyping. Each of these exploration methods has advantages and drawbacks.

In this paper we focus on animation. Animating specifications is particularly promising for developing formal specifications for a number of reasons. First, and arguably most importantly, animation can be highly automated and thus cheap to perform. In addition, static analysis of the specification can help to provide insight into the specification and the assumptions and implications implicit in the model. Second, animation can be very effective at detecting problems with the specification because animating a specification provides a means of testing a model *interactively*. If the properties that we are interested in can be formulated in the mathematical language of the specification then these too can be tested interactively against the model. These two properties of animation make animation very suitable for earlier iterations when the model is more likely to be incorrect or incomplete and still evolving.

The third reason why animation is promising is that it does not require extensive expertise like that required for theorem proving. The developer does not need detailed knowledge of the underlying mathematical theories or the proof strategies required to prove theorems. The fourth reason why animation is promising is that it allows a (formal) specification to be demonstrated to a client interactively which is useful since it permits validation to be carried out. By bringing a formal specification to life, animation enables clients and domain experts to provide valuable feedback. Finally, unlike some prototyping methods, an explicit relationship between the specification and the animation can be maintained which prevents the divergence of the specification and the animation.

One drawback of animation is that, like other testing oriented approaches, it relies on finding counter-examples to properties, that is it can't always be shown that a property holds but rather a counter-example to property is found and so shows that for some values at least, it is false. Consequently, an animation can never *prove* that the formulation of a model is consistent or correct or complete but it can be used to obtain the same level of confidence in the consistency, correctness or completeness of the model as that given by testing.

Animation falls into the class of *light weight* [5] approaches to formal methods. A light weight approach trades off completeness for speed and ease of use. It is characterised by an emphasis on automation and by a willingness to accept lower levels of assurance in return for more rapid feedback and a reduced reliance on user-expertise. The light weight approach is in contrast to the rigorous or formalist verification methods which rely more on mathematical proof and automated reasoning. Consequently, light weight tools and techniques are viewed in this paper as complementary to rigorous or formal verification methods. Further, light weight techniques have been successfully applied to industrial projects [1, 2] which makes them the subject of current interest in formal methods.

Perhaps a good analogy for the kinds of assurance given by animation is that given by a spell-checker. The spell-checker can be used to detect most spelling errors in a document and so increase our level of confidence in the document, but no-one would claim that once the document has been spell-checked then it is error free. The spell-checker analogy is not quite accurate because even with light weight tools the levels of confidence that are provided should be as high as possible without sacrificing their ease of use and their lower conceptual overheads. Note that despite their shortcomings, spell-checkers are quite useful in practice.

While there are clear benefits to animating specifications, the analysis and verification of models using typical animation tools is not well understood. Previous papers in animating **Z** specifications have concentrated on describing tool architectures as in [16], outlining approaches to animating **Z** specifications as in

[2]Exploration here is taken to mean both the verification and validation of the model as well as the exercising of the model by executing various scenarios or test cases which increase the developer's and the client's understanding of the system and the problem domain.

[17], or showing examples of animation [3, 4]. Little, if any, attention has been given to the way in which animation can be used to explore a specification for verification or validation purposes. None of these papers directly discuss the exploration of models expressed in the **Z** notation, which is our major aim.

Previous papers discussing light weight tools concentrate on showing how light weight tools can aid the requirements modelling process as in [6], but only by conducting non-rigorous analyses and verification of the model against client understanding. These studies are valuable, but in this paper we seek to go beyond this and to put light weight approaches to formal methods on a more rigorous basis.

The aim of this paper is twofold: (1) to study the analysis and verification of a simple dependency management system written in **Z** using the Pipedream light weight tool set[3], and (2) relate the verification back to the requirements modelling and analysis task. The primary contribution of this paper is to argue that animation can be used as an exploration tool and that the exploration activity enabled by animation can be used to perform a range of verification and validation tasks, and is highly effective as a cheap way of finding errors.

The remainder of this paper is organised as follows. In section 2 a simple example is presented which is used in sections 3 and 4 to illustrate the major concepts. In section 3 we describe the Pipedream approach and in particular we describe the *mode* and *subtype* analyses which are used to derive computational information from **Z** specifications. In section 4 we show how the Pipedream tools can be used to increase our level of confidence in the Dependency Management System model. We conclude in section 5

2 The Dependency Management System

The Dependency Management System (DMS) is presented in [7] and essentially maintains a directed acyclic graph of unspecified data. The data stored in the dependency management system is a parameter to the specification

$$[DATA]$$

and is not analysed further. The state schema is then given below where *nodes* is the set of nodes in the graph, *ddo* (directly depends on) is the set of dependencies (or edges) in the graph and *tc* is the transitive closure. The initialisation sets the *nodes* to $\emptyset$ and from this and the state invariant it can be inferred that $ddo = \emptyset$ and $tc = \emptyset$.

DepManSys

$$nodes : \mathbb{F}\, DATA$$
$$ddo : DATA \leftrightarrow DATA$$
$$tc : DATA \leftrightarrow DATA$$

$$ddo \subseteq nodes \times nodes$$
$$tc = ddo^{+}$$
$$\neg\, (\exists\, x : DATA \bullet (x, x) \in tc)$$

InitDepManSys

$$DepManSys'$$

$$nodes' = \emptyset$$

Throughout the remainder of the specification the usual conventions for Δ and Ξ are assumed [12]. With these definitions in place there are now two operations for observing properties of the dependency graph:

[3] Pipedream is the name of the project researching light weight tools and methods at The University of Melbourne. The Pipedream approach to animation is discussed in more detail in section 3.

DependsOn

$\Xi DepManSys$
$d? : DATA$
$dependents! : \mathbb{F}\, DATA$

$d? \in nodes$
$dependents! = \{n : nodes \mid (n, d?) \in tc\}$

Finally, there are the usual operations for adding and removing nodes

AddNode

$\Delta DepManSys$
$d? : DATA$

$d? \notin nodes$
$nodes' = nodes \cup \{d?\}$
$ddo = ddo'$

RemoveNode

$\Delta DepManSys$
$d? : DATA$

$d? \in nodes \setminus \mathrm{ran}\, ddo$
$nodes' = nodes \setminus \{d?\}$
$ddo' = \{d?\} \nrightarrow ddo$

and adding and removing dependencies.

AddDependency

$\Delta DepManSys$
$x?, y? : DATA$

$\{x?, y?\} \subseteq nodes$
$ddo' = ddo \cup \{(x?, y?)\}$
$nodes' = nodes$

RemoveDependency

$\Delta DepManSys$
$x?, y? : DATA$

$(x?, y?) \in ddo$
$ddo' = ddo \setminus \{(x?, y?)\}$
$nodes' = nodes$

The dependency management system presented in this section will be used throughout the remainder of this paper to illustrate the concepts involved in animating the specification as well as illustrating how several common properties of **Z** specifications can be verified.

3 The Pipedream Approach

3.1 Analysing and Animating Z Specifications

The Pipedream approach to creating and verifying models is to combine static analysis and animation to systematically explore properties of the model, including the correctness and completeness of the *functions* specified in the model, and the *behaviour* implied by the model.

After typechecking, the first step is to translate the type correct **Z** specification into first order set theory. Each definition in **Z** is translated to its underlying semantics expressed as the definition of a predicate which extends first order set theory. Consider, for example, the schema

DepManSys

$nodes : \mathbb{F}\, DATA$
$ddo : DATA \leftrightarrow DATA$
$tc : DATA \leftrightarrow DATA$

$ddo \subseteq nodes \times nodes$
$tc = ddo^+$
$\neg\ (\exists\, x : DATA \bullet (x, x) \in tc)$

which is translated to

$$\forall\, nodes \forall\, ddo \forall\, tc \bullet DepManSys(nodes, ddo, tc) \Leftrightarrow$$
$$nodes \in \mathbb{F}\, DATA \wedge ddo \in DATA \leftrightarrow DATA \wedge tc \in DATA \leftrightarrow DATA \wedge$$
$$ddo \subseteq nodes \times nodes \wedge tc = ddo^+ \wedge \neg\ (\exists\, x \bullet (x, x) \in tc)$$

Note that the full translation also includes a global environment which is passed around and used to access global values such as *DATA*.

The translated forms of the **Z** statements constitute a set of clausal definitions [8] for the **Z** constructs in the specification. This will be referred to as the *clausal form* of a **Z** construct. We will also refer to the translation of the **Z** specification as the clausal form of the specification. The details are given in [20].

The clausal form of the **Z** specification is not necessarily executable. Indeed, the predicates in the clausal form of the specification may involve infinite or even uncountable sets and predicates which are not computable. The next step is to *analyse* the clausal form of the **Z** specification in order to derive more information about the objects making up the specification: which *modes* predicates can be executed in, the type structure of the specification, which data sets are finite and which predicates are deterministic. With this information the clausal form of the specification can be *transformed* so that it is more readily executed.

Mode analysis is used to determine control information. The control information specifies how to execute a formula by specifying the flow of control, for example, consider the predicate *AddNode* below

$$AddNode(nodes, nodes', ddo, ddo', tc, tc', d?) \Leftrightarrow$$
$$DepManSys(nodes, ddo, tc) \wedge DepManSys(nodes', ddo', tc')$$
$$\wedge\ d? \notin nodes \wedge ddo = ddo' \wedge nodes' = nodes \cup \{d?\}.$$

One possible way of computing with this predicate is to provide values for *nodes*, *ddo* and $d?$ and then compute the value for $nodes'$, ddo', *tc* and tc'. This mode for *AddNode* is where the values of *nodes*, *ddo* and $d?$ *determine* the values of $nodes'$, ddo', *tc* and tc' which is written as

$$AddNode :: \{nodes, ddo, d?\} \Rightarrow \{nodes', ddo', tc, tc'\}.$$

Another mode of use is to provide the values for $nodes'$, ddo' and $d?$ and compute possible values of *nodes*, *ddo*, *tc* and tc', that is, the mode

$$AddNode :: \{nodes', ddo', d?\} \Rightarrow \{nodes, ddo, tc, tc'\}.$$

In the first mode the values of *nodes*, *ddo* and $d?$ are required while in the second mode, the values for $nodes'$, ddo' and $d?$ need to be supplied. If the set $nodes'$ is finite then *nodes* can easily be computed as the set difference of $nodes'$ and $\{a\}$, but observe that the algorithm for computing $nodes'$ in the first mode is quite different to that for computing *nodes* in the second mode. Thus, the control information is essentially

an abstract description of the possible combinations of arguments and computed results in a predicate. Of course, the *AddNode* predicate can also be assigned the mode

$$AddNode :: \{nodes', ddo', nodes\} \Rightarrow \{d?, ddo, tc, tc'\}$$

Modes play a number of roles in the analysis. First, if a predicate can be assigned a mode then there is a procedure for *executing* the predicate in that mode, for example, if *AddNode* is assigned the mode

$$AddNode :: \{nodes', ddo', d?\} \Rightarrow \{nodes, ddo, tc, tc'\} \quad (1)$$

then there is a procedure for computing *nodes*, *ddo*, *tc* and tc' from $nodes'$, ddo' and $d?$. Thus, mode analysis determines which predicates can be given an executable semantics and how they can be used in practice. Second, the modes determined for a predicate can be used to select an efficient implementation for a predicate. If mode analysis determines the mode in (1) for *AddNode* and finiteness analysis detects that the definition of *AddNode* has only one possible result in this mode then a function can be used to implement the *AddNode* predicate in this mode. On the other hand, to implement *AddNode* in the mode

$$AddNode :: \{nodes', ddo', d?\} \Rightarrow \{nodes, ddo, tc, tc'\}$$

requires finding all tuples $(nodes, ddo, tc, tc')$ such that $nodes' = nodes \cup \{d?\}$ and the predicate *DepManSys* is satisfied. This in turn requires that the union operation $union(x, y, z) \Leftrightarrow z = x \cup y$ can be used in the mode $\{z\} \Rightarrow \{x, y\}$. The algorithms for mode analysis in the context of animating **Z** specifications are given in [18, 19].

Z's type system consists of a class of *base* types and can be simply extended to include a hierarchy of *subtypes* [13, 14] which are defined in terms of the base types. Operations like $\_ \frown \_$ are only guaranteed to return a sequence if the two arguments are also sequences, but sequences themselves are defined in terms of the base type $\mathbb{P}(\mathbb{N} \times A)$ where A is the type of the elements of the sequence. Subtype analysis determines which checks can be removed in the clausal form of a **Z** paragraph.

For example, consider the *DepManSys* schema which includes a conjunct of the form $\neg\ \exists x : DATA \bullet (x, x) \in tc$ which translates to $\neg\ \exists x, y \bullet x \in DATA \wedge y \in tc \wedge y = (x, x)$. Since the specification is type correct we can remove the conjunct $x \in DATA$ which leaves the predicate $\neg\ \exists x, y \bullet y \in tc \wedge y = (x, x)$. The mode analysis returns the mode $\{tc\} \Rightarrow \{\}$ and consequently determines that the predicate is executable for input *tc*. Indeed, this can be seen by recognising that $y \in tc$ can be executed in the mode $\{tc\} \Rightarrow \{y\}$ and then $y = (x, x)$ is a simple test.

3.2 Soundness of the Animation

The underlying computational model, which is used to execute the clausal form of the specification, is the SLDNF-resolution strategy [8] used in Mercury [11]. This seems natural as the strategy is explicitly aimed at executing programs composed of clauses efficiently. The analyses discussed in the previous section are used to transform the clauses into an equivalent form which is executable and which is is *sound* with respect to the SLDNF-resolution strategy. The developer can run *queries* and get back *answers*.

Consider again the dependency management system from section 2. The state, initialisation and *DependsOn* schemas from the dependency management system are shown in their translated and optimised form in figure 1. Given that *DependsOn* can be executed in the mode

$$\{nodes, ddo, d?\} \Rightarrow \{dependents!, nodes', ddo', tc', tc\}$$

and that we have the following definitions

$$nodes = \{a, b, c, d\}, ddo = \{(a, b), (a, c), (a, d)\}, \text{ and } d? = c$$

Figure 1 The state, initialisation and *DependsOn* schemas in translated form.

$$DepManSys(nodes, ddo, tc) \Leftrightarrow$$
$$\quad finite(nodes) \land ddo \subseteq nodes \times nodes \land tc = ddo^{+} \land \neg \exists x \bullet (x, x) \in tc.$$

$$InitDepManSys(nodes', ddo', tc') \Leftrightarrow$$
$$\quad DepManSys(nodes', ddo', tc') \land nodes' = \emptyset.$$

$$DependsOn(nodes, ddo, tc, nodes', ddo', tc', d?, dependents!) \Leftrightarrow$$
$$\quad DepManSys(nodes, ddo, tc) \land DepManSys(nodes', ddo', tc') \land$$
$$\quad d? \in nodes \land finite(dependents!) \land$$
$$\quad nodes' = nodes \land tc' = tc \land ddo' = ddo \land$$
$$\quad dependents! = \{n : nodes \mid (n, d?) \in tc\}.$$

A query of the form

$$DependsOn(nodes, ddo, tc, nodes', ddo', tc', d?, dependents!) \tag{2}$$

can be presented to the system which yields the answer $dependents! = \{a\}$. Perhaps the most important property for an animation, like that in figure 1, is that the animation system is *sound*. If a query is supplied to the system and an answer is obtained, for example, the answer $dependents! = \{a\}$[4] in the case of the query above, then that answer must be a logical consequence of the clauses making up the program.

The answer $dependents! = \{a\}$ is actually a substitution $\theta = \{dependents! \mapsto \{a\}\}$ which, if substituted back into the original query, makes it true. In general, if A is the clausal form of a specification and θ is a substitution resulting from an answer to a query G then $G\theta$ is a logical consequence of A, that is,

$$A \models G\theta. \tag{3}$$

Further, we also want to be able to conclude that $G\theta$ also follows from the original **Z** specification S and this requires that A is sound with respect to S, that is, if $A \models G\theta$ then $G\theta$ is a consequence of the semantics of the **Z** specification. Our animation strategy is aimed at preserving this property.

In the absence of property (3) the developer could never be certain that an answer derived from the system is a consequence of the specification. Prolog systems, because of their omission of the occurs check in unification can violate property (3) and so extra care must be taken if generating Prolog to ensure that all unifications are *safe* [8]. Fortunately, Mercury uses its own static mode, type and determinism analyses to ensure that (3) is not violated. The omission of the occurs check in Prolog is largely a matter of efficiency [8] but the loss of soundness makes Prolog an unsafe language for animation.

4 Verifying Properties of the Dependency Management System Model

Once a specification has been written then there is a need converge rapidly to a correct model of the system. In each iteration of the modelling/evaluation cycle, we can determine if we have captured properties essential to the system by testing these properties against the specification. This section illustrates how this testing technique can be used to verify specific properties of the dependency management system specification.

[4] Along with values for tc, tc', $nodes'$ and ddo'

4.1 Performing an Initialisation Check

The usual check performed for the initialisation schema ensures that the initial state is a valid state, that is, the property:

$$\exists\, InitDepManSys \bullet true$$

In the case of the dependency management system this means simply showing that *InitDepManSys* is a logical consequence of the clausal form of the specification which in turn means simply executing the query,

$$\exists\, nodes', ddo', tc' \bullet InitDepManSys(nodes', ddo', tc')$$

Using the *InitDepManSys* predicate in this way means using it in the mode

$$InitDepmanSys :: \{\} \Rightarrow \{nodes', ddo', tc'\}$$

which is a valid mode for *InitDepManSys* and so the query can simply be executed. The resulting execution appears as follows

$$\begin{array}{l} \vdash InitDepManSys(nodes', ddo', tc') \\ \vdash nodes' = \emptyset \wedge DepManSys(nodes', ddo', tc') \\ \vdash DepManSys(\emptyset, ddo', tc') \\ \vdash finite(\emptyset) \wedge ddo' \subseteq \emptyset \times \emptyset \wedge tc' = ddo'^{+} \wedge \neg\ \exists\, Y \bullet (Y, Y) \in tc' \\ \vdash true \end{array}$$

which succeeds because all of the conjuncts in the third line are executable. Note that $\subseteq$ must be used in the mode $x \subseteq y :: \{y\} \Rightarrow \{x\}$ in order to do this.

4.2 Verifying Preconditions

A more interesting check is to compare an expected precondition with the calculated precondition of an operation [5]. Consider the following modified version of the *RemoveNode* schema

$$\begin{array}{|l} \textit{RemoveNode}_A \\ \hline \Delta DepManSys \\ d? : DATA \\ \hline d? \in nodes \setminus \mathrm{ran}\ ddo \\ nodes' = nodes \setminus \{d?\} \\ \mathbf{ddo'} = \mathbf{ddo} \\ \hline \end{array}$$

The expected precondition for the $RemoveNode_A$ is that the dependency management system state *DepManSys* is valid and that no nodes in the system depend on the node to be removed $d?$. The calculated precondition is $\exists\, DepManSys' \bullet RemoveNode_A$. The check is to show that the expected precondition is stronger than the calculated precondition, that is, we want to show a property of the form

$$\forall\, DepManSys;\ d? : DATA \bullet d? \in nodes \setminus \mathrm{ran}\ ddo \Rightarrow \exists\, DepManSys' \bullet RemoveNode_A \qquad (4)$$

[5] See [12] for more on the precondition check.

The query corresponding to this property is

$$\neg\ \exists\, nodes, ddo, tc, d? \bullet DepManSys(nodes, ddo, tc) \land d? \in nodes \setminus \operatorname{ran} ddo \land \\ \neg\ \exists\, nodes', ddo', tc' \bullet RemoveNode_A(nodes, ddo, tc, d?, nodes', ddo', tc')$$

which would return either true or false but cannot be moded. However, we can execute specific instances of this query, for example, given

$$nodes = \{a, b\},\ ddo = \{a \mapsto b\}\ and\ d? = a$$

we can execute the query which fails and this in turn means that the values above give a counter-example to the conjecture. This is a similar approach to that used in model checking. The sequence of execution steps appears as follows:

$$\begin{array}{l} \vdash\ \neg\ \exists\, tc \bullet (DepManSys(\{a,b\}, \{a \mapsto b\}, tc) \land a \in \{a,b\} \setminus \operatorname{ran}\{a \mapsto b\} \land \\ \quad \neg\ \exists\, nodes', ddo', tc' \bullet RemoveNode_A(\{a,b\}, \{a \mapsto b\}, tc, nodes', ddo', tc', a)) \\ \vdash\ \neg\ (\neg\ \exists\, nodes', ddo', tc' \bullet RemoveNode_A(\{a,b\}, \{a \mapsto b\}, \{a \mapsto b\}^+, nodes', ddo', tc', a)) \\ \vdash\ \exists\, nodes', ddo', tc' \bullet \\ \quad \{a \mapsto b\} \subseteq \{a,b\} \times \{a,b\} \land tc = \{a \mapsto b\}^+ \land \neg\ (\exists\, x \bullet (x,x) \in \{a \mapsto b\}^+) \land \\ \quad a \in \{a,b\} \setminus \operatorname{ran}\{a \mapsto b\} \land nodes' = \{a,b\} \setminus \{a\} \land ddo' = \{a \mapsto b\} \land \\ \quad \{a \mapsto b\} \subseteq \{b\} \times \{b\} \land tc' = \{a \mapsto b\}^+ \land \neg\ (\exists\, x \bullet (x,x) \in \{a \mapsto b\}^+) \end{array}$$

The second step is derived from the first step by unfolding *DepManSys* by its definition and then evaluating all of the conjuncts in *DepManSys* and $a \in \{a,b\} \setminus \operatorname{ran}\{a \mapsto b\}$ to true[6] The predicate in the third step fails because of the conjunct $\{a \mapsto b\} \subseteq \{b\} \times \{b\}$ which is no longer true after a has been removed from the set of nodes. The statement $ddo' = ddo$ in the $RemoveNode_A$ schema is at fault.

The failure in the third step of $\{a \mapsto b\} \subseteq \{b\} \times \{b\}$ needs to be related back to the specification in order to make the deduction that $ddo' = ddo$ leads to the problem. From the method viewpoint, a property of the dependency management system was conjectured and then shown to false by finding a counter-example. If a theorem prover were being used to prove this property, a counter-example would still have been needed in order to show the negation of the conjecture. Consequently, in this case the same method of proof has been used in the animation as would have been used in a theorem prover to show the conjecture false.

4.3 A Simple Reachability Property

More challenging than properties of schemas are properties of traces. Define a relation R to be *non-branching* if and only if

$$\forall\, x, y, z \bullet xRz\ \land\ yRz\ \Rightarrow x = y,$$

that is if the relation R is injective. Suppose we are interested in the relation *ddo* in the dependency management system and its branching properties. Define the schema

NonBranching

DepManSys

$ddo \in \rightarrowtail$

[6] We have already discussed the execution of $\neg\ \exists\, x \bullet (x,x) \in tc$ in section 3.1.

It is obvious that the *AddDependency* schema does not in general preserve the non-branching property. If the dependency management system is in a state which is non-branching after applying the *AddDependency* operation, must it have been in a non-branching state before the operation? The formulation appears as follows

$$\forall\, AddDependency \bullet NonBranching' \Rightarrow Nonbranching$$

The logically equivalent query is

$$\begin{array}{l} \neg\ \exists\, nodes, nodes', ddo, ddo', tc, tc', x?, y? \bullet \\ \quad AddDependency(nodes, ddo, tc, nodes', ddo', tc', x?, y?) \wedge NonBranching(nodes', ddo', tc') \wedge \\ \qquad \neg\ NonBranching(nodes, ddo, tc) \end{array}$$

which has the mode $\{nodes', ddo', x?, y?\} \Rightarrow \{nodes, ddo, tc, tc'\}$. Consequently let
$nodes' = \{a, b, c\}$, $ddo' = \{a \mapsto b, b \mapsto c\}$, $x? = a$ and $y? = b$.

The key point in executing this query is that the *AddDependency* must be executed *backwards,* that is, the values of variables in the pre-state *nodes, ddo* and *tc* must be obtained from the values of the variables in the post-state. The predicate *NonBranching* in the conclusion acts only as a test that *ddo* in the pre-state is non-branching. It is the relational aspects of logic programming languages which permit this kind of execution The sequence of execution steps appears as follows:

$$\begin{array}{l} \vdash\ \neg\ \exists\, nodes, ddo, tc, tc' \bullet \\ \quad AddDependency(nodes, ddo, tc, \{a, b, c\}, \{a \mapsto b, b \mapsto c\}, tc', a, b) \wedge \\ \quad NonBranching(\{a, b, c\}, \{a \mapsto b, b \mapsto c\}, tc') \wedge \\ \quad \neg\ NonBranching(nodes, ddo, tc) \\ \vdash\ \neg\ \exists\, nodes, ddo, tc, tc' \bullet \\ \quad (DepManSys(nodes, ddo, tc) \wedge DepManSys(\{a, b, c\}, \{a \mapsto b, b \mapsto c\}, tc') \wedge \\ \qquad \{a, b\} \subseteq nodes \wedge \{a \mapsto b, b \mapsto c\} = ddo \cup \{a \mapsto b\} \wedge \{a, b, c\} = nodes) \wedge \\ \quad NonBranching(\{a, b, c\}, \{a \mapsto b, b \mapsto c\}, tc') \wedge \\ \quad \neg\ NonBranching(nodes, ddo, tc) \\ \vdash\ \neg\ \exists\, ddo, tc \bullet \\ \quad ddo \cup \{a \mapsto b\} = \{a \mapsto b, b \mapsto c\} \wedge DepManSys(\{a, b, c\}, ddo, tc) \wedge \\ \quad \neg\ NonBranching(\{a, b, c\}, ddo, tc) \end{array}$$

The two calls to the *DepManSys* in the second step are tests to ensure that the values of the variables in the pre-state and the post-state satisfy the state invariant. The key step in executing *AddDependency* backwards is to determine *ddo* from $\{a \mapsto b, b \mapsto c\} = ddo \cup \{a \mapsto b\}$ and *nodes* from $\{a, b, c\} = nodes$. Once this is done *tc* and tc' are easily determined and the remaining conjuncts are simple executable tests and this example succeeds.

We can vary the values supplied to the query in above in order to increase the coverage of the input domain and the cases of interest. Each successful test increases the level of confidence that the property holds of the specification. Further, we can start in a particular state and keep executing *AddNode* and *AddDependency* predicates until we reach the initial state. More generally, for unsafe states, this procedure would let us determine if that state was reachable from the initial state.

5 Conclusions/Further work

The ability to execute predicates in different modes allows the specification to be explored in a number of different ways. The examples in section 4 demonstrate some of the possibilities. There, we've illustrated how

a check on the initialisation can be performed, how a pre-condition check can be performed and have indicated how a more complex property like a reachability property can be tested. From a modelling perspective, these tests increase our confidence in the consistency and correctness of the model. More interestingly, the testing of the non-branching property illustrates how trace properties can be explored interactively, and can be used to provide insight into the implied *behaviour* of the model. The analysis of predicates helps the testing process. We can readily see from the mode analysis which values need to be provided and consequently which domains need to be covered by the tests. Further, the tests were completely automatic once the desired property had been properly formulated which facilitates exploration of a specifications by inexpert users.

The general method for exploring specifications is characterised by a search for counter-examples and this is no different to testing methods in general. However, in Pipedream we are not simply executing operations to test the values resulting from the operations for certain test cases. We are using animation to explore general properties of the specification and these properties can be any assertion which can be expressed in the language of the specification. To carry out this kind of exploration requires that the tools are logically sound[7]. The Pipedream approach draws on results from logic programming and analysis to achieve this.

In addition to continuing to develop our toolset, we intend to develop a collection of properties and techniques for verifying and/or testing these properties using animation technology. In particular, the application of animation to validation activities (by demonstration to clients and/or domain experts) deserves further investigation. Finally, we intend to formally prove our tools correct, and to look at the benefits and issues involved in integrating our tools with other tools such as theorem provers.

Acknowledgements

The authors would like to thank Leon Sterling at The University of Melbourne for his interest in and support of this work, Dan Hazel and Owen Traynor at the University of Queensland for their interest and many useful discussions, and the SVRC for their support of Michael Winikoff's visit there.

References

[1] Juan Bicarregui, Jeremy Dick, Brian Matthews, and Eoin Woods. making the most of formal specification through animation, testing and proof. *Science of Computer Programming*, 29:53–78, 1997.

[2] Steve Easterbrook, Robyn Lutz, Richard Covington, John Kelly, Yoko Amp, and David Hamilton. Experiences using lightweight formal methods for requirements engineering. *IEEE Transactions on Software Engineering*, 24(1), 1998.

[3] Daniel Hazel, Paul Strooper, and Owen Traynor. Requirements engineering and verification using specification animation. Manuscript, May 1998.

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

[5] D. Jackson and J. Wing. Lightweight formal methods. *IEEE Computer*, 29(4):21, April 1996.

[7] A property which is absent from other Z animation tools.

[6] Xiaoping Jia. A pragmatic approach to formalizing object-oriented modeling and development. In *Proc. 21st Annual Int'l Computer Software and Applications Conf.*, pages 240–245, Washington, D.C., USA, August 1997.

[7] Peter Lindsay and Erik van Keulen. Case studies in the verification of specifications in VDM and Z. Technical Report 94-3, Software Verification Research Center, Department of Computer Science, The University of Queensland, March 1994.

[8] J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, second edition, 1987.

[9] D.N.P. Murthy, N.W. Page, and E.Y. Rodin. *Mathematical Modelling: A Tool for Problem Solving in Engineering Physical, Biological and Social Sciences*. Pergamon Press, 1990.

[10] Software Engineering Standards Commitee of the IEEE. *IEEE Recommended Practice for Software Requirements Specifications*. IEEE, 1993.

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

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

[13] J. M. Spivey. Richer types for Z. *Formal Aspects of Computing*, 8:565–584, 1996.

[14] J. M. Spivey and B. A. Sufrin. Type inference in Z. In D. Bjørner, C. A. R. Hoare, and H. Langmaack, editors, *VDM and Z – Formal Methods in Software Development*, volume 428 of *Lecture Notes in Computer Science*, pages 426–438. VDM-Europe, Springer-Verlag, 1990.

[15] A.M. Starfield, K.A. Smith, and A.L. Bleloch. *How to Model It: Problem Solving for the Computer Age*. McGraw-Hill Publishing Company, 1990.

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

[17] M. M. West and B. M. Eaglestone. Software development: Two approaches to animation of Z specifications using Prolog. *IEE/BCS Software Engineering Journal*, 7(4):264–276, July 1992.

[18] M. Winikoff, P. Dart, and E. Kazmierczak. Rapid prototyping using formal specifications. In Chris McDonald, editor, *Proceedings of the 21st Australasian Computer Science Conference*, pages 279–293. Springer-Verlag, February 1998.

[19] Michael Winikoff. Analysing modes and subtypes in Z specifications. Technical Report 98/2, Department of Computer Science, Melbourne University, 1998.

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