Working Paper Series
ISSN 1170-487X

Experiences Using Z Animation Tools

By Greg Reeve and Steve Reeves

Working Paper 01/3
May 2001

Department of Computer Science
The University of Waikato
Private Bag 3105
Hamilton, New Zealand

Experiences using Z animation tools

Greg Reeve and Steve Reeves

Department of Computer Science,
University of Waikato, New Zealand

Abstract. In this paper we describe our experience of using three different animation systems. We searched for and decided to use these tools in the context of a project which involved developing formal versions (in Z) of informal requirements documents, and then showing the formal versions to people in industry who were not Z users (or users of any formal techniques). So, an animator seemed a good way of showing the behaviour of a system described formally without the audience having to learn Z. A requirement, however, that the tools used have to satisfy is that they correctly animated Z (whatever that may mean) and they behave adequately in terms of speed and presentation. We have to report that none of the tools we looked at satisfy these requirements—though to be fair all of them are still under development.

1 Introduction

In this paper we describe our experiences of using some animation tools for the animation and testing of Z specifications. The purpose of this process was an attempt to find a freely available tool that could be used to demonstrate the benefits of formal specification to our industrial partners in the ISuRF research project. (Information about the ISuRF project can be found at the project web site `http://www.cs.waikato.ac.nz/cs/Research/fm/`.)

An animator is a system which, given inputs to a formally specified (in this case described in the formal specification language Z) system, calculates the outputs. One of the uses of animation of a specification is to allow us to see how a specified system behaves without needing to go to the expense of writing code. Also, we, together with the domain experts (who may not be Z users), can validate the specification against the informal requirements of the system without requiring the experts to understand Z. This seems to us to be a central requirement of such a process—we bring our expertise of Z to the validation, and the other participants bring their domain expertise—neither side should have to become as expert as the other in order for this process to be useful.

So, we have been using animators as a way of validating specifications, *i.e.* detecting errors and ensuring they do what we expect, and plan to use animation to present some specifications of real-world software to non-Z speakers. These uses imply certain obvious qualities for an animator: it should correctly express the semantics of Z; it should be efficient enough to be productively used; it should have an interface that makes it more understandable than the Z itself, and also provide useful feedback about the animation process. These qualities are amongst the ones against which we finally judged the systems we used—they seem to be the most obvious ones, though others will doubtless also suggest themselves to the reader.

The three tools that we have used to date are Possum, ZANS, and ZETA/ZAP.

Possum is an animator been developed by the Software Verification Research Centre at the University of Queensland in Australia (see [3])[1].

ZANS was created by Xiaoping Jia from the School of Computer Science, Telecommunication and Information Systems at DePaul University in Chicago[2].

ZETA is a framework for combining tools to edit, browse, analyse and animate Z specifications that was developed (and is still developing) at the Technishe Universität Berlin[3]. The animation tool that ZETA uses is called ZAP.

[1] URL: `http://www.svrc.it.uq.edu.au/pages/Animation.html`

[2] URL: `http://saturn.cs.depaul.edu/~fm/zans.html`

[3] URL: `http://uebb.cs.tu-berlin.de/zeta/`

The plan of the paper is as follows: we have three main sections, each of which describes the tool and then makes some comments about it. We finish with an overall conclusions section which aims to give guidelines, based on our experience, for a good animation tool for Z.

2 ZANS

2.1 ZANS' strategies

The ZANS animation approach separates operation schemas into two categories, explicit and non-explicit. An operation schema is said to be explicit if its outputs can be computed from its inputs. ZANS animates explicit schemas only as non-explicit schemas may require elaborate manipulation. However it is claimed by the author of ZANS that a study of the specifications in [2] has shown that 94% of the operation schemas are explicit or can be made so with some minor modifications, *e.g.* adding $tested = \varnothing$ to the standard version of the class manager assistant (see below) is an example of a necessary minor modification. In the standard version the only equation in the predicate part of *ClassInit* is $enrolled' = \varnothing$, and the fact that $tested' = \varnothing$ is inferred from the state invariant $tested \subseteq enrolled$, but ZANS does not do this, hence the need to add the extra equation.

ZANS translates explicit operation schemas into an intermediate notation known as *extended guarded command language* (EGC). EGC is an extension to Dijkstra's guarded command language.

To summarise the ideas and algorithm behind the translation mechanism:

- predicates involving equalities between post-names[4] and other simple expressions are converted to assignments, *e.g.* $x' = y + 4$ becomes $x := y + 4$
- these assignments are ordered so that the values of all the post-names can be computed, *e.g.* $x' := y' + 4;\ y' := 6$ is re-written so as to reverse the order of the assignments
- predicates which are not converted and which involve pre-names[5] become entry guards, and those involving post-names become exit guards, *e.g.* $x = y + 4$ is used as an entry guard and $x' * y = 0$ is used as an exit guard since it cannot be made into an assigment (at least not in general by any uniform technique that does not involve theorem-proving)

The following gives an example of this translation.

Example Z translation Given part of a typical Z specification of a class manager assistant [5]:

$[Student]$ $\qquad\qquad$ $Response ::= success \mid alreadyenrolled \mid \ldots$

$$
\begin{array}{|l}
\text{— } Class \text{ —} \\
enrolled, tested : \mathbb{P}\, Student \\
\hline
tested \subseteq enrolled \\
\hline
\end{array}
$$

$ClassInit \mathrel{\widehat{=}} [Class' \mid enrolled' = \varnothing \wedge tested' = \varnothing]$

$$
\begin{array}{|l}
\text{— } Enrolok \text{ —} \\
\Delta Class \\
s? : Student \\
r! : Response \\
\hline
s? \notin enrolled \\
enrolled' = enrolled \cup \{s?\} \\
tested' = tested \\
r! = success \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\text{— } AlreadyEnrolled \text{ —} \\
\Xi Class \\
s? : Student \\
r! : Response \\
\hline
s? \in enrolled \\
r! = alreadyenrolled \\
\hline
\end{array}
$$

$Enrol == Enrolok \vee AlreadyEnrolled$

ZANS produces the following EGC code ...

[4] post-names are those labels in a schema decorated with prime or exclamation mark.

[5] pre-names include undecorated labels and those decorated with a question-mark.

operation *ClassInit*
egc $[enrolled', tested' : \mathbb{P}\, Student]$
$\quad true \Rightarrow$
$\qquad enrolled' := \varnothing;\ tested' := \varnothing;$
$\quad \Leftarrow tested' \subseteq enrolled'$
cge
operation *Enrol*
egc $[enrolled, tested,$
$\quad enrolled', tested' : \mathbb{P}\, Student;$
$\quad s? : Student;\ r! : Response]$
$\quad s? \notin enrolled;\ tested \subseteq enrolled \Rightarrow$
$\qquad enrolled' := enrolled \cup \{s?\};$
$\qquad tested' := tested;\ r! := success;$
$\quad \Leftarrow tested' \subseteq enrolled'$
$[]\quad s? \in enrolled;\ tested \subseteq enrolled \Rightarrow$
$\qquad enrolled' := enrolled;$
$\qquad tested' := tested;$
$\qquad r! := alreadyenrolled;$
$\quad \Leftarrow tested' \subseteq enrolled'$
egc

The way that the translation goes should be quite evident. The *ClassInit* schema has *true* as its precondition, which becomes its entry guard. Then the equalities involving post-names become assignments directly and the state invariant $tested' \subseteq enrolled'$ becomes an exit guard. Because the *Enrol* operation involves schema disjunction the generated EGC code has a conditional construct denoted by [] and is guarded by the entry guards of the respective schemas in the disjunct.

ZANS is written in C++, it contains a class library to handle the mathematical objects in Z, *e.g.* sets, relations, *etc.* ZANS translates Z into EGC which is then interpreted to provide the animation.

The intermediate form (EGC) is proposed as the basis for efficient code generation from Z specifications.

2.2 Problems—ZANS

The evaluation of many predicates, for some reason, turn out to be 'undefined': for instance checking if a set of pairs is a function. ZANS reports this information as part of its command line output and continues the evaluation ignoring undefined predicates, or treating them as true. This can allow a solution despite breaking system invariants. For example if we modify the specification of the class manager system as follows:

Class
$enrolled : \mathbb{P}\, Student$
$tested : Student \rightarrow\!\!\!\!\!+ \mathbb{Z}$

$\mathrm{dom}\, tested \subseteq enrolled$

Testok
$\Delta Class$
$s? : Student$
$g? : \mathbb{Z}$
$r! : Response$

$s? \in enrolled$
$tested' = tested \cup \{s? \mapsto g?\}$
$enrolled' = enrolled$
$r! = success$

Leaveok

$\Delta Class$
$s? : Student$
$r! : Response$
$g! : \mathbb{Z}$

$s? \in enrolled$
$enrolled' = enrolled \setminus \{s?\}$
$((s? \in \mathsf{dom}\, tested \wedge tested' = \{s?\} \ndres tested \wedge r! = cert \wedge g! = tested\, s?)$
$\quad \vee (s? \notin \mathsf{dom}\, tested \wedge tested' = tested \wedge r! = nocert))$

ZANS will now allow us to enter more than one grade for a student (breaking the condition that *tested* is a function), giving the output:

$\ldots$
anim > **execute Testok**[6]
...Execute schema : *Testok*
Enter input arguments :
$s?$ − > **a**
$g?$ − > **11**
###*Try branch* #1
    ∗ ∗ ∗*Entry guards* :
        s? in enrolled
        −− > *True*
        dom tested subseteq enrolled
        −− > *True*
        tested in Student +−> *Z*
Exception : *ZMT class error* @ *Rel*().
        Run–time typing error.
        −− > *Undef*
    ∗ ∗ ∗*Statements* :
        tested' := *tested* || *s?*−> *g?*;
        enrolled' := *enrolled*;
        r! := *success*;

(continued)

∗ ∗ ∗*Exit guards* :
        dom tested' subseteq enrolled'
        −− > *True*
        tested' in Student +−> *Z*
Exception : *ZMT class error* @ *Rel*().
        Run–time typing error.
        −− > *Undef*
###*Branch* #1 *succeed*.
Schema : *Testok*
enrolled : *a*
tested : $(a, 10)$
enrolled' : *a*
tested' : $(a, 10), (a, 11)$
s? : *a*
g? : 11
r! : *success*

which is clearly incorrect because ZANS is unable to check whether the observation *tested* is a legitimate partial function. In this contrived example it is obvious that the output is not what the user expects, however as specifications become larger and more complex it becomes less obvious. A revised user interface could improve this problem by explicitly concluding that this is a possible answer given that some invariants are ignored.

ZANS appears to have shortcomings in the way Z is interpreted, for example

- free types are not considered to be sets (or if the are considered to be sets then they are always empty sets), which means checking membership of an element from a free type is not possible. For example using the free type definition of *Response* from the class manager specification and using the ZANS command line evaluation of predicates results in the output:

  anim > **pred success \in Response**
  Exception : *ZMT class error* @ *in*().
      Run − *time typing error*. *Expecting Set*.
  Undef

- ZANS is unable to pick an arbitrary element from a set. For example consider:

$$\begin{array}{|l}\hline S \\ x : \mathbb{N} \\ \hline\end{array}$$

$$init \mathrel{\widehat=} [S' \mid (\exists\, a : \{1, 2, 3\} \bullet x' = a)]$$

results in the output:

$\#\#\# Branch\ \#1\ succeed.$
$Schema : init$
$x' : \quad < undef >$

– promotion is not well supported because schemas do not appear to be considered a set of bindings, but rather as something that has a current state. In general we want to hide the local state that is being promoted, however doing this means ZANS fails to find a solution. For example consider the specification:

$$\begin{array}{|l}\hline R \\ x : \mathbb{P}\,\mathbb{Z} \\ \hline\end{array} \qquad \begin{array}{|l}\hline newR \\ R' \\ \hline x' = \varnothing \\ \hline\end{array} \qquad \begin{array}{|l}\hline changeR \\ \Delta R \\ xin? : \mathbb{Z} \\ \hline x' = x \cup \{xin?\} \\ \hline\end{array}$$

$$\begin{array}{|l}\hline S \\ db : \mathbb{N} \pfun R \\ \hline\end{array} \qquad \begin{array}{|l}\hline init \\ S' \\ \hline db' = \varnothing \\ \hline\end{array}$$

$$\begin{array}{|l}\hline newS \\ \Delta S \\ R' \\ \hline db' = db \oplus \{(\#db) \mapsto \theta R'\} \\ \hline\end{array} \qquad \begin{array}{|l}\hline rTos \\ \Delta S \\ \Delta R \\ s? : \mathbb{Z} \\ \hline s? \in \operatorname{dom} db' \\ (\theta R) = db\ s? \\ db' = db \oplus \{s? \mapsto \theta R'\} \\ \hline\end{array}$$

$$addR \mathrel{\widehat=} (newR \land newS) \setminus (x') \qquad updateR \mathrel{\widehat=} changeR \land rTos$$

$$updateR1 \mathrel{\widehat=} (changeR \land rTos) \setminus (x, x')$$

Now we can add new schema bindings to the partial function *db* in the schema *S* as expected.

anim > **execute addR**
$\ldots Execute\ schema :\ addR$
$\#\#\# Try\ branch\ \#1$
$\ldots$
$\#\#\#\ Branch\ \#1\ succeed.$
$Schema :\ addR$
$db : \quad \{\}$
$db' : \ \{0{-}{>}{<}|\ \ x :\ \{\}\ \ |{>}\}$

The *updateR* operation schema can be used to update one of the bindings in *db*. However each successive application of *updateR* will only work for that binding.

...
anim > **execute addR**
... *Execute schema* : *addR*
...
anim > **execute addR**
... *Execute schema* : *addR*
...
$db : \{0{-}{>}{<}|\ x : \{\}\ |{>}\}$
$db' : \{0{-}{>}{<}|\ x : \{\}\ |{>},$
$\quad 1{-}{>}{<}|\ x : \{\}\ |{>}\}$
anim > **execute updateR**
...*Execute schema* : *updateR*
Enter input arguments :
xin?− > **3**
s?− > **0**
###*Try branch* #1
...
###*Branch* #1 *succeed*.
Schema : *updateR*
$x : \{\}$
$x' : \{3\}$
$xin? : 3$
$db : \{0{-}{>}{<}|\ x : \{\}\ |{>},$
$\quad 1{-}{>}{<}|\ x : \{\}\ |{>}\}$
$db' : \{1{-}{>}{<}|\ x : \{\}\ |{>},$
$\quad 0{-}{>}{<}|\ x : \{3\}\ |{>}\}$
$s? : 0$

(continued)

anim > **execute updateR**
... *Execute schema* : *updateR*
Enter input arguments :
xin?− > **6**
s?− > **1**
Try branch #1
∗ ∗ ∗ *Entry guards* :
$(|\ [\ x\ ]\ |) = db\ \ s?$
−− > *False*
Branch #1 *fail*.
Execution of operation schema
updateR failed!
anim > **show −v R**
Schema : *R*
$x : \quad \{3\}$

This appears to be because the state of the schema *R* is being remembered by ZANS as is shown by using the ZANS *show* command to give the current state of the schema *R* above. The obvious solution is to hide the local state (that of *R*) as in the schema *updateR*1. Doing this causes all attempts to animate this operation to fail in the same manner as the second application of *updateR*. This happens because the θR predicate of the schema *rTos* cannot be satisfied *i.e.* ZANS cannot pick the appropriate binding from the set described by the schema *R*.

– There are semantically equivalent Z statements for which ZANS behaves differently, *e.g.* the statement $eval\ \varnothing \oplus \{(1, 2)\}$[7] gives

  Exception : *ZMT class error* @ *Override*().
  Run–time typing error. *Expecting Pair*.

  whereas $eval\ \varnothing \oplus \{(1 \mapsto 2)\}$ correctly returns $\{1 \mapsto 2\}$. Also, from the promotion example given above, the schema definition $addR \mathrel{\widehat=} (newR \land newS) \setminus (x')$ is evaluated as expected in ZANS whereas $addR \mathrel{\widehat=} \exists R' \bullet newR \land newS$ is not.

2.3 ZANS—Conclusions

Our conclusions are based on the sort of examples given above and some subjective impressions. The animator has a "try to execute at all costs" approach that unfortunately allows inconsistency in the specification. ZANS does not handle general constraint satisfaction or non-explicit operation schemas, in particular if an observation in the state is not constrained by an operation ZANS gives this observation a value of undefined whereas Z assumes the unconstrained label can take any value in its type, therefore picking a value from its type set may prove more fruitful. In larger specifications it is not always a trivial task to make schemas ZANS-explicit and it is commonplace to want to pick

[7] The word *eval* can be used as a command for evaluating expressions in ZANS

values from a set especially when using schemas as records through promotion. There are a lot of holes in the execution semantics implemented for Z. Semantically equivalent Z statements can cause different behaviour.

Some of the advantages of ZANS include: its speed of execution; the reordering of equalities algorithm appears to be sophisticated, for example deciding whether a predicate using equality ("=") should be animated as assignment (":=") or a guard, *i.e.* given the specification:

$$\begin{array}{|l}\hline S \\ x : \mathbb{P}\,\mathbb{Z} \\ y : \mathbb{P}\,\mathbb{Z} \\ \hline x = y \\ \hline\end{array} \qquad \begin{array}{|l}\hline init1 \\ S' \\ \hline x' = \varnothing \\ y' = \varnothing \\ \hline\end{array} \qquad \begin{array}{|l}\hline init2 \\ S' \\ \hline x' = \varnothing \\ \\ \hline\end{array}$$

the schema $init1$ causes the state predicate $x = y$ to be translated to an exit guard whereas $init2$ causes the same predicate to be translated to an assignment operator to obtain a value for y' as demonstrated in the following ZANS output.

$\ldots$ *Execute schema* : $init1$
Try branch #1
    $***$ *Statements* :
        $x' := \{\};$
        $y' := \{\};$
    $***$ *Exit guards* :
        $\mathbf{x}' = \mathbf{y}'$ *Note*!!
        $-- >$ *True*
Branch #1 *succeed*.
Schema : $init1$
$x' : \{\}$
$y' : \{\}$

(continued)

$\ldots$ *Execute schema* : $init2$
Try branch #1
    $***$ *Statements* :
        $x' := \{\};$
        $\mathbf{y}' := \mathbf{x}';$ *Note*!!
Branch #1 *succeed*.
Schema : $init2$
$x' : \{\}$
$y' : \{\}$

ZANS allows a specification to be loaded from more than one file, *e.g.* a promotion can be done in separate files and then loaded over the top of the file containing the local definitions; ZANS allows batch files to be run that specify an animation sequence, the assign command line argument allows the user to set values for sets, *etc.*, of the system. For example

anim > **assign Response** := {**success**, **noroom**}
{*success*, *noroom*}
anim > **pred success \in Response**
True

allows the aforementioned problem with free types to be solved by assigning the free type identifier to equal the set of alternatives described for that free type.

Considering ZANS as it is designed, *i.e.* ignoring development errors:

- the large amount of modification to a specification (which increases with the size and complexity of the specification) needed before ZANS can be used for animation makes it unusable on 'real-world' examples.
- the speed of execution of ZANS is refreshing and could be considered advantageous for validation of a specification during development. However, the simplified model of execution implemented by ZANS could cause the specifier to forsake eloquent description and abstraction for the purpose of writing a ZANS-executable specification.

3 ZETA/ZAP

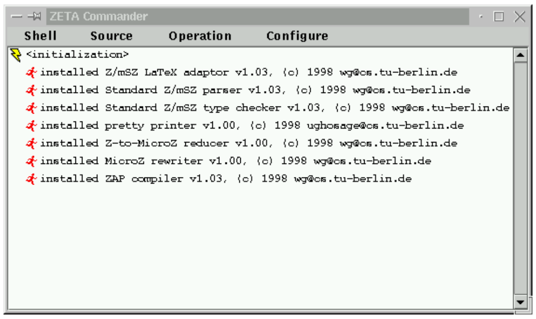
3.1 ZETA's strategy/philosophy

ZETA aims to be a framework for combining established modelling techniques with formal ones—*e.g.* State-charts and Z. The developers of ZETA see this as a way of providing an "incremental migration" of formal methods (FM) into industry. ZETA is implemented in Java using the Pizza superset to provide algebraic data types *etc.* and has a Java-based API.

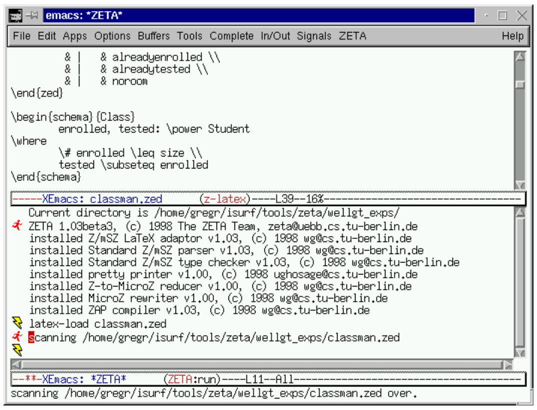
The integration of tools by ZETA happens on three levels:

- Data integration—the environment provides uniform data formats into and out of which different tools can map their own data—there is also a common data repository;
- Control integration—automatically controlled 'tool chains' which ensure, *e.g.*, that if an animator requires type-checking to be done on a part of the specification because it has changed, the type-checking tool is run;
- Presentation integration—attempts to standardise the interaction with the different tools by providing a common user interface as far as possible. In some cases this is not possible, *e.g.* Statemate has its own 'closed world', hence there are 'rough edges'.

There are two user interfaces for using ZETA, a graphical user interface (GUI) using the Java Swing libraries, figure 1(a), and an XEmacs based interface, figure 1(b).



(a) Java-GUI



(b) In XEmacs

Fig. 1. ZETA User Interfaces

ZAP—Z Animation program ZAP (Z animation program) is an animation tool for Z specifications that was developed to be integrated with ZETA. According to the author of ZAP the execution model implemented for Z is 'oriented towards higher-order functional languages', *i.e.* ZAP is best used on specifications that have a functional ('constructive') formulation. The model also includes:

- a transparent concept for sets, *i.e.* they can be described intensionally or extensionally;
- the ability to enumerate intensionally defined sets though this 'should be done sparingly' since ZAP has no techniques for dealing with them efficiently;
- a complete treatment of the schema calculus;

Efficiency in general has not been a big issue for the developers rather experimentation with the implemented execution model, for example ZAP spawns a concurrently executing Java thread for each predicate in an operation schema, i.e. concurrent unification of schema properties, while this is conceptually clean and convenient for implementors it is not very efficient *i.e.*evaluation of a recursive definition may create thousands of threads.

3.2 ZETA/ZAP—examples/problems

The ZETA environment implements the concept of tool-chains described under "Control Integration" above. It appears that ZETA checks the date stamp of the current files containing the Z-specification that is being examined to decide what tools need to be run on which files. This makes ZETA inefficient when making small modifications to a large Z-specification that is presented in one file. Between each modification to the specification ZETA will need to run the type-checker, code compiler, *etc.* before the specification can be animated again. While allowing multiple files to be used to present specifications in general a small modification still causes a non-negligible delay.

Another problem concerning ZAP is the number of additions needed to the specification before it can be animated. For example functions need to be defined for each operation to be animated to allow inputs to be given for the evaluation of operation schemas, *e.g.* a function for the enrol operation given in section 2.1 would be:

$$
\begin{array}{|l}
\mathit{enrol} == \lambda\, s : \mathit{Student} \bullet \\
\qquad\qquad\qquad (\exists\, s? == s \bullet \mathit{Enrol})
\end{array}
$$

Rather than maintaining a current state for the system being animated, ZAP appears to unfold the schema derived from the composition of all operation before and including the operation being animated. For example a typical input to ZAP for animating the *Test* operation of the class manager specification would be,

$$
\begin{array}{l}
\mathit{ClassInit}' \\
\quad \fatsemi\ \mathit{enrol}(\mathit{Steve}) \setminus (r!) \\
\quad \fatsemi\ \mathit{test}(\mathit{Steve}, 10)
\end{array}
$$

where $\mathit{ClassInit}'$ is the initialisation schema and *enrol* and *tested* are functions defined for their corresponding operations as described above. This means that the input to ZAP (a composition of a sequence of operations) must be compiled and then evaluated for each operation animated. When an error occurs in this sequence it is not obvious which operation was the cause of the error. For testing an operation in a sequence of n operations, we must insure that the animation of the first $n-1$ operations of the sequence provide the expected state.

It is possible to provide an alternative initialisation schema to initialise to a particular state to which the operation can be applied, hence replacing the initial sequence of operations, however this method allows the error of providing an initial state that can not be reached by any sequence of the operations specified. Also a separate initialisation schema would be needed for each operation being tested.

Evaluating an animation by unfolding the composition of a sequence of operation schemas means ZAP cannot prompt the user for inputs and uncomputable outputs of operations, which is considered by your author to be a good mode of interaction for animation.

Some semantically equivalent Z statements do not give the same result when animated. This appears to be because, as mentioned above, the execution model implemented in ZAP is 'oriented towards higher-order functional languages'. For example, consider the example where the class system specification has grades added as in Section 2.2, *i.e.* the state schema *Class* is:

$$
\begin{array}{|l}
\hline
\textit{Class} \\
\hline
\mathit{enrolled} : \mathbb{P}\, \mathit{Student} \\
\mathit{tested} : \mathit{Student} \pfun \mathbb{Z} \\
\hline
\operatorname{dom} \mathit{tested} \subseteq \mathit{enrolled} \\
\hline
\end{array}
$$

Given an operation to enquire about a students grade specified by the schema *Enquire* as,

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{Enquire}} \\
\hline
\Xi \mathit{Class} \\
s? : \mathit{Student} \\
r! : \mathit{Response} \\
g! : \mathbb{Z} \\
\hline
s? \in \operatorname{dom} \mathit{tested} \\
r! = \mathit{alreadytested} \\
g! = \mathit{tested}\ s? \\
\hline
\end{array}
$$

Also a semantically equivalent alternative for this operation *Enquire*2,

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{Enquire2}} \\
\hline
\Xi \mathit{Class} \\
s? : \mathit{Student} \\
r! : \mathit{Response} \\
g! : \mathbb{Z} \\
\hline
s? \in \operatorname{dom} \mathit{tested} \\
r! = \mathit{alreadytested} \\
(s?, g!) \in \mathit{tested} \\
\hline
\end{array}
$$

Using ZAP to animate these operations provides the expected result for the operation described by the schema *Enquire*:

$\mathit{ClassInit}'$
$\quad \fatsemi\ \mathit{enrol}(\mathit{Sally}) \setminus (r!)$
$\quad \fatsemi\ \mathit{test}(\mathit{Sally}, 10) \setminus (r!)$
$\quad \fatsemi\ \mathit{enquire}(\mathit{Sally})$

```
⟶ {<enrolled′ == {Sally},g! == 10,r! == alreadytested,
       tested′ == {(Sally,10)}>}
```

However, trying to animate the operation schema *Enquire*2 fails.

$\mathit{ClassInit}'$
$\quad \fatsemi\ \mathit{enrol}(\mathit{Sally}) \setminus (r!)$
$\quad \fatsemi\ \mathit{test}(\mathit{Sally}, 10) \setminus (r!)$
$\quad \fatsemi\ \mathit{enrol4}(\mathit{Sally})$

```
⟶ ERROR[LTX:classtest.zed(60.5-63.29)]:
    execution failed
     reason:
      unresolvable constraint in value of enumeration:
       value: <enrolled′ == _,g! == _,r! == _,tested′ == _>
       constraint: LTX:classtest.zed(60.5-63.29)
     backtrace:
      at evaluating command input
```

It is reasonably obvious in this case that the predicate $(s?, g!) \in \mathit{tested}$ from the schema *Enquire*2 can be transformed (preserving semantics) into $g! = \mathit{tested}\ s?$, which allows ZAP to evaluate a result. However, the output from ZAP does not help in locating this error.

The text $[\mathit{LTX} : \mathit{classtest.zed}(60.5 - 63.29)]$ from the ZAP output is a link that can be clicked on to identify where in the specification the error was caused. Unfortunately in this, and several other cases, the link is pointing to the input string of composed operators. Another problem with this output is that the reason given for failure is

```
—→ unresolvable constraint in value of enumeration:
        value: <enrolled' == _,g! == _,r! == _,tested' == _>
```

which does not identify $g!$ as being the uncomputable observation.

The algorithms used by ZAP to convert a Z specification into Java code contains an error that discards some simple equality predicates from schemas. For instance take the *Enrolok* operation from the modified class manager specification,

$$
\begin{array}{|l}
\textit{Enrolok} \\
\hline
\Delta Class \\
s? : Student \\
r! : Response \\
\hline
s? \notin enrolled \\
\#enrolled < size \\
enrolled' = enrolled \\
tested' = tested \cup \\
r! = success \\
\hline
\end{array}
$$

The predicate $enrolled' = enrolled$ in this operation schema is discarded by ZAP during translation, leaving an unresolvable value for $enrolled'$. This happens, we believe, because somewhere in the process of developing code an equation with, what can be seen as, variables on either side will 'unify away'. Since the equation can be made true by substituting either for the other. This can be avoided by changing this predicate to a non-trivial expression such as $enrolled' = enrolled \cup \varnothing$. This has been identified as an error by the author of ZAP and is going to be fixed.

3.3 Conclusions—ZETA/ZAP

Some of the considered advantages of ZETA/ZAP include:

- Z sections and the related idea of refinements;
- nice XEmacs user interface features
- good feedback as computation proceeds
- concept of tool chains;
- compiling to Java means that there is the opportunity of interfacing code to a GUI;
- can lift the interface by functional programming in Z to allow test sequences;
- philosophy behind framework compelling

To expand these points ZETA/ZAP allows a specification to contain Z sections which can be used to divide a specification into logical units in one or many files . One example of this is that a specification written for use with ZETA/ZAP can have a specification section and an execution section that refines the specification to make it more animatable. This is a particularly nice feature of ZETA that allows, for example, given sets in the specification section to be overridden by free-types in the execution section.

The ideas behind ZETA's XEmacs user interface are good. In particular expandable list items allow ZETA's details of an operation to be expanded or hidden by clicking on an icon; the hyper-link style indexes that allow the user to be directed to appropriate places in the specification, though the calculation of the relevant point in the specification needs improving in ZAP.

There is little user documentation for ZETA and ZAP and it is difficult to distinguish exactly the boundaries between what ZETA is responsible for and what ZAP is responsible for. This tool is being actively developed.

4 Possum

4.1 Strategy/philosophy

The design goal and mode of evaluation adopted by the developers of Possum was to build a system that would work on a collection of *existing* specifications. Possum was designed to animate the SUM specification language [3] but can also be used to animate Z.

Possum was originally implemented in Qu-Prolog and then later using Mercury. It has a GUI built in Tcl/TK, figure 4.1.

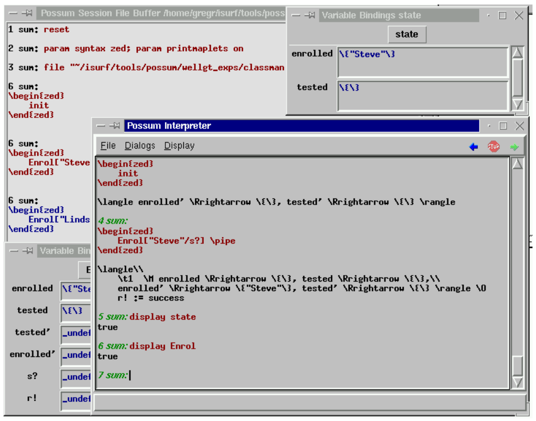


Fig. 2. The Possum Graphical User Interface

The algorithm used for evaluation of specified operations in Possum simplifies predicates into subgoals that can be categorised as 'chests' or 'checkers' as described below. The algorithm then attempts to order sub-goals in the evaluation process based on projected chest sizes.

- Chests are predicates that can be used to generate values for variables, *e.g.* for x, in predicates like $x = 1$ or $0 \leq x \leq 10$;
- Checkers are predicates which are used to decide whether or not a value of a variable meets a condition.
- *e.g.* for the set comprehension $\{d : 0 \mathrel{..} 5000 \mid d = 4\}$ — $d = 4$ would be used as a chest and then $d : 0 \mathrel{..} 5000$ used as a checker—if we used $d : 0 \mathrel{..} 5000$ as a chest and $d = 4$ as a checker, we would have 5001 numbers generated to check;

If a predicate is to generate a binding for a label there must be at least one chest for that label. Some predicates contain no chests and therefore cannot be animated by Possum. Whether a predicate is a chest or not has been decided by the Possum implementors, and their choices were made on the grounds of whether a predicate can in principle be a chest and, if so whether it is computationally feasible for it to be one. It may be ruled out, for example, if it would take too long to compute.

4.2 Possum—example/problems

The state schema of a specification must be called "state" for Possum to carry out animation. For instance take the class manager specification from section 2.1 again. If we attempt to animate this specification with Possum the *ClassInit* operation schema can be evaluated, however following this by the *Enrol* operation results in 'solution unknown', *i.e.* Possum is unable to evaluate the effect *Enrol* has on the state. If we then change the name of the state schema *Class* to *state* Possum is able to animate the other operations.

Possum does not accept all of Z as described by Spivey [4]. This is not surprising given the animator was written for the SUM specification language and supporting Z is secondary to its purpose.

For example, axiomatic definitions evaluate to false if they have no predicate parts. Also the schema operator θ, used to select a binding from the set of bindings given by a schema, is not handled by Possum as expected. Further, set union is permitted between sets of different type, *e.g.* the state schema *Class* from the class manager specification could be defined as follows,

$$
\begin{array}{|l}
\hline
\textit{state} \\
\textit{enrolled}, \textit{tested} : \mathbb{P}\,\textit{Student} \cup \mathbb{N} \\
\hline
\#\textit{enrolled} \leq \textit{size} \\
\textit{tested} \subseteq \textit{enrolled} \\
\hline
\end{array}
$$

which allows students or integers to be 'enrolled'.

Possum does not appear to treat schemas as sets of bindings. Like ZANS, Possum keeps a record of the current values for the state throughout animation. This does not generalise well, particularly for promotion because there are generally several possibilities for the value of the local state being promoted. If we hide the local state, *i.e.* try to recreate the binding each time it is used, Possum fails to find a solution.

When attempting to animate Z specifications the order of disjuncts seems to defeat the algorithm that selects the best order to simplify/evaluate predicates. This can be demonstrated by using the class manager specification with two alternative operations specified for indicating a student has been tested, *Test* and *Test*1:

$$
\begin{array}{|l}
\hline
\textit{Testok} \\
\Delta \textit{state} \\
s? : \textit{Student} \\
r! : \textit{Response} \\
\hline
s? \in \textit{enrolled} \\
s? \notin \textit{tested} \\
\textit{tested}' = \textit{tested} \cup \{s?\} \\
\textit{enrolled}' = \textit{enrolled} \\
r! = \textit{success} \\
\hline
\end{array}
\quad
\begin{array}{|l}
\hline
\textit{AlreadyTested} \\
\Xi \textit{state} \\
s? : \textit{Student} \\
r! : \textit{Response} \\
\hline
s? \in \textit{tested} \\
r! = \textit{alreadytested} \\
\hline
\end{array}
\quad
\begin{array}{|l}
\hline
\textit{NotEnrolled} \\
\Xi \textit{state} \\
s? : \textit{Student} \\
r! : \textit{Response} \\
\hline
s? \notin \textit{enrolled} \\
r! = \textit{notenrolled} \\
\hline
\end{array}
$$

$$\textit{Test} \mathrel{\widehat=} \textit{NotEnrolled} \lor \textit{AlreadyTested} \lor \textit{Testok}$$

$$\textit{Test}1 \mathrel{\widehat=} \textit{Testok} \lor \textit{AlreadyTested} \lor \textit{NotEnrolled}$$

When *Test* is tried by animating the operations *ClassInit*, *Enrol* and *Test*, Possum returns a result reasonably quickly.

$$
\begin{array}{ll}
3\ \textit{sum}: & \textit{ClassInit} \\
4\ \textit{sum}: & \textit{Enrol}[\text{"}\textit{Joe}\text{"}/s?] \gg \\
5\ \textit{sum}: & \textit{Test}[\text{"}\textit{Steve}\text{"}/s?] \gg
\end{array}
$$

$$
\begin{array}{ll}
\longrightarrow & \langle \textit{enrolled} \Rrightarrow \{\text{"}\textit{Joe}\text{"}\}, \textit{tested} \Rrightarrow \{\}, \textit{enrolled}' \Rrightarrow \{\text{"}\textit{Joe}\text{"}\}, \textit{tested}' \Rrightarrow \{\}\rangle \\
& r! := \textit{notenrolled}
\end{array}
$$

However, if this process is repeated using the operation schema *Test*1 Possum does not return a result.

When an operation that cannot be evaluated or has a large search space is being evaluated by Possum there is no progress feedback. This makes it difficult to know whether the evaluation will take five minutes or will never evaluate.

Possum has two outputs for un-animatable operations, 'no solution' and 'solution unknown'. The first means there is no possible solution to the operation given the specifications constraints. The second means Possum cannot evaluate any solutions though there may be some. These error messages are the only given and there is no indication of why the operation could not be animated.

Therefore the user is left to inspect the specification and try to guess why, which clearly makes Possum less valuable for verifying the correctness of specifications.

4.3 Conclusions—Possum

Possum has several features we consider advantageous. These include:

- A well thought out GUI,
- The ability to create, record, save and re-run scripts,
- Tcl/Tk graphical visualisations of system.

Possum's GUI provides a main interpreter window in which interactive commands can be entered and a script window that is used to open previously saved or created scripts. The script commands can be sent to the interpreter as a whole script or one command at a time. There is a parameter window to modify the variable behaviours of Possum. Also the maximum integer that Possum will use in its evaluation can be set here.

The user can open a window representing each schema that is currently loaded in the interpreter. These windows have a field for each label in the schema. When a window is in focus there are key combinations defined for the available operations.

Possum allows Tcl/Tk graphical visualisations of the specified system to be controlled by Possum during animation. This increases the utility of Possum as a tool to validate a specification against informal requirements by demonstration of the specified system's behaviour. The visualisations were not explored, although some examples distributed with Possum appear to work well.

One of the major criticisms of Possum is that it has no documentation to speak of. This means learning to use the tool through experimentation.

5 Conclusions

5.1 Other work

Breuer and Bowen's paper (see [1]) talks about some more formal characteristics of animation techniques. These are:

- correctness—giving only correct answers, partial or complete.
- coverage—the portion of the Z grammar handled by the animator;
- efficiency—the speed at which the animator can evaluate results;
- sophistication—the ability of the animator to terminate

In particular they are concerned with a trend for animators to forsake correctness for the other three categories mentioned above, whereas these issues should be orthogonal (considered as well as) correctness. Breuer and Bowen also give one possible classification for animation techniques by means of their treatment of sets:

a) sets must be finite and are modelled by finite arrays;
b) sets may be countably infinite and are modelled by an enumeration algorithm;
c) sets are cardinally unbounded and modelled by their characteristic function.

A comparison of each of these is given.

5.2 Some properties of a good animator

From evaluating the three animators discussed in this paper, we consider the following list to be desirable properties for a usable Z animation system.

- A Z animator must preserves the semantics of Z;
- should have dedicated human computer interaction techniques applied for user interface design;

- supply the ability to refine (make more concrete for the purpose of executability) a specification in well defined, distinguishable sections of the specification document.
- provide good feedback as computation proceeds, clearly document partial results including reasons for their partiality.
- be free in terms of the GNU Project [8] definition of free;
- be distributed with good user documentation, not containing only a dedicated example;
- support from developers—especially if it is an experimental system;
- have the ability to connect to graphical visualisations of the specified system to allow better validation of specification with non-Z users.

5.3 Final conclusions

This paper is rather uneven since tools have varying amounts written about them. Bad or non-existent documentation means that discovering what the tools can and cannot do was done by experimentation. This is an inefficient way of evaluation that is slow and error prone. Therefore the results presented here are qualitative, *i.e.* subjective, rather than quantitative, *i.e.* a corpus of specs and measurements under well defined categories.

The examples in this paper are taken from a specification of a trivial system. The problems get worse as the problem being specified increases in size and complexity.

Some of the conclusions from the experimentation presented. It is such hard work getting specifications into an animatable form that the verification of correctness of the specification obtained by the animation itself is almost negligible *i.e.* getting ready for animation subsumes inspection! More seriously, since specifications need adapting for animation there are issue of proving these changes preserve meaning, which would ask efficiency questions of the animation process. We *might* present adaptations as alternative formalisations of informal requirements. This means there is proof obligations to show the more abstract original specification is not more or less constraining.

References

1. 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.
2. I. Hayes, editor. *Specification Case Studies*. Prentice-Hall, 1987.
3. D. Hazel, P. Stooper, and O Traynor. An Animator for the SUM Specification Language. Technical report, Software Verification Research Centre, School of Information Technology, University of Queensland, Austrailia, 1997.
4. J. M. Spivey. *The Z notation: A reference manual*. Prentice Hall, 2nd. edition, 1992.
5. J. Wordsworth. *Software Development with Z*. Addison-Wesley, 1992.

[8] URL: `http://www.gnu.org`