An Approach to Animating Z Specifications

Xiaoping Jia
Division of Software Engineering
School of Computer Science, Telecommunication, and and Information Systems
DePaul University
Chicago, IL 60604
E-mail: jia@cs.depaul.edu

Abstract

This paper presents the ZANS approach to animating a large subset of Z specifications. Unlike most other animation approaches that are based on functional or logical programming paradigms, the approach presented here attempts to transform the specifications into an imperative intermediate language, which can also serve as the basis of code synthesis from Z specifications. ZANS is showing promise of being an effective tool for requirements validation and assisting students learning Z specification languages.

1 Introduction

The recent development in model-oriented formal specification methods, particularly Z[1, 2], and their applications in industrial software development projects[3, 4] have demonstrated the practicality and the potential benefits of formal methods. However major obstacles in applying formal methods still remain. One of such obstacles is that the rigor of mathematical notations used in formal specifications makes them much more precise than natural language based specifications, but also less intuitive and much more difficult to understand by non-professionals. This makes the validation of formal specifications a very difficult task. A remedy for that is to animate or execute the formal specifications to demonstrate the behavior of the system being specified. The advantages of executable, or operational, specifications are well-known[5]. However most of the executable specification languages are overly restrictive, and the specifications usually contain design considerations[6]. Model-oriented formal specifications languages, such as VDM and Z, gain their popularity largely due to their generality, flexibility, and the ability to separate design considerations from specifications. These characteristics also render such specification languages non-executable in general.

This paper presents an experimental tool for animating Z specifications. It is capable of animating a large and useful subset of Z specifications. The goals of the ZANS tool are:

- Facilitate validation of Z specifications;
- Experiment design refinement and code synthesis based on Z specifications; and
- Assist students to learn the Z notation.

Currently, it supports the following features:

- type checking of Z specifications;
- expansion of schema expressions;
- evaluation of expressions and predicates;
- execution of operation schemas.

The front end of ZANS is compatible with the Z type checker ZTC [7]. It accepts input in LaTeX with `zed` or `oz` or ZSL. A Z specification prepared for ZTC can be animated by ZANS with little or no modification.

For the remainder of this paper, familiarity with Z is assumed. In section 2, *explicit operation schemas*, the key concept, is introduced. A determination and translation algorithm is given in section 3, followed by discussions in section 4.

2 Explicit Operation Schemas

The key to our animation approach is the classification of operation schemas in Z into two categories: *explicit* and *non-explicit* operation schemas. Informally,

- an explicit schema is one in which all the output variables are directly or indirectly *defined* by the input variables;
- whereas, an implicit schema is one in which some of the output variables are *constrained* by the input variables.

Explicit schemas can be made executable. While non-explicit schemas may need elaborate algorithms and/or data structures, or inefficient constraints satisfying mechanisms to compute the values of the output variables from the input variables. A study of the Z specifications collected in *Specification Case Studies* edited by Ian Hayes[10] shows that an overwhelming majority (94%) of the operation schemas are explicit, or can be made explicit with minor modifications.

More precisely, an operation schema is *explicit* if the values of all of its output variables, including the post state, can be determined by evaluating some of the expressions in the schema in some sequential order and each expression is evaluated no more than once. Consider following two schemas S_1 and S_2:

S_1

$$x?, y?, q!, r! : \mathbb{N}$$

$$q! = x? \text{ div } y?$$
$$r! = x? - q! * y?$$

S_2

$$x?, y?, q!, r! : \mathbb{N}$$

$$x? = q! * y? + r!$$
$$r! < y?$$

Both S_1 and S_2 specify that $q!$ and $r!$ are the quotient and remainder of $x?$ divided by $y?$. S_1 is explicit, because the values of the output variables $q!$ and $r!$ can be determined by evaluating the expressions in S_1 in the following order:

1. $q! := x? \text{ div } y?$
2. $r! := x? - q! * y?$

The two equations are considered definitions, so the equal signs are replaced by the assignment operators. On the other hand, S_2 is non-explicit, since $q!$ and $r!$ are specified with constraints they must satisfy. In general, it is not straightforward to derive an algorithm to calculate the values of the output variables based on the constraints given.

To determine whether a schema is explicit is not trivial. We will present a determination algorithm in the next section, but first, some definitions are in order.

A *simple name* is a variable name, possibly decorated with ', !, or ?, such as *name*, *name'*, *name*!, or *name*?. The undashed names and the names with ?-suffix are called *pre-names*. They refer to the input to the operation and the variables in the pre-state. The dashed names and the names with !-suffix are called *post-names*. They refer to the output of the operation and the variables in the post-state.

Definition 1 *Simple predicate.* A predicate is called a simple predicate if it is in one of the the following forms:

a) E_1 *inrel* E_2, or

b) *prerel* E_1.

where E_1 and E_2 are expressions, and *inrel* is an infix relational operator, such as $=, \in, \subseteq$, and *prerel* is a prefix relational operator, such as partition.

Our approach focuses on *definitive* simple predicates, which are simple predicates involving equalities. The goal is to convert some of the definitive predicates to assignments and order these assignments so that the values of all the post-names can be determined.

Definition 2 *Simple definition.* A simple predicate is called a simple definition, if it is in the following form:

$$v = E$$

where v is a simple post-name and E is an expression that is not a simple post-name. We say that v is defined by E.

Let Var[E] denote the set of free variable names in E. If all the variables in Var[E] are defined, then expression E can be evaluated, and the value of v can be determined, i.e., the simple definition above can be converted to an assignment $v := E$.

Definition 3 *Reversible definition.* A simple predicate is a reversible definition, if it is in the following form:

$$v = u$$

where v and u are both simple post-names.

If either v or u are defined, then the value of the other name can also be easily determined, i.e., the reversible definition above can be converted to either $v := u$ or $u := v$.

Simple definitions and reversible definitions are called *definitive* predicates. Conversion of non-definitive predicates to assignments is non-trivial in general. Currently, we do not attempt to convert non-definitive predicates.

Definition 4 *Assignment conversion set.* Let p be a simple predicate, the assignment conversion set of p, denoted $\mathcal{A}[p]$, is defined as follows:

a) For a simple definition $v = E$, $\mathcal{A}[v = E]$ is $\{v := E\}$;

b) For a reversible definition $v = u$, $\mathcal{A}[v = u]$ is $\{v := u, u := v\}$;

c) For a non-definitive predicate p, $\mathcal{A}[p] = \varnothing$.

Definition 5 *Well-ordered assignment sequence.* Let N_0 be the set of pre-names. A assignment sequence $A = \langle v_1 := E_1, v_2 := E_2, \ldots, v_n := E_n \rangle$ is well-ordered with respect to N_0, if and only if

a) For every $i, j, 1 \leq i, j \leq n$, if $i \neq j$ then $v_i \neq v_j$; and

b) For every $i, 1 \leq i \leq n$,
$\text{Var}[E_i] \subseteq \{v_1, v_2, \ldots, v_{i-1}\} \cup N_0$.

In other words, each name is assigned only once, and if the assignments are executed in the given order, the variable names in E_i are either pre-names or have already been assigned values. If the values of the pre-names are given, then the execution of the assignment sequence will determine the values of $v_1, v_2, \ldots, v_n$. We use Def[A] to denote the set of names defined in A, i.e., $\text{Def}[A] = \{v_1, v_2, \ldots, v_n\}$.

Definition 6 *Sub-permutation.* Let $\langle i_1, i_2, \ldots, i_m \rangle$ be an integer sequence. For some n, $n \geq m$, we say $\langle i_1, i_2, \ldots, i_m \rangle$ is a sub-permutation of $1 \,..\, n$, if $\langle i_1, i_2, \ldots, i_m \rangle$ is a permutation of a subsequence of $\langle 1, 2, \ldots, n \rangle$.

Definition 7 *Explicit simple predicates set.* Let $Pred$ be a set of simple predicates $p_1, p_2, \ldots, p_n$. $Pred$ is *explicit* with respect to pre-name set $PreNames$ and post-name set $PostNames$, if there exists a assignment sequence $A = \langle a_{i_1}, a_{i_2}, \ldots, a_{i_m} \rangle$ such that

a) A is well-ordered with respect to $PreNames$; and

b) $\text{Def}[A] = PostNames$; and

c) $a_{i_k} \in \mathcal{A}[p_{i_k}]$, for $1 \leq k \leq m$; and

d) $i_1, i_2, \ldots, i_m$ is a sub-permutation of $1 \,..\, n$.

A is called the *assignment conversion sequence* of $Pred$ with respect to $PreName$, denoted $\mathcal{A}[Pred, PreName]$.

If $Pred$ is explicit, the values of the post-names can be determined by executing the assignment sequence A. Furthermore, predicates $p_{i_1}, p_{i_2}, \ldots, p_{i_m}$ will hold after the execution. We say that predicates $p_{i_1}, p_{i_2}, \ldots, p_{i_m}$ are *covered* A. If predicate p is not covered by assignment, then it is called an *entry guard* if it contains pre-names only, otherwise it is called an *exit guard*. Therefore, $Pred$ can be partitioned into three disjoint sets:

- $\mathcal{P}_{assign}[Pred, PreName]$: the set of predicates covered by A;
- $\mathcal{P}_{entry}[Pred, PreName]$: the set of entry guards; and
- $\mathcal{P}_{exit}[Pred, PreName]$: the set of exit guards.

Definition 8 *Conjunctive schema.* An operation schema is conjunctive, if the axiom part of the schema is a sequence of simple predicates, which is interpreted as a conjunction of these simple predicates.

Let O be a conjunctive operation schema in the following form:

$O \mathrel{\widehat{=}} [d_1;\ d_2;\ \ldots;\ d_k \mid p_1;\ p_2;\ \ldots;\ p_n]$

where $d_1, d_2, \ldots, d_k$ are declarations and $p_1, p_2, \ldots, p_n$ are simple predicates. We use Pre[O] and Post[O] to denote the sets of pre-names and post-names of O respectively.

Definition 9 *Explicit conjunctive schema.* Conjunctive operation schema O is explicit if $\{p_1, p_2, \ldots, p_n\}$ is explicit with respect to Pre[O] and Post[O].

An explicit conjunctive operation schema O can be executed as described below, assuming that the values of all its pre-names are given:

Step 1. **if** any predicate in $\mathcal{P}_{entry}$ is false
then execution fails.

Step 2. Execute the assignment sequence.

Step 3. **if** any predicate in $\mathcal{P}_{exit}$ is false
then execution fails.
else execution succeeds.

A successful execution will yield a binding of values to all post-names such that all $p_1, p_2, \ldots, p_n$ will hold. A failure of execution indicates that there does not exist a binding of the post-names that will satisfy all the predicates, which indicates that the operation schema specifies a partial operation.

For a non-conjunctive schema O, we can convert its axiom part to the disjunctive normal form, $P_1 \vee P_2 \vee \cdots \vee P_k$, where $P_i, i = 1, 2, \ldots, k$ are conjunctions of simple predicates.

Definition 10 *Explicit non-conjunctive schema.* A non-conjunctive operation schema O is explicit, if all $P_i, i = 1, 2, \ldots k$, are explicit with respect to Pre[O] and Post[O].

If all P_i are explicit, than each P_i corresponds to an *branch*, which consists of a set of entry guards, an assignment sequence, and a set exit guards as before. The execution of O succeeds when one of its branches succeeds, it fails when all of its branches fail. It's easy to see that a successful execution will yield a binding of values to all post-names such that $P_1 \vee P_2 \vee \cdots \vee P_k$ will hold.

Boolean EX(**in** $Pred, PreNames, PostNames$,
out $EntryGuard, AssignSeq, ExitGuard$)

1. /* Initialization */
   $EntryGuard := \{p : Pred \bullet p$ does involve post-names $\}$;
   $Definition := \{p : Pred \bullet p$ is definitive $\}$;
   $ExitGuard := Pred \setminus (EntryGuard \cup Definition)$;
   $AssignSeq := \langle\rangle$;
   $DefNames := \varnothing$;
   $done :=$ **false**;
2. /* Ordering definition */
   while $\neg done$ **do**
   $progress :=$ **false**;
   for each definition $d \in Definition$ **do**
   case d **is**
   when a simple definition: $v = E$ **=>**
   if $\mathrm{Var}[E] \subseteq DefNames \cup PreNames$ **then**
   $Definition := Definition \setminus \{d\}$;
   $DefNames := DefNames \cup \{v\}$;
   $AssignSeq := AssignSeq^\frown \langle v := E\rangle$;
   $progress :=$ **true**;
   end if;
   when a reversible definition: $v = u$ **=>**
   if $u \in DefNames$ **then**
   $Definition := Definition \setminus \{d\}$;
   $DefNames := DefNames \cup \{v\}$;
   $AssignSeq := AssignSeq^\frown \langle v := u\rangle$;
   $progress :=$ **true**;
   elsif $v \in DefNames$ **then**
   $Definition := Definition \setminus \{d\}$;
   $DefNames := DefNames \cup \{u\}$;
   $AssignSeq := AssignSeq^\frown \langle u := v\rangle$;
   $progress :=$ **true**;
   end if;
   end case;
   end for;
   $done := \neg progress \vee Definition = \varnothing \vee$
   $DefNames = PostNames$;
   end while;
   $ExitGuard := ExitGuard \cup Definition$;
3. /* Check whether every post-name is defined */
   if $DefNames = PostNames$ **then**
   return true; /* $Pred$ is explicit */
   else
   return false; /* $Pred$ is non-explicit; */

Figure 1: Algorithm **EX**.

3 Determination Algorithm

In this section, we will present an algorithm determining whether an operation schema is explicit, and translating it an executable intermediate representation when it is explicit.

3.1 The Algorithm

Algorithm **EX** determines if a set of simple predicates is explicit with respect to given sets of pre-names and post-names. Specifically, the input to algorithm **EX** is the following:

- a set of simple predicates $Pred = \{p_1, p_2, \ldots, p_n\}$;
- a set of pre-names $PreNames$;
- a set of post-names $PostNames$.

If $Pred$ is explicit with respect to $PreNames$ and $PostNames$, **EX** returns **true** and generate the following output:

- a set of entry guards $EntryGuard$;
- an assignment sequence $AssignSeq$;
- a set of exit guards $EntryGuard$.

If $Pred$ is not explicit, **EX** returns **false**.

Algorithm **EX** is shown in Figure 1. When $Pred$ is explicit, upon termination **EX** ensures that:

$$AssignSeq = \mathcal{A}[Pred, PreNames\ ] \wedge$$
$$EntryGuard = \mathcal{P}_{entry}[Pred, PreNames\ ] \wedge$$
$$ExitGuard = \mathcal{P}_{exit}[Pred, PreNames\ ].$$

The invariant of the outer loop, i.e., the **while** loop, is the following:

($AssignSeq$ is well-ordered with respect to $PreNames$) $\wedge$
$\mathrm{Def}[AssignSeq] = DefNames$

The time complexity of **EX** is $O(n^2m^2)$, where $n = \#Pred$, i.e., the number of predicates, and $m = \#(PreNames \cup PostNames)$, i.e., the number of names used in the predicates.

3.2 Extended Guarded Commands

We introduce the *extended guarded command* (EGC) as an intermediate representation for explicit operation schemas. The extended guarded command is an extension to Dijkstra's guarded command. In general, an extended guarded command is in the following form:

egc $[d_1, d_2, \ldots, d_l]$
$g_1 \Longrightarrow s_{1,1};\ s_{1,2};\ \ldots s_{1,n_1} \Longleftarrow h_1$
$[]\quad g_2 \Longrightarrow s_{2,1};\ s_{2,2};\ \ldots s_{2,n_2} \Longleftarrow h_2$
$\ldots$
$[]\quad g_k \Longrightarrow s_{k,1};\ s_{k,2};\ \ldots s_{k,n_k} \Longleftarrow h_k$
cge

where d_i are declarations, g_i are entry guards, h_i are exit guards, and $s_{i,j}$ are statements.

Informally, the semantics of the extended guarded command above is defined as follows:

- Declarations $d_1, d_2, \ldots, d_l$ introduce local variable names, whose scope extends to the end of the extended guarded command, and the usual rules for name clash apply.
- Execution of branch i is successful if
  a) the entry guard g_i evaluates to true;

b) all the statements $s_{i,1}, s_{i,2}, \ldots, s_{i,n_i}$ execute successfully; and

c) the exit guard h_i evaluates to true.

- If one or more branches can be executed successfully, then one of these branches is chosen and executed non-deterministically, and the execution of the extended guarded command is successful.

- If none of the branches can be executed successfully, the execution of the extended guarded command fails.

When all the exit guards are **true**, the extended guarded command reduces to the original guarded command.

For an operation schema O, its explicitness can be determined and it can be translated to an EGC when it is explicit as follows:

Step 1. Expand schema O to handle schema inclusions and schema operations such as conjunction and disjunction. Let *Decl* and *Pred* be the declaration and axiom part of the resulting schema, and Pre[O] and Post[O] be the sets of pre-names and post-names, respectively.

Step 2. Convert *Pred* to the disjunctive normal form. Let $Pred \equiv P_1 \lor P_2 \lor \cdots \lor P_k$, where $P_1, P_2, \ldots, P_k$ are conjunctions of simple predicates.

Step 3. For $i = 1, 2, \ldots k$, invoke
EX(P_i, Pre[O], Post[O], $EntryGuard_i$, $AssignSeq_i$, $ExitGuard_i$).
Schema O is explicit when every P_i is explicit with respect to Pre[O].

Step 4. If O is explicit, it can be translated to the following extended guarded command:

$[Student]$

$Response ::= success \mid alreadyenrolled \mid \cdots$

Class

$enrolled, tested : \mathbb{P}\, Student$

$tested \subseteq enrolled$

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

Enrolok

$\Delta Class$
$s? : Student$
$r! : Response$

$s? \notin enrolled$
$enrolled' = enrolled \cup \{s?\}$
$tested' = tested$
$r! = success$

AlreadyEnrolled

$\Xi Class$
$s? : Student$
$r! : Response$

$s? \in enrolled$
$r! = alreadyenrolled$

$Enrol == Enrolok \lor AlreadyEnrolled$

Figure 2: Specification of Class Manager's Assistant

egc [*Decl*]
$EntryGuard_1 \Longrightarrow AssignSeq_1 \Longleftarrow ExitGuard_1$
[] $EntryGuard_2 \Longrightarrow AssignSeq_2 \Longleftarrow ExitGuard_2$
$\cdots$
[] $EntryGuard_k \Longrightarrow AssignSeq_k \Longleftarrow ExitGuard_k$
cge

An example is given below to illustrate the process. The example is extracted from the class manager's assistant example in [2], only the portion involving the *Enrol* operation is discussed here. The specification is shown in Figure 2. The specification involves schema inclusion and schema disjunction. The operation schemas are explicit. Figure 3 shows the extended guarded commands for *ClassInit* and *Enrol*.

operation *ClassInit*
egc $[enrolled', tested' : \mathbb{P}\, Student]$
  true $\Longrightarrow$
    $enrolled' := \varnothing;\ tested' := \varnothing;$
  $\Longleftarrow tested' \subseteq enrolled'$
cge

operation *Enrol*
egc $[enrolled, tested, enrolled', tested' : \mathbb{P}\, Student;$
  $s? : Student;\ r! : Response]$
  $s? \notin enrolled;\ tested \subseteq enrolled \Longrightarrow$
    $enrolled' := enrolled \cup \{s?\};\ tested' := tested;$
    $r! := success;$
  $\Longleftarrow tested' \subseteq enrolled'$
[] $s? \in enrolled;\ tested \subseteq enrolled \Longrightarrow$
    $enrolled' := enrolled;\ tested' := tested;$
    $r! := alreadyenrolled;$
  $\Longleftarrow tested' \subseteq enrolled'$
cge

Figure 3: EGC of Class manager's Assistant

4 Discussion

The generated EGC can be used to directly animate operation O, or serve as the basis for code synthesis from Z specification. The expressions in EGC involve mathematical objects defined in Z, such as sets, relations, and functions. They are not supported by most of the programming languages. A library handling these mathematical objects must be supplied in order to execute EGC. ZANS (Z ANimation System) is an experimental tool developed based on the approach presented above. It is developed using C++ and contains a C++ class library that handles all the mathematical objects and their operations defined in Z. To ensure termination of the execution, currently ZANS does not allow infinite sets. The integer sets $\mathbb{Z}, \mathbb{N}$ are implemented as finite sets $-N\,..\,N$ and $0\,..\,N$ respectively, where N is a positive integer. A number of fairly large specifications of real systems have been successfuly animated by ZANS.

Completely excluding infinite sets, sometimes is too restrictive. Many operations involving infinite sets also guarantee termination. We are currently investigating techniques such as symbolic and lazy evaluation, to incorporate infinite sets while maintaining termination of execution.

A limitation of ZANS is that it does not support constraint satisfaction for non-explicit operation schemas. We are exploring mechanisms to animate some of the non-explicit schemas:

- Extend reversible definitions to include all simple predicates in the form of $x = f(y)$ where both f and f^{-1} can be calculated. In this case, $\mathcal{A}[x = f(y)] = \{x := f(y), y = f^{-1}(x)\}$. Furthermore, there are situations that multiple-variable constraints are convertible to assignments, such as $\langle X, Y\rangle$ partition Z. Its assignment conversion set $\mathcal{A}[\langle X, Y\rangle \text{ partition } Z]$ can be $\{X := Z \setminus Y, Y := Z \setminus X, Z := X \cup Y\}$.

- Use an extensive library of generic algorithms to refine the non-explicit operation schemas.

EGC generated from explicit operation schemas can also serve as the basis for synthesizing efficient code from Z specifications.

The ZANS approach is different from the other animation approaches in the following aspects:

a) Most of the other animation approaches of Z are indirect. They translate Z specifications to programs in logical or functional programming languages, such as ML and PROLOG, and then execute the programs[8, 9]. They require modifications to the original specifications, and a notation-change is needed when interpreting the animation results. The translation is usually manual allows ample chances to introduce errors. Our approach directly animates Z specifications with little or no modification and the animation results refer to the entities in the original specifications directly.
b) Most of the other approaches support only some basic features of Z and exclude features such as schema calculus and promotion that are essential for building specifications of large-scale software systems. Our approach supports all the features defined in Z. It only excludes those specifications that are not executable.
c) Our approach uses an intermediate representation that is amenable to synthesizing efficient code in an imperative language. It serves as the basis of an ongoing research effort to synthesize efficient code from Z specifications.

Acknowledgement

This work is partly sponsored by the National Science Foundation under contract CCR-9410236.

References

[1] J.M. Spivey, *The Z Notation: A Reference Manual. 2nd ed.* Prentice-Hall, London, 1992.

[2] J.B. Wordsworth, *Software Development with Z: A Practical Approach to Formal Methods in Software Engineering*, Addison-Wesley, 1992.

[3] I. Houston and S. King, "CICS Project Report," *Proc. VDM'91 – Formal Software Development Methods,* LNCS No. 551, pp. 588-596.

[4] D. Craigen, S. Gerhart, and T. Ralston, *An International Survey of Industrial Application of Formal Methods.* Mar. 1993, NISTGCR 93/626.

[5] N.E. Fuchs, "Specifications are (Preferably) Executable," *Software Engineering Journal*, Vol. 7, No. 5, Sept. 1992, pp. 323-334.

[6] I.J. Hayes, and C.B. Jones, "Specifications are not Necessary Executable," *Software Engineering Journal,* Vol. 4, No. 6, Nov. 1989, pp. 330-338.

[7] X. Jia, *ZTC: A Z Type Checker, User's Guide, version 2.0.* April 1995. (The ZTC tool and documentation are available on Internet via anonymous ftp at `ise.cs.depaul.edu`.)

[8] V. Doma, and R. Nicholl, "EZ: A System for Automatic Prototyping of Z Specifications," *Proc. VDM'91: Formal Software Development Methods*, LNCS No. 551, 1991, pp. 189-203.

[9] M.M. West, and B.M. Eaglestone, "Software Development: Two Approaches to Animation of Z Specifications Using Prolog," *Software Engineering Journal*, Vol. 7, No. 4, July 1992, pp. 264-276.

[10] I.J. Hayes (ed.), *Specification Case Studies.* Second edition. Londen: Prentice-Hall, 1992.