Analyzing and animating declarative specifications

S. Cimato and C. Mascolo
Dipartimento di Scienze dell'Informazione
Università di Bologna
e-mail: {cimato,mascolo}@cs.unibo.it

Abstract

Z is a declarative, not executable specification language. We introduce two methods for formal analysis and testing of behavioral aspects of Z specifications. We define a chemical operational semantics, which supports an abstract execution model and some new logic constructs to allow the verification of dynamic properties. We then describe a parallel animator of Z specifications which automatically constructs distributed prototypes in Shared Prolog directly from a refined specification.

1 Introduction

The introduction of formal methods in software engineering practice can increase the quality and the reliability in the development process of complex software systems. Specification documents, written with mathematical notation, give a more precise and rigorous description of the system being developed, reducing possible ambiguities and inconsistencies. Consequently, errors can be earlier recognized and earlier corrected, saving in this way time and money.

Z notation is a well known specification language currently used to formally describe and analyze the requirements and the architectures of a wide range of hardware and software systems. The main field of application for Z language has been the specification of sequential systems with many examples found in literature. Concurrent, reactive, or even distributed systems, are in general difficult to be fully described and analyzed using Z, since it offers very poor support for the inspection of the behavioral aspects of this kind of systems.

To overcome these difficulties, different attempts have been made to integrate Z with some other formalism well suited to specify dynamic properties as Petri Nets [8], temporal logic [3] or TLA [10]. The drawback of these integrations is that no further tool is usually supplied to support the engineering of formal requirements written in these hybrid notations,

In this paper we focus on Z specifications of concurrent systems, offering a specifier two complementary approaches to validate and test a specification. In addition to the abstract, declarative semantics given in [13], we introduce a new operational semantics based on the *chemical metaphor* embedded in the notation of the Chemical Abstract Machine [1]. On such a semantics we define a Unity-like logic offering a number of constructs which can be used to define and analyze dynamic properties. In addition we offer a parallel animator which automatically generates a distributed prototype of the system. The tool is a parallel animator: it consists of a source-to-source compiler for the automatic translation of a Z specification into a program written in the coordination language Shared Prolog, which has a chemical semantics as well. The obtained prototype, running in a truly concurrent, framework can be used to debug it from errors and inconsistencies. The use of a combination of logic programming with coordination mechanisms has the effect that the resulting animation can be executed in a truly distributed environment to make observable the concurrent behaviors of the system being specified.

2 An Operational Semantics for Z

The standard Z semantics [12, 2] is declarative and does not offer any formalization for concurrency. Thus, we have defined an operational semantics based on a chemical model focusing, for simplicity and conciseness, on a restricted version of Z. The interpretation we give for Z specification allow us to deal with concurrent dynamics.

A Chemical Abstract Machine is a triple (G, C, R) where G is a grammar, C is a set of configurations (the language generated by the grammar) or molecules, and R is a set of the rules $condition(C) \times \mathrm{bag}\ C \times \mathrm{bag}\ C$. A solution is a multiset of molecules: $\mathrm{bag}\ C$.

Concurrency in the Chemical Abstract Machine (CHAM) is easily managed since several rules can react simultaneously on a solution, providing they use different molecules in their reactions. If two rules conflict, in the sense that they "consume" the same molecules a non deterministic choice is made.

To give semantics to Z constructs we recognize State schemas and Operation schemas as the elementary components of any Z document. We associate each instance of a state schema with a solution where, in some way, each variable is a subsolution (in many cases a single molecule). An operation schema is associated with a chemical rule where premises and consequences are solutions composed of pre and post conditions of the operation.

A molecule is a tuple including a name, a type, and a value, while a solution is a bag of molecules. A rule is composed of a conditional part which defines the applicability of the rule, and two solutions, to indicate molecules to be deleted (*pretuples*) and added (*posttuples*) respectively, to the state solution:

$MOLECULE == NAME \times TYPE \times VALUE$

$SOLUTION == \mathrm{bag}\ MOLECULE$

$RULE == CONDITION \times SOLUTION \times SOLUTION$

A rule is applicable to a solution if the solution contains molecules that satisfy the conditional part ($CONDITION$) of the rule and molecules that match the pretuples of the rule.

Functions *Fsem* and *Fsem_op* associate a solution with a schema instance and a rule with an operation schema respectively; they are formally defined in [4]. Every identifier of the schema instance is associated with a subsolution while *Fsem_op* associates pre and postconditions of a schema with different part of the rule. We remark that Z sets and bags are decomposed by *Fsem* in several molecules so to increase potential concurrency of the model. Rules can fire concurrently when they are enabled by conditions and do not conflict on pretuples molecules; hence, following our interpretation, enabled operation schemas which do not modify the same variables can react in parallel.

On the semantics just described, we can now define an *execution model*, namely a way of abstractly executing a Z specification document. It represents the unfolding of the application of the rules of the operational semantics. For every state schema s we can construct an abstract execution tree which has the root node void and to which the first operation applied is the initialization operation without any preconditions. From every node there can be several different applicable operation sets. Each branch of the tree corresponds to the application of a group of enabled operations which do not conflict, as dictated by the CHAM model [1].

Every node is labeled with (s, seq), where s is an instance of a state schema and seq is a sequence of operations sets. The chemical interpretation imposes that for every node:

- all operations in the sets belonging to the sequence seq must be enabled on s and must act on the state schema of which s is instance;
- all the operations applied to a node cannot conflict;
- for every s', label of one of the children of the node labeled (s, seq), there must hold the postconditions of all the operations in the operations set applied to reach that node.

Having described the execution model is now possible to verify dynamic properties describing the behavior of the system: properties have a chemical interpretation as well, so that we can analyze their truth on the execution tree. However, we need a fairness assumption on the execution model: repeatedly enabled operations will eventually be executed, that is, the non determinism in the choice of non conflicting operations ready to fire does not cause unlimited delay. In order to be able to reason on dynamic properties, like deadlocks and starvation, we borrow a few constructs (**unless**, **stable**, **invariant**, **ensures**, **leads_to**) from the Unity logic and formalize them on our model (details in [4]). It is now possible to introduce and verify dynamic requirements with the help of these constructs. Properties are predicates (as the ones in the operation schemas) built using some logic operators ($\wedge$, $\vee$, $\neg$, $\Leftrightarrow$, $\Rightarrow$) and Unity constructs stated out of every operation schema.

3 A case of study: a lift system

In order to explain our approach we use an example widely discussed when talking about concurrent dynamics of formal specifications: the Lift System [7, 11, 6].

A lift controller system has to service requests coming from the buttons placed on the floors of a building. The lift is moved by the controller in a direction satisfying the pending requests until no more requests are found; in this case the lift changes direction to service other new or pending requests.

A Z specification of a lift system is given in [7] where Unity like logic constructs are used to state some liveness and safety properties. However we found that the intended operational semantic model is strongly underspecified and the analysis method is unclear. With respect to [7], we give a very similar version of the lift problem, but we provide a completely different interpretation.

A lift can be defined by its position, direction, state, and door state. The direction of the lift can be *up* or *down*, while the state indicates if it is moving or stopped. The lift door opens when the lift arrives at a floor and it is closed while the state is *moving*. Possible requests are up or down requests.

Lift

$position : \mathbb{N}$
$direction : DIRECTION$
$state : STATE$
$door : DOOR$

Lift_System

$Lift$
$requests : \mathbb{P}(REQUEST\_TYPE \times \mathbb{N})$

Init_Lift_System

$Lift\_System'$

$position' = 1$
$direction' = up$
$state' = stopped$
$door' = open$
$requests' = \emptyset$

Schemas above describe the system and its initialization operation. *Make_Requests* schema adds a new request to the *requests* set:

Make_Requests

$\Delta Lift\_System$
$r? : REQUEST\_TYPE$
$f? : \mathbb{N}$

$requests' = requests \cup \{(r?, f?)\}$

$$
\begin{array}{|l}
\hline
\multicolumn{1}{l}{\textit{Move\_Up\_Up}} \\
\Delta \textit{Lift\_System} \\
\hline
door = closed \\
(\exists f : \mathbb{N} \mid (up\_request, f) \in requests \bullet f > position) \\
direction = up \\
position' = position + 1 \\
direction' = up \\
state' = moving \\
door' = closed \\
\hline
\end{array}
$$

Operation schema *Move_Up_Up* defines the operation of moving the lift up if up requests are presents above the lift. *Move_Up_Down* is a similar operation that defines the changing of direction; *Move_Down_Down* and *Move_Up_Down* are the specular schemas for the moving of the lift down, we do not report here for conciseness. The other operation schemas are *Open_Door* and *Close_Door* which respectively describe the operation of opening the door when requests are present at the current floor and the operation of closing the door after served requests.

We can now illustrate our interpretation showing the mapping of the operation schema above to a chemical rule. The initialization operation (*Init_System*) is mapped on a chemical rule having no conditions and no pretuples and as posttuples the following solution:

$$
\begin{aligned}
&(position, \mathbb{N}, 1), (direction, DIRECTION, up),\\
&(state, STATE, stopped), (door, DOOR, open)
\end{aligned}
$$

The rule corresponding to the operation *Move_Up_Up* has $f > pos$ as condition and:

$$
\begin{aligned}
&(door, DOOR, closed), (requests, \mathbb{P}(REQUEST\_TYPE \times \mathbb{N}), (up\_request, f)),\\
&(position, \mathbb{N}, pos), (direction, DIRECTION, up)
\end{aligned}
$$

as premises and:

$$
\begin{aligned}
&(direction, DIRECTION, up)), (position, \mathbb{N}, pos + 1)),\\
&(state, STATE, moving), (door, DOOR, closed),\\
&(requests, \mathbb{P}(REQUEST\_TYPE \times \mathbb{N}), (up\_request, f))
\end{aligned}
$$

as consequences. The rules obtained from the mapping of the other operations are very similar.

Using the constructs above introduced we can now state some properties about the Lift system which can be easily proved [4].

- **Theorem 1**: **Invariant** $door = open \Leftrightarrow state = stopped$
- **Theorem 2**: $(t, f) \in requests \wedge f = position \wedge door = closed$ **ensures** $door = open$
- **Theorem 3**: $(r, f) \in requests \wedge r > position \wedge direction = up \wedge state = moving \wedge door = closed$ **leads_to** $position = r$

4 Animating a Z specification

A formal model of the system to be developed allows the specifier to state the properties the system is expected to have and to formally analyze its functionalities [9]. In general, specifications are not necessarily executable since their aim is to define abstract properties of the system being built, and not the design decisions or the implementation details used in the system. Techniques like animation have been introduced to overcome the difficulty of using a non executable specification language, allowing the specifier to either test or rapidly implement his specification document.

Z is not executable because it allows to declare not computable entities, like infinite sets or non computable functions, and to express properties and operations on them. A number of approaches have been used to transform a Z specification into an executable form, using different programming languages and different methodologies [14]. There are several problems that must be faced according to the chosen method of translation and the target language. In general, most problems derive from trying to match different abstraction levels, namely the one of the Z specification with that one of a particular programming language. Any acceptable solution has to balance declarativeness versus efficiency in the sense that we want not only an executable form of a very high level specification, but also a reasonably efficient execution to test the specification [2].

A straightforward way to animate Z documents, seems to be the mapping of Z specifications into Prolog programs. Practice shows that most predicates found in Z documents have an easy implementation in terms of Prolog clauses and many attempts have been made in this direction [14]. The main novelty of our approach is that the target language is a coordination language based on multiset rewriting, namely Shared Prolog [5]: it can be considered as an extension of Prolog with support for explicit parallelism. We have built a compiler which produces a formally correct distributed program in Shared Prolog directly from its abstract specification in Z. In our knowledge this is the first attempt to build an animation of Z specifications using a coordination language.

4.1 Shared Prolog

Shared Prolog is a coordination language based on the combination of the shared dataspace coordination model (as in Linda) with logic programming computing (as in Prolog) [5]. A SP program consists of a set of modules called *theories*, an *initial goal* and an initial coordination medium called *tuple_space* or *blackboard*. The blackboard is a multiset of logical tuples (Prolog atoms). The initial content of the blackboard is defined by a special goal (tuple_space) and another special goal (initial_goal) defines some agents which share the tuple space:

```
tuple_space{tuple-1, ... ,tuple-n}.
initial_goal(agent-1 '||' .. '||' agent-n).
```

An agent represents a process executing a "theory" and accessing the tuple space for reading/consuming/producing tuples. A *theory* is a Prolog program extended with mechanisms for coordination through the tuple space. Each theory is composed of a set of *patterns* and a *knowledge base*. Syntactically a theory looks as follows:

```
agent theory_name (V_1,..,V_n)
   eval
        pattern_1 # ... # pattern_k
with knowledge base
```

The theory name is a functor with zero or more arguments V_i which are logic variables scoping over all the patterns. A pattern has the following form:

```
{in_guard},read_guard
--->
body {out_set}, fail failure_set
```

The *in_guard* and the *read_guard* are evaluated trying to unify a multiset of tuples in the blackboard with those contained in the guard. The body of a pattern contains predicates defined in the knowledge base of the theory. Semantically, each pattern consists of two components named *preactivation* and *postactivation*. The former defines an activation condition consisting of reading and/or deleting some tuples from the blackboard. The latter consists of a Prolog goal (*body*) and two multisets of tuples: one to be added to the blackboard in case of successful evaluation of the goal (*out_set*), and the other to be added in case of body failure (*failure_set*). Initially every pattern of a theory is examined to check the satisfiability of its own guard. If a guard is satisfied, the corresponding body is evaluated and tuples are added to the blackboard. If several patterns satisfy their preactivation, one is nondeterministically chosen and executed.

4.2 Animating a Z specification with SP

According to the specification style described in section 2, a Z specification document is composed of state schemas which contain predicates on the abstract state of the system, and operation schemas which contain predicates describing how the state of the system may evolve. Operation schemas consist of a set of preconditions which specify the set of valid states of the system to which it is possible to apply such an operation and a set of postconditions which define the set of states that can be reached after the application of the operation itself.

An operation is enabled if the current state of the system satisfies its preconditions and two (or more) operations are concurrent if they are enabled at the same time. Once satisfied the preconditions of the schema considered, the defined operation can be executed. This implies that in an animation context, several operations can be executed at the same time. For this reason the translation of a specification into a concurrent language has the immediate effect of capturing and making observable the concurrent characteristics of the system being specified. In order to simplify its translation we have restricted Z to a number of admissible constructs (we call *Zel* - Z elementary - the resulting notation). The limitations we have introduced concern types and use of variables, form of predicates, and usability of some quantifiers. No restrictions are imposed about schemas construction and connections.

Types can be defined in Zel as: *predefinite* type (as $\mathbb{N}$ e $\mathbb{Z}$), *given set*, *structured type* (defined by enumerating elements), applying *type constructors* (as $\mathbb{P}$,bag,seq,$\times$), *schema type*.

Variables in the declarative part of a schema must be defined as $\mathbb{P}$, bag or seq of another type X, because they are mapped into SP tuples, as we will see later. These restrictions do not apply if a variable is inside the scope of an existential or universal quantifier in the predicative part of a schema. Special attention must be paid to i/o variables. If the type of an input(output) variable is defined as $\mathbb{P}\,X$, the corresponding predicates must be translated using another variable of type X existentially quantified and whose singleton set must be equal to the original variable.

In our approach only a subset of Z predicates can be used in specifications, since not all the predicates are easy to animate. In the predicative part of a schema, it is possible to use only $\wedge$ as logical operator, while $\neg$ can be used in conjunction with membership tests. The order of the predicates is important and influences the SP program obtained from the animation. One must pay attention to define predicates without unbound variables, so that the corresponding SP program does not evaluate non ground terms.

However, properties can be defined using non animatable predicates and the syntactical restrictions are not applied; only under these circumstances other logical operators (e.g. $\vee$) or the universal quantifier are allowed. Although these predicates are type checked, they produce no effect on the resulting animation.

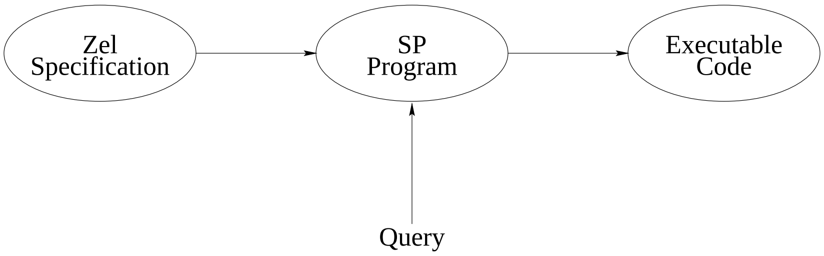
Once a Zel specification document has been obtained (by manual refinement of the original Z specification) the animation process continues using a translator tool which automatically produces a SP program.

We briefly describe now the ideas we used to build the translator. Each simple schema not defined by a schema calculus expression in the Zel document is transformed into a SP pattern; instead, a schema defined by a schema calculus expression is translated into a theory containing the patterns corresponding to the simple schemas which are operands in the expression. Each variable defined in the predicative part of a Zel schema corresponds to a unique SP tuple in the blackboard, obtained concatenating the name of the variable and the schema in which it is defined. Input variables are removed from the blackboard, while output variables are added. For each schema, the system tries to determine a couple of animatable predicates in the list of precondition and postconditions, and the corresponding action is added to the pattern.

In the SP program this is equivalent to include a tuple in the in-guard of the corresponding pattern, with the meaning that it will be removed from the blackboard.

4.3 Querying the animation

In the version we use, SP programs are static: it is not possible to activate agents during the execution, they must be all defined in the initial goal. So, the SP program obtained from the transformation process has to be initialized before it can be executed. The initial state of the blackboard and the initial goal aren't obtained directly by the automatic translation of the specification, but must be defined in the query. Moreover, tuples representing input data for the animation must be added to the tuple space in the initial blackboard.



The diagram above visualizes the animation process for a given specification. Once the initial goal has been defined and the tuple space has been opportunely modified, the SP program is passed to the Shared Prolog compiler producing the executable code corresponding to the requested animation. This code runs on a network of workstations.

5 Animating the Lift System

Let us now consider again the lift system introduced in Section 3 to explain the process of animating a specification. The first step consists of refining the Z document obtaining a Zel specification of the system. The following Zel schema corresponds to the Z schema shown in sect. 3 describing the operation of moving the lift up, provided that there is at least one pending request above its current position. Few changes, concerning variables defined as $\mathbb{P}\ sometype$ and i/o variables, have been necessary to reformulate the schema.

$$
\begin{array}{l}
\textit{Move\_Up\_Up} \\
\hline
\Delta \textit{LiftSystem} \\
\hline
\exists\, \textit{tf} : \textit{REQUESTTYPE} \times \mathbb{N};\ t : \textit{REQUESTTYPE}; \\
f, p1, p2 : \mathbb{N};\ d : \textit{DOOR};\ c : \textit{DIRECTION} \bullet \\
\quad (\textit{tf} \in \textit{requests} \wedge \textit{tf} = (t, f) \wedge \\
\quad p1 \in \textit{position} \wedge f > p1 \wedge p2 = p1 + 1 \wedge \\
\quad d = \textit{closed} \wedge d \in \textit{door} \wedge \\
\quad c = \textit{up} \wedge c \in \textit{direction} \wedge \\
\quad \textit{position}' = \textit{position} \setminus \{p1\} \cup \{p2\} \wedge \\
\quad \textit{direction}' = \{\textit{up}\} \wedge \\
\quad \textit{state}' = \{\textit{moving}\} \wedge \\
\quad \textit{door}' = \{\textit{closed}\}) \\
\hline
\end{array}
$$

The next step is the automatic translation of the specification in Shared Prolog. Here is the code obtained by translating the schema *Move_Up_Up* above reported.

```
    {moveupup},
    {position__liftsystem(P1)}, requests__liftsystem(TF),
    TF=(T,F), F>P1, P2 is P1 + 1,
    D=closed, door__liftsystem(D),
    C=up, direction__liftsystem(C),
    {findall(direction__liftsystem(_V3),_V4)},
    {findall(state__liftsystem(_V5),_V6)},
    {findall(door__liftsystem(_V7),_V8)}
  ->
    {position__liftsystem(P2), direction__liftsystem(up),
     state__liftsystem(moving), door__liftsystem(closed)}
```

As we said above, Sp programs are static. So we have to modify the initial tuple space obtained from the *InitSystem_Lift* schema, in order to satisfy a particular query to be animated. For example, we put into the tuple space a set of tuples representing a sequence of service requests for the lift. We have also to activate an agent in the initial goal:

```
tuple_space{position__liftsystem(1),direction__liftsystem(up),
          state__liftsystem(stopped),door__liftsystem(open),
          r((upRequest,3)),r((upRequest,1)),r((upRequest,4))}.
initial_goal (op_liftsystem).
```

The execution of the code with the above sequence of service requests produces as result the trace in figure (b) where the final state of the lift is also shown. By examining different traces from different test cases, the specifier can debug a specification, possibly correcting unwanted behavior of the system due to mistakes in the specification. The part (a) of the figure shows, for example, the animation report if we "forget" to include in the *Move_Up_Up* and *Move_Down_Up* schemas the test concerning the shutting of door:

```
req upRequest,3
up from 1 to 2
up from 2 to 3
open 3
req upRequest,4
close[]
up from 3 to 4
open 4
req upRequest,5
up from 4 to 5
open 5
close[]
down from 5 to 4
open 4
final set:[(upRequest,4)]
final position:  floor 4 door:  open
```

(a)

```
req upRequest,3
req upRequest,1
close[]
up from 1 to 2
up from 2 to 3
open 3
req upRequest,4
close[]
up from 3 to 4
open 4
final set:[(upRequest,4)]
final position:  floor 4 door:  open
```

(b)

The above trace shows that a lift can move to a new floor to satisfy a request *without closing its doors*; moreover, since requests are deleted in the *Close_Door* schema, request for floor 4 remains among the pending requests list and it is serviced twice.

Testing a specification can be carried on giving in input different test cases that are instances of a particular functional requirement and observing the resulting animation. An unsuccessful case shows that the functional requirements tested is not valid, but only if all instances of a functional requirement are tested satisfiability is demonstrated [9]. However, the advantages of the animation are in terms of accelerating the validation of the specification with respect to the user requirements and increasing the degree of trust of the specifier in the system being built. As result an early detection of errors can be accomplished, saving development time and reducing overall costs.

6 Conclusions and future works

Formalizing dynamics of concurrent and distributed systems is one of the topic challenges of declarative languages. In this paper we provide two methodologies for the analysis and testing of declarative specifications. The former is more abstract and is able to deal with global behavioral components. The latter is more concrete and offers an high level prototype for the study of system behavior using some specific test cases. The main novelties of our approach are the introduction of a Z chemical semantics to analyze operational properties of reactive systems and the integration of parallelism into the prototyping language to highlight concurrent behaviors.

Future developments of our work aim at making clear the relationship between the original declarative semantics and the chemical semantics we provide for Z specification, studying its correctness and completeness. The animator could be improved with the extension both of Zel notation, augmenting the number of animatable predicates, and of SP language adding new constructs to increase its expressive power.

Acknowledgments. The authors would especially like to thank Paolo Ciancarini and Mauro Gaspari for their helpful comments and suggestions. Partial support for this work was provided by the Commission of European Union under ESPRIT Programme Basic Research Project 9102 (COORDINATION), and by the Italian MURST 40%- Progetto "Ingegneria del Software".

References

[1] G. Boudol. Some Chemical Abstract Machines. In J. deBakker, W. deRoever, and G. Rozenberg, editors, *A Decade of Concurrency*, volume 803 of *Lecture Notes in Computer Science*, pages 92–123. Springer-Verlag, Berlin, 1993.

[2] P. Breuer and J. Bowen. Towards Correct Executable Semantics for Z. In J. Bowen and J. Hall, editors, *Proc. 8th Z Users Workshop (ZUM94)*, Workshops in Computing, pages 185–212, Cambridge, 1994. Springer-Verlag, Berlin.

[3] D. Carrington, D. Duke, R. Duke, P. King, G. Rose, and G. Smith. Object-Z: an Object-Oriented Extension to Z. In *Formal Description Techniques (FORTE 89)*, pages 281–296. North-Holland, 1989.

[4] P. Ciancarini, S. Cimato, and C. Mascolo. Engineering formal requirements: an analysis and testing method for Z documents. Technical Report UBLCS-6, Dipartimento di Scienze dell'Informazione, Università di Bologna, Italy, 1996.

[5] P. Ciancarini and M. Gaspari. Rule Based Coordination of Logic Programs. *Computer Languages*, (to appear), 1996.

[6] J. Cuellar, I. Wildgruber, and D. Barnard. Combining the design of industrial systems with effective verification techniques. In *FME'94: Industrial Benefit of Formal Methods*, volume 873 of *Lecture Notes in Computer Science*, pages 639–658, Barcelona, Spain, 1994. Springer-Verlag, Berlin.

[7] A. Evans. Specifying and Verifying Concurrent Systems Using Z. In M. Bertran, T. Denvir, and M. Naftalin, editors, *Proc. FME'94 Industrial Benefits of Formal Methods*, volume 873 of *Lecture Notes in Computer Science*, pages 366–380. Springer-Verlag, Berlin, 1994.

[8] X. He. PZ Nets: A Formal Method Integrating Petri Nets with Z. In *Proc. 7th Int. Conf. on Software Engineering and Knowledge Engineering*, pages 173–180, Rockville, Maryland, 1995. Knowledge Systems Institute.

[9] R. Kemmerer. Testing Formal Specifications to Detect Design Errors. *IEEE Transactions on Software Engineering*, 11(1):32–43, January 1985.

[10] L. Lamport. TLZ. In J. Bowen and J. Hall, editors, *Proc. 8th Z Users Workshop (ZUM94)*, Workshops in Computing, pages 267–268, Cambridge, 1994. Springer-Verlag, Berlin.

[11] D. Richardson, S. Aha, and T. O'Malley. Specification-based Test Oracles for Reactive Systems. In *Proc. 14th IEEE Int. Conf. on Software Engineering*, pages 105–118, Melbourne, Australia, 1992.

[12] J. Spivey. *Understanding Z.* Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1988.

[13] J. Spivey. *The Z Notation. A Reference Manual.* Prentice-Hall, 2 edition, 1992.

[14] L. Sterling, P. Ciancarini, and T. Turnidge. On the Animation of Not Executable Specifications by Prolog. *Int. Journal on Software Engineering and Knowledge Engineering*, 6(1):(to appear), 1996.