Analyzing the dynamics of a Z specification

Paolo Ciancarini and Cecilia Mascolo

Dipartimento di Scienze dell'Informazione,
Università di Bologna,
e-mail: {cianca,mascolo}@cs.unibo.it

Abstract. We present a method for analyzing the dynamics of a Z document describing a non-sequential system. First a formal operational semantics based on the chemical metaphor is given to Z. Then, some Unity-like temporal logic constructs are defined on such a formal operational semantics in order to allow the specification and analysis of dynamic and temporal properties of concurrent systems, such as safety and liveness properties.

1 Introduction

The introduction of formal methods increased the usefulness of software specification documents by allowing to automatically check them and to formally reason on them. The Z notation [18] is currently widely used as a non executable specification language to formally describe and analyze the requirements and the architectures of software systems. However, Z has been mostly used for the specification of sequential systems. In fact, even if in the recent years it has been used for specifying concurrent, reactive, or even distributed systems, in general non-sequential systems are difficult to be perfectly described and then analyzed using Z.

Even if Z is not executable several researchers have tried to improve the ability of Z to express and support the analysis of dynamic features of non-sequential systems. The integration of Z with operational notations like CSP [1] or Petri Nets [10, 12], or the use of temporal logic for analyzing Z documents [16, 5, 9, 14] are some of the approaches suggested. These approaches all suffer from the same problem: the integration of different notations in a uniform specification method is not formalized because a clear and consistent integration is in general difficult to accomplish.

The approach we introduce here is new, insofar as we formally define in a unified framework both an operational semantics and a logic based on such a semantics to reason on Z documents. The operational semantics we introduce is based on the *chemical metaphor* embedded in the notation of the Chemical Abstract Machine (Cham) [2], The logic includes a number of constructs which allow the definition of dynamic properties of a system specification. We have chosen some Unity-like [6] logic constructs because of their expressiveness (a similar approach can be found in [9]) and because it has already been proved suitable to be the proof system basis for Swarm language [8]: a multiset transformation based language like the Cham.

The semantics of the constructs is defined on an *execution model* based on the operational semantics.

This paper has the following structure: in Sect. 2 a specification style and an interpretation of Z documents as sets of state and operation schemas are given; Sect. 3 describes the operational semantics based on the chemical metaphor we adopted. Sect. 4 contains the definition of the execution model imposed on the operational semantics; Sect. 5 introduces the new logic constructs inherited from the Unity language while the final section contains comparisons and conclusions.

2 A specification style and its semantics

For conciseness, our specification style considers a restricted version of Z; we specify such a fragment using Z itself, thus following the Z tradition [17, 11].

The elementary components of a Z specification are State schemas and Operation schemas. A schema is defined as follows:

$$
\begin{array}{|l}
\hline
\textit{SCHEMA\_STATE} \\
\textit{name} : \textit{NAME} \\
\textit{schema\_imp} : \mathbb{P}\ \textit{NAME} \\
\textit{decl} : \mathbb{P}\ \textit{VAR} \\
\textit{imported} : \mathbb{P}\ \textit{VAR} \\
\hline
\textit{name} \notin \textit{schema\_imp} \\
\forall\, d : \textit{VAR} \mid d \in \textit{imported} \bullet \exists\, s : \textit{NAME} \mid s \in \textit{schema\_imp} \\
\qquad d \in \textit{stateschema}(s).\textit{decl} \\
\hline
\end{array}
$$

where $[\textit{NAME}]$ is a basic type specifying names of variables or schemas, *name* identifies the schema, *schema_imp* is a set of imported schemas names [1], *decl* is a set of identifier declarations, *imported* is the set of imported identifiers; the predicate states that every imported identifier should be declared in one of the imported schemas.

Intuitively, we only consider State schemas without predicative part since we will be able to express these predicates as invariant properties using the logic defined in Sect.5.

Semantically, a State schema s can be seen as the set of all its possible instantiations [17]. A schema instance is an instantiated State schema:

$$
\begin{array}{|l}
\hline
\textit{SCHEMA\_INSTANCE} \\
\textit{sch} : \textit{NAME} \\
\textit{values} : \textit{VAR} \rightarrowtail \textit{IDENT} \\
\hline
\forall\, v : \textit{VAR} \bullet (v \in \textit{sch.decl} \lor v \in \textit{sch.imported}) \Leftrightarrow v \in \operatorname{dom} \textit{values} \\
\hline
\end{array}
$$

where *VAR* has a name and a type and *IDENT* is a bound variable. The predicate in *SCHEMA_INSTANCE* ensures that every variable is mapped on an identifier with same name and type.

[1] We only consider two levels of imported schemas.

An Operation schema is:

$$\begin{array}{|l}
\hline
\textit{SCHEMA_OP} \\
name : NAME_OP \\
delta : NAME \\
environment : ENV \\
inputs : \mathbb{P}\, IDENT \\
precondition : \mathbb{P}\, PRE \\
postcondition : \mathbb{P}\, POST \\
\hline
\exists\, s : SCHEMA_STATE \mid s.name = delta \bullet \\
\quad \forall\, id : IDENT \mid id \in environment.decl \bullet \\
\quad \exists\, v : VAR \mid v.name_id = id.name_id \wedge \\
\quad v.type_id = id.type_id \bullet v \in s.decl \\
\hline
\end{array}$$

where *name* is the operation name, *delta* is the name of the State schema on which the operation acts, *environment* is the environment of the variables declaration, *inputs* are the inputs of the schema, *precondition* is the precondition predicates set and *postcondition* is the postcondition predicates set.

The initialization operations are represented by particular operation schemas without preconditions.

3 Operational semantics

The standard Z semantics [17, 4] does not offer formalization for concurrency. Thus, we have defined a new operational semantics based on the concurrency offered by the chemical model.

3.1 The chemical metaphor

In the Chemical Abstract Machine model [2, 3] *Molecules*, *Solutions*, and *Rules* are the fundamental elements. 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 $condition(C) \times \text{bag } C \times \text{bag } C$. A solution is a multiset of molecules: bag C; $\{\!|\ |\!\}$ symbols usually delimit a solution. Solutions are considered the Abstract Machine states. They can be composed of other subsolutions using $\uplus$: $S = S_1 \uplus S_2$.

There are some general laws valid for any Cham:

- **Reaction Law**: an instance of the right-hand side of a rule can replace the corresponding instance of its left-hand side if conditions on the molecules hold. Given a rule
  $condition(m_1, m_2..m_k) \rightarrow m_1, m_2..m_k \Rightarrow m_1', m_2'..m_l'$
  if $M_1, M_2..M_k, M_1', M_2'..M_l'$ are instances of the m_i's and the m_j''s by a common substitution, then
  $condition(M_1, M_2..M_k) \rightarrow \{\!|\ M_1, M_2..M_k\ |\!\} \Rightarrow \{\!|\ M_1', M_2'..M_l'\ |\!\}$

- **Chemical Law**: reactions can freely happen in a solution

$$\frac{S \Rightarrow S1}{S \uplus S2 \Rightarrow S1 \uplus S2}$$

- **Membrane Law**: a subsolution evolves freely in every context

$$\frac{S \Rightarrow S1}{\{\!|\ C[S]\ |\!\} \Rightarrow \{\!|\ C[S1]\ |\!\}}$$

where $C[\,]$ indicates a context.

In a Cham two instances of rules can fire concurrently if they do not need the same molecules to react on; so many instances of rules can progress simultaneously on a solution. If two instances of rules conflict, in the sense that they "consume" the same molecules, the choice of which to let react is non deterministic.

We consider a *fair* Cham where repeatedly enabled rules will eventually be fired: in this way it is possible to prove properties defined using Unity logic constructs (Sect. 5).

3.2 An Operational Semantics for Z

We introduce an interpretation of Z specifications which allows us to deal with concurrent dynamics. Intuitively, an instance of a State schema (*inst*) is associated with a solution where, in some way, each variable is a subsolution (in many cases a single molecule). Instead, an operation schema corresponds to a chemical rule where pretuples and posttuples are solutions composed of pre and post conditions of the operation.

A molecule is represented as a tuple including a name, a type, and a value; 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 and added, respectively, to the state solution:

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

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

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

We call the first $SOLUTION$ "pretuples" and the second "posttuples" to avoid ambiguities. A rule is applicable to a solution if the solution contains molecules that satisfy the conditional parts ($CONDITION$) of the rule and molecules that match the pretuples of the rule.

The semantic function $FSem$ associates a solution to a schema_instance:

$Fsem : SCHEMA\_INSTANCE \rightarrow SOLUTION$

Every identifier in the schema instance is associated with a subsolution (not necessarily a single molecule). We remark that Z sets and bags are decomposed by this function in several molecules to increase potential concurrency.

$Fsem\_op$ associates a rule to an operation schema [2]:

[2] A similar function can be defined for the initialization operation, where no preconditions are present [3].

$Fsem\_op : SCHEMA\_OP \rightarrow RULE$
$Fsem\_op$ associates to pre and postcondition different part of the rule:

- Every Z schema postcondition that specifies the removal of an element from a set or bag is mapped on a pretuple of the rule (molecule to be deleted).
- Every postcondition that specifies the insertion of an element in a set or bag is mapped on a posttuple of the rule (molecules to be added).
- Every Z precondition that defines a membership ($\in$, in) is mapped on a pretuple (a removal) and also on a posttuple (reinsertion) if the Z postcondition does not contain an indication of removal of that element: in other words, a check of membership is evaluated as a removal followed by a reinsertion.
- Postconditions containing mathematical operators ($+$, $-$,..) on naturals are encoded deleting one molecule and adding the updated molecule.
  Example: $x' = x + 1$ is evaluated as $(x, \mathbb{N}, v)$ in pretuples and $(x, \mathbb{N}, v + 1)$ in posttuples of the rule.
- Preconditions containing relational operators ($<$,$>$) are encoded as conditions, but the molecule corresponding to the variable is deleted and readded as already described [3].
  Example: $x < 5$ is seen as $v < 5 \rightarrow (x, \mathbb{N}, v) \Rightarrow (x, \mathbb{N}, v)$

Now, thanks to the chemical laws, rules can fire concurrently when they are enabled by conditions and non conflicting on pretuples molecules.

3.3 A simple example

Consider the classic dining philosophers problem. What follows is its formalization in our style.

$FORK ::= fork1 \mid fork2 \mid fork3 \mid fork4 \mid fork5$
$PHILO ::= philo1 \mid philo2 \mid philo3 \mid philo4 \mid philo5$

The following schema illustrates the basic State schema of the system:

System
$think : \mathbb{P}\ PHILO$
$eat : \mathbb{P}\ PHILO$
$have\_right : \mathbb{P}\ PHILO$
$available : \mathbb{P}\ FORK$
$left, right : \mathbb{P}(PHILO \times FORK)$

where: *eat* denotes the set of eating philosophers; *have_right* is the set of philosophers who got the right fork, and wait for the left one; *think* is the set of thinking philosophers; *available* is the set of available forks; *left* and *right* indicate for every philosopher respectively the left and right fork.

[3] This is done according to the chemical semantics where conditions can only be stated on the local molecules involved in the rule [3].

$$
\begin{array}{|l}
\hline \textit{Init\_system} \\
System' \\
\hline
eat' = \varnothing \\
have\_right' = \varnothing \\
think' = \{philo1, philo2, philo3, philo4, philo5\} \\
available' = \{fork1, fork2, fork3, fork4, fork5\} \\
right' = \{(philo1, fork1), (philo2, fork2), (philo3, fork3), \\
\quad (philo4, fork4), (philo5, fork5)\} \\
left' = \{(philo1, fork5), (philo2, fork1), (philo3, fork2), \\
\quad (philo4, fork3), (philo5, fork4)\} \\
\hline
\end{array}
$$

Initially, all philosophers are thinking and all forks are available.

We now define the operations for philosophers:

$$
\begin{array}{|l}
\hline \textit{Right\_Request} \\
\Delta System \\
ph? : PHILO \\
f? : FORK \\
\hline
ph? \in think \\
f? \in available \\
(ph?, f?) \in right \\
have\_right' = have\_right \cup \{ph?\} \\
available' = available \setminus \{f?\} \\
think' = think \setminus \{ph?\} \\
\hline
\end{array}
$$

The schema *RightRequest* defines the operation of taking the right fork.

Operation *Left_Request* is similar: we do no specify it formally.

When $ph?$ has the right fork he can ask for the left one: if the fork is available it is assigned to him.

$$
\begin{array}{|l}
\hline \textit{Thinking} \\
\Delta System \\
ph? : PHILO \\
f?, ff? : FORK \\
\hline
ph? \in eat \\
(ph?, f?) \in right \\
(ph?, ff?) \in left \\
think' = think \cup \{ph?\} \\
eat' = eat \setminus \{ph?\} \\
available' = available \cup \{f?, ff?\} \\
\hline
\end{array}
$$

If $ph?$ quits eating he puts down both forks and begins thinking again.

The initialization operation (*Init_System*) is mapped on a chemical rule having no conditions and no pretuples and as posttuples the following solution:

$(think, \mathbb{P}\, PHILO, philo1), .., (think, \mathbb{P}\, PHILO, philo5),$
$(available, \mathbb{P}\, FORK, fork1), .., (available, \mathbb{P}\, FORK, fork5),$
$(right, \mathbb{P}(PHILO \times FORK), (philo1, fork1)), ..,$
$(right, \mathbb{P}(PHILO \times FORK), (philo5, fork5)),$
$(left, \mathbb{P}(PHILO \times FORK), (philo1, fork5)), ..,$
$(left, \mathbb{P}(PHILO \times FORK), (philo5, fork4))$

The State schema instance obtained after the application of the operation is the same solution presented above. The rule associated with the operation schema *RightRequest* has the following pretuples:

$(think, \mathbb{P}\, PHILO, ph?), (available, \mathbb{P}\, FORK, f?),$
$(right, \mathbb{P}(PHILO \times FORK), (ph?, f?)),$

and posttuples:

$(have\_right, \mathbb{P}\, PHILO, ph?), (right, \mathbb{P}(PHILO \times FORK), (ph?, f?))$

The rule corresponding to the operation *LeftRequest* is similar.
The operation *Thinking* corresponds to the following rule with pretuples:

$(eat, \mathbb{P}\, PHILO, ph?), (right, \mathbb{P}(PHILOF \times ORK), (ph?, f?)),$
$(left, \mathbb{P}(PHILO \times FORK), (ph?, ff?))$

and posttuples:

$(available, \mathbb{P}\, FORK, f?), (available, \mathbb{P}\, FORK, ff?),$
$(think, \mathbb{P}\, PHILO, ph?), (right, \mathbb{P}(PHILO \times FORK), (ph?, f?)),$
$(left, \mathbb{P}(PHILO \times FORK), (ph?, ff?))$

Because of the concurrent interpretation of Z that we are going to give, we make the following assumption: all variables not explicitly mentioned in the postconditions of an operation schema may change (i.e. they have not to be invariant: other operations can concurrently modify them).

This assumption is needed in our interpretation and allows concurrency of the operations. In some papers the assumption introduced is exactly the contrary: "Variables not mentioned in the schemas are considered unchanged" e.g. [16] but this is not standard Z too.

4 The execution model

We make the operational semantics (defined in Sect.3) explicit, to build an *execution model*, namely a way of abstractly executing a Z specification document written according to the style outlined in Sect.2. The execution model is defined on the semantics just described and it represents the unfolding of the application of the semantics rules. From every State schema s a tree (execution model) can be constructed in the following way:

- the root node is void;
- the first operation applied is the initialization operation without any preconditions;
- from every node several different applicable operation sets can exist, (chosen among all the enabled operations on that node), thus introducing non determinism in the choice of the operations being in conflict.
- Each branch corresponds to the application of a group of enabled operations which could be applied without conflicts, as dictated by the Cham model.

In order to allow the specification of the Unity logic constructs using Z as meta-language, we introduce a concept of *execution tree*:

$$\begin{aligned} \mathit{TREE} ::= {} & \mathit{Void\_tree} \\ | \ & \mathit{fork}\langle\!\langle \mathit{PAIR} \times \mathrm{seq}\ \mathit{TREE} \rangle\!\rangle \end{aligned}$$

where

$$\mathit{PAIR} == \mathit{SCHEMA\_INSTANCE} \times \mathrm{seq}\, \mathbb{P}\, \mathit{SCHEMA\_OP}$$

The function *Exec* maps every State schema on an execution tree with particular properties (we omit the Z specification of the function); the chemical interpretation imposes that for every node label (s, seq), where s is an instance and seq is a sequence of operations sets:

- all the operations in the sets belonging to the sequence seq must be enabled on s;
- all the operations in the sets belonging to the sequence seq must act on the State schema of which s is an instance;
- each set, member of the sequence seq, must contain operations that can concur (that is without conflicts);
- 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 (sequence structure help to keep link between nodes and operations set).

5 The logic

Liveness (namely “a good thing will eventually occur”) or safety (namely “a bad thing never happens”) properties can be expressed. Properties are predicates (as the ones in the operation schemas) built using some logic operators ($\wedge$, $\vee$, $\neg$, $\Leftrightarrow$, $\Rightarrow$) and Unity constructs. Properties have a chemical interpretation as well, so that we can analyze the truth of them on the execution model, based on the chemical metaphor too. In order to be able to reason on dynamic properties, we borrow a few constructs from the Unity logic:

- p **unless** q says that whenever p is true during the execution, surely either q will become true or p continues to hold. In particular, on the tree: if p is true on some nodes then on their children q is true or p still holds.

- **Stable** is an alias for p **unless** false, that is when p becomes true it will hold forever. On the tree: if p is true on a node it will remain true for all the subtree of that node.
- **Invariant** p says that p is true forever. That is, for every node of the execution tree p is valid.
- The meaning of p **ensures** q is that when p becomes true then eventually q will hold and before that moment p is still valid. That is, if p is true on a node N, then each branch through N there is a node M below N where q holds and on nodes between nodes N and M in the path, p holds.
- p **leads_to** q has quite the same meaning as **ensures** except that it does not ensure that p is valid until q becomes true. On the tree: if p is true on a node q will eventually hold on a node in all its sub-branches.

The following axiomatic schema shows how we formalize the meaning of the logic constructs on the execution model. We report only the **ensures** definitions:

$$
\begin{array}{|l}
\mathit{ensures} : \mathit{PROPERTY} \times \mathit{PROPERTY} \rightarrow \mathbb{B} \\
\hline
\forall\, p, q : \mathit{PROPERTY} \bullet \mathit{ensures}(p, q) = \mathrm{true} \Leftrightarrow \\
\quad ((\mathit{unless}(p, q) = \mathrm{true}) \wedge \\
\quad (\forall\, e : \mathit{TREE};\ e1 : \mathrm{seq}\ \mathit{TREE}; \\
\quad \mathit{schema} : \mathit{SCHEMA\_STATE};\ \mathit{set} : \mathrm{seq}\,\mathbb{P}\,\mathit{SCHEMA\_OP}; \\
\quad \mathit{inst} : \mathit{SCHEMA\_INSTANCE} \mid \\
\quad \mathit{subtree}(\mathit{Exec}(\mathit{schema}), e) = \mathrm{true} \wedge \mathit{fork}(\mathit{inst}, \mathit{set}, e1) = e \\
\quad \wedge\ \mathit{valid}(\mathit{inst}, \mathit{and}(p, \mathit{not}(q)) = \mathrm{true} \bullet \\
\quad (\exists\, e3 : \mathrm{seq}\ \mathit{TREE};\ \mathit{set}' : \mathrm{seq}\,\mathbb{P}\,\mathit{SCHEMA\_OP}; \\
\quad \mathit{inst}' : \mathit{SCHEMA\_INSTANCE};\ e2 : \mathit{TREE} \mid \\
\quad \mathit{subtree}(e2, e) \wedge \mathit{fork}(\mathit{inst}', \mathit{set}', e3) = e2 \\
\quad \bullet\ \mathit{valid}(\mathit{inst}', q)) = \mathrm{true}))
\end{array}
$$

where function *valid* indicates when a property holds on an instance state. Intuitively this is done considering every property as a solution and analyzing the matching with the state solution like what has been done for rules.

This formalization of *ensures* derives from *unless*; in fact, p **ensures** q if p **unless** q and there exists a branch of the tree that from a state where p is valid (and not q) leads to a state where q holds.

Example We state some properties about the dining philosophers system:

- **Theorem 1** $\mathit{philo}1 \in \mathit{think}$ **ensures** $\mathit{philo}1 \in \mathit{have\_right}$
- **Theorem 2 stable** $(\mathit{philo}1 \in \mathit{have\_right} \wedge \mathit{philo}2 \in \mathit{have\_right} \wedge$ $\mathit{philo}3 \in \mathit{have\_right} \wedge \mathit{philo}4 \in \mathit{have\_right} \wedge \mathit{philo}5 \in \mathit{have\_right})$

The first property states that if $\mathit{philo}1$ is thinking, he will eventually get the right fork; this property can be stated for all other philosophers as well. The second property defines a deadlock: when every philosopher has got the right fork then the system cannot proceed.

Proof of Theorem 1: If p **ensures** q (where p is $philo1 \in think$ and q is $philo1 \in have\_right$) must be valid, first p **unless** q has to hold (see **ensures** formalization). This means that for every enabled operations set on the solution containing the molecule $(think, \mathbb{P}\ PHILO, philo1)$, the application leads to a state where either molecule $(have\_right, \mathbb{P}\ PHILO, philo1)$ is present or the previous molecule is still in the solution (this is the **unless** formalization of our execution model).

Considering our Z specification, we notice that for all the enabled operations sets that we could choose, each of them modifies the instance solution leaving the molecule (in case we choose only operations acting on other philosophers) or $(have\_right, \mathbb{P}\ PHILO, philo1)$ is inserted (in case the operations set contains *RightRequest* that is the only operation enabled for $philo1$ on the state solution considered). Hence, p **unless** q holds.

To prove p **ensures** q is now necessary (following the *ensures* formalization) to ensure that, given a state where $p \wedge \neg\ q$ holds exists an enabled operations set applicable, that leads to a state on which q holds. In our specification we have to prove that on a state where p holds exists an enabled (also not continuously) operation set that leads to q. This set contains the instance of the operation *RightRequest* on $philo1$ and other operations acting on other philosophers. Then remembering that our Cham is fair (Sect. 3.1) we can state that the set will eventually be applied. This completes the proof.

6 Comparisons and conclusions

We have defined a chemical semantics for a fragment of Z, and showed that it offers a good basis for the formalization of logic constructs which allow the expression and the analysis of concurrent properties.

We are studying the possibility to map Z schema inclusion using membranes of Cham and airlock. Some other dynamic aspects could be treated such as execution order, synchronization and communication; we are also studying the possibility to introduce real time in our model.

Formalizing dynamics of concurrent and distributed systems is one of the topical challenges to formal languages. Z has been used in this sense several times; the simplest solution consists of considering an intuitively concurrent semantics for schemas: operations are considered atomic and non determinism guides the choice of the operation to apply. Such a model produces a specification whose analysis can hardly expose concurrent properties. In [13] a specification of the distributed IBM Customer Information Control System (CICS) is presented: although no formalization of Z dynamics can be found in the paper, this is considered one of the most successful applications of Z in this sense, because of the reduction of production costs that the Z specification involved. The paper [15] contains the formal specification of a reactive dialog system: Z schemas are used in order to state invariant properties and a formal interpretation of the behavior of the system is given. However, the approach described in the paper is weak in term of formalization of concurrency and semantics.

More formal approaches integrate Z with other notations; for instance, Benjamin [1] integrates Z schemas with CSP notation. CSP is used to specify an abstract system while Z defines more detailed aspects of the design. The integration is minimal and not formally specified. In [12] Petri Nets are used to formalize control flows, causal relations, and dynamic behavior of systems statically defined using Z; nevertheless the formalization of the interaction between the two notations is also minimal. [10] studies a more formal model of integration of Z with Petri Nets: Petri Nets are mapped on Z specifications so that graphical representation given by Petri Nets can be used to animate Z documents, yet we think a visualization cannot replace formal semantics.

In [16] a formalism based on temporal logic is used to integrate Z schemas. The use of temporal logic has offered good starting points to the study of the dynamics of Z specifications however integration is not supported by semantics. Something more formal has been done for Object-Z [5]: a sequential execution model is introduced, defining a notion of abstract trace as a sequence of pairs (states and operations), and using some temporal logic operators ($\Diamond$, $\Box$, $\bigcirc$) to reason on such a model. TLA has been proposed to be integrated with Z as well, however in this case Z is only used to define *actions* specification [14]. In [9] a Unity like logic is used to formalize properties on the behavior of systems; an interleaving model with atomic operation interpretation is given but not formalized. The simplicity of Unity logic constructs fit quite well with the purpose of effectively specifying systems dynamics.

We remark that the use of Unity logic on a model based on multiset transformation is not new, in fact it has been applied to the Swarm language in order to provide a proof system to a parallel language [8]. The Swarm experience, in which the idea consists of mapping Unity-like constructs on a coordination language similar to Linda, inspired us to make some experiments in concurrent animation of Z. In fact, our semantic model based on multiset transformation offers a good basis for the parallel animator of specifications described in [7]. The animator can compile the Z language into programs written in a coordination language so as to allow a truly concurrent animation.

Acknowledgments. 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. M. Benjamin. A Message Passing System. An example of combining Z and CSP. In J. Nicholls, editor, *Proc. 4th Z Users Workshop (ZUM89)*, Workshops in Computing, pages 221–228, Oxford, 1989. Springer-Verlag, Berlin.
2. G. Berry and G. Boudol. The Chemical Abstract Machine. *Theoretical Computer Science*, 96:217–248, 1992.
3. 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.

4. 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.
5. 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.
6. K. M. Chandy and J. Misra. *Parallel Programming Design*. Addison-Wesley, 1988.
7. P. Ciancarini, S. Cimato, and C. Mascolo. Engineering Formal Requirements: Analysis and Testing. In *Proc. 8th Int. Conf. on Sw. Eng. and Knowledge Eng. (SEKE)*, Lake Tahoe, Ca, June 1996.
8. H. Cunningham and G. Roman. A Unity-Style Programming Logic for Shared Dataspace Programs. *IEEE Transactions on Parallel and Distributed Systems*, 1(3):365–376, July 1990.
9. 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.
10. A. Evans. Visualizing Concurrent Z Specifications. In J. Bowen and J. Hall, editors, *Proc. 8th Z Users Workshop (ZUM94)*, Workshops in Computing, pages 269–281, Cambridge, 1994. Springer-Verlag, Berlin.
11. P. Gardiner, P. Lupton, and J. Woodcock. A simpler semantics for Z. In J. Nicholls, editor, *Proc. 5th Z Users Workshop*, Workshops in Computing, pages 3–11. Springer-Verlag, Berlin, 1990.
12. 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.
13. I. Houston and M. Josephs. Specifying distributed CICS in Z; accessing local and remote resources. *Formal Aspects of Computing*, 6(6):569–579, 1994.
14. 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.
15. K. Narayama and S. Dharap. Invariant Properties in a Dialog System. In M. Moriconi, editor, *Proc. ACM SIGSOFT Int. Workshop on Formal Methods in Software Development*, volume 15:4 of *ACM SIGSOFT Software Engineering Notes*, pages 67–79, 1990.
16. 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.
17. J. Spivey. *Understanding Z*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1988.
18. J. Spivey. *The Z Notation. A Reference Manual*. Prentice-Hall, 2 edition, 1992.

This article was processed using the LaTeX macro package with LLNCS style