Engineering Formal Requirements: an Analysis and Testing Method for Z Documents

P. Ciancarini and S. Cimato and C. Mascolo
Dipartimento di Scienze dell'Informazione
Università di Bologna
Via Mura Anteo Zamboni 7, I-40127 Bologna, Italy
phone:+39 51 354506, fax:+39 51 354510
e-mail: {ciancarini,cimato,mascolo}@cs.unibo.it

Abstract

Z is a declarative, non-executable specification language; its diffusion in the field of requirements engineering outside academia is slow but growing. In this paper we focus on some methods for analyzing and testing Z specification documents, with special emphasis on non-sequential systems specifications. We describe two techniques we have adopted: the former allows the specifier to add to the requirements document a number of properties that then can be checked using a formal semantics; the latter makes it possible to build directly from the requirements specification document a distributed prototype which can be executed and tested over a network of workstations.

1 INTRODUCTION

The Z notation [Spivey, 1992, Brien & Nicholls, 1992] has been used as a specification language to formally describe and analyze the requirements and the design architectures of a wide range of hardware and software systems; for some examples see [Hayes, 1993]. However, Z has been successfully used especially for the specification of sequential systems rather than concurrent systems. Since Z is an extremely abstract specification language, specifiers of concurrent or distributed systems have very poor support and need to take special care of behavioral aspects of their formal documents.

In fact, even though in the recent years Z has been sometimes used for specifying concurrent, reactive, or even distributed systems [Hayes, 1993], non-sequential systems are difficult to be fully described and analyzed using Z. A number of researchers have attempted to overcome the difficulties, usually combining Z with some other formalism well suited to specify dynamic properties; combinations of Z with either Petri Nets [vanHee *et al.*, 1991, He, 1995, Evans, 1994a], temporal logic [Carrington *et al.*, 1989], or TLA [Baumann & Lermer, 1995] have been studied. Usually no tool is given to support the engineering of formal requirements written in these hybrid notations, except a syntax checker based on a LaTeX-like concrete syntax.

The approach we present here offers the specifier two complementary techniques for analyzing, testing, and validating the formal requirements specification of a concurrent system.

The first approach allows a formal analysis of a Z document: its dynamics is formally expressed and studied on an *execution model*, in order to be considered a sort of abstract symbolic execution in the sense of [Kemmerer, 1985]. The second approach consists of using a tool for automatic generation of a distributed prototype of the system which helps to debug the specification from errors, inconsistencies, and ambiguities in a truly concurrent framework.

Z has a very abstract, declarative semantics [Spivey, 1992]. We introduce a new operational semantics: it is based on the *chemical metaphor* embedded in the notation of the Chemical Abstract Machine [Berry & Boudol, 1992], which itself has been recently proposed as a specification notation [Inverardi & Wolf, 1995]. On such a semantics we define a logic offering a number of constructs which can be used to define and analyze dynamic properties. Moreover, the semantics has been used also to design a tool that is a parallel animator: it consists of a source-to-source translator which compiles a Z document into a program written in the concurrent language Shared Prolog, which has a chemical semantics as well. The resulting animation can be executed in a distributed environment in order to make observable the concurrent behaviors of the system specified.

This paper has the following structure: Sect.2 shortly introduces Z and surveys the main tools that a requirements engineer can use to validate a formal requirements document. Sect.3 describes our analysis techniques discussing a simple example with concurrency features we will use in the rest of the paper. Sect.4 describes the formal operational semantics and the logic we use to analyze Z documents. Sect.5 discusses the concept of animating a Z specification; in Sect.6 and Sect.7 we discuss an actual animation and its refinement. Finally, in Sect.8 we describe some related and current work.

2 USING Z FOR REQUIREMENTS ENGINEERING

We think of the specifier acting as an expert in formal methods who works on behalf of the customer. Specifiers act differently from designers as they do not deal with architectural or organizational issues. The specifier writes a formal requirements document cooperating with the user. At this step, a first statement of the problem to be analyzed is given by means of the presentation of the general framework which is subsequently implemented. The accuracy in the description of the required system has great importance, because it helps especially to avoid errors that could be more expensive to correct if discovered in subsequent phases. Any error found by the specifier at this stage requires a reformulation of the specification, limiting the required modifications at a very high abstraction level. Any requirements inconsistency or incompleteness found at subsequent stages in the development process could imply a redefinition of the initial specification document, affecting the whole development process [Kemmerer, 1985]. The use of formal notations and tools during this phase ensures a rigorous approach to requirements analysis in obtaining coherent, complete, correct, and unambiguous specifications.

We now shortly survey Z and the tools that a specifier can use to improve confidence in his requirements document.

2.1 The Z Notation

Z is a formal specification language based on set theory and first-order logic [Diller, 1990, Spivey, 1992]. A Z specification includes a set of entities, called *schemas* representing either abstract states of a system being specified or operations on them. The specification outlines an abstract model defined by means of typed entities and their related operations, expressed through a rich set of mathematical constructs.

A schema consists of a declarative part (the *signature*), where data structures, variables, and other schemas can be included, and a predicative part consisting of assertions, properties, and invariants. For instance, the following example shows four Z schemas specifying a Buffer. The invariant specified in the *Buffer* schema states that the buffer cardinality is less than some *limit*.

```
┌─ Buffer ──────────────────────────
│ buffer : ℙ(MESSAGE)
│ limit : ℕ
├───────
│ #buffer < limit
└───────────────────────────────────
```

```
┌─ Init_Buffer ─────────────────────
│ Buffer'
├───────
│ buffer' = ∅
│ limit' = 10
└───────────────────────────────────
```

$$
\begin{array}{|l}
\hline
\multicolumn{1}{l}{\textit{Produce}} \\
\Delta Buffer \\
m? : MESSAGE \\
\hline
\# buffer < limit - 1 \\
buffer' = buffer \cup \{m?\} \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\multicolumn{1}{l}{\textit{Consume}} \\
\Delta Buffer \\
m! : MESSAGE \\
\hline
m! \in buffer \\
buffer' = buffer \setminus \{(m!\} \\
\hline
\end{array}
$$

There are two basic kinds of schemas: those describing abstract data structures and those representing operations on them. The description of an operation is based on the concept of predicate on state transformation, that is a logic formula including a pre-state and a post-state. Schemas which "change" the state are marked by the presence of the Δ operator, while schemas keeping unchanged the current state of the system are marked with the Ξ operator. Initialization schemas allow the definition of only post-states by simply providing primed signatures in their declarative part (see the *Init* schema definition in the Buffer example). The Z description of abstract data structures is given by means of schemas in which no Δ or Ξ inclusions are present.

Variables ending with "?" represent input data, while those ending with "!" represent output. Furthermore, primed variables represent the state after an operation whereas unprimed ones denote the before state.

A major feature of Z is the *schema calculus*, that is a way of composing schemas. It uses some special symbols describing how different parts of the specification document can be combined together (sequential composition, parameter passing, combination through logical connectives, etc.). For a detailed description of the Z notation the reader should refer to [Spivey, 1992].

2.2 The Role Of Tools

We concentrate on the process that starts from a formal requirements document and produces a validated specification. Ideally, a number of tools can help the requirements engineer during such a process:

- Pretty printers and text formatters, to improve readability and to provide indexing. For instance, Z has a LaTeX-based syntax, that immediately compiles with TeX.
- Type checkers, to verify consistency in a specification document containing typed entities. For instance, fuzz is a type checker of Z documents written in LaTeX[Spivey, 1988b].
- Theorem provers, to prove formal properties improving the confidence in their correctness [Bowen & Gordon, 1994, Bloesch *et al.*, 1995].

- Model Checkers to analyze state machines by exhaustive search of their state spaces; for instance, Nitpick is a model checker for a subset of Z [Damon & Jackson, 1996].
- Animators to compile and execute specification documents written in non-executable formal languages. Animators support rapid prototyping, but are also helpful in proving that a specification is really implementable; some sequential animators for Z are described in [Dick *et al.*, 1989, Sterling *et al.*, 1996].
- Test case generators to generate test suites directly from specifications, that then can be used by programmers to test and debug the real implementation; a test case generator for Z is described in [Richardson *et al.*, 1992].

Pretty printers and type checkers are the first tools a specifier can use to parse and check a Z document. Type checkers like Fuzz [Spivey, 1988b] or ZTC [Jia, 1994] determine if a Z specification is both syntactically correct and correctly typed.

Unfortunately, parsers and type checkers are not sufficient to guarantee the correctness of the specification. They can detect mistakes like misusing a variable but they cannot reveal semantical inconsistencies. For instance, consider the *Buffer* specification and the following new operation:

$$
\begin{array}{|l}
\hline
\textit{OverProduce} \\
\hline
\Delta \textit{Buffer} \\
m? : \textit{MESSAGE} \\
\hline
\# \textit{buffer} = \textit{limit} - 1 \\
\textit{buffer}' = \textit{buffer} \cup \{m?\} \\
\hline
\end{array}
$$

OverProduce inserts a message in the buffer when its cardinality is $limit - 1$. The schema is clearly inconsistent with the *Buffer* schema (in fact it does not satisfy the invariant since *buffer* could reach value *limit*) even if the type checker is unable to reveal violation. Consider again the *Produce* operation: suppose the specifier forgot to put ' after the first *buffer*: the checker successes in checking the schema and does not report any errors.

The analysis of types cannot alone justify the effort of formalizing a specification. There must be some means of investigating details of behavior [Jackson, 1994]. Specification verification is a process that should be used to check whether a specification is correct in some sense [Barden *et al.*, 1994].

In order to verify the product quality, theorem provers are very important tools, because a formal document should be proved correct with respect to the requirements specification. Theorem provers allow automatic and detailed analysis of specification documents. More specifically, provers for Z provide automatic computation of preconditions of operations checking whether the operations are feasible, or if there is consistency between pairs of schema predicates. For instance, the calculus of the precondition for the operation *OverProduce* shows that the specification is inconsistent [Wordsworth, 1992]: in fact,

pre *OverProduce* is $\# \textit{buffer} = \textit{limit} - 1 \wedge \# \textit{buffer} < \textit{limit} - 1$

Theorem provers also help to reveal what is implicit in the specification: a fully formal proof gives a much higher level of confidence in the specification, but it is not always worthwhile doing it. The Z/Eves system [Saaltnik, 1989] provides a theorem prover (Eves) able to deal with Z documents: Z code is translated in the first order predicate calculus supported by Eves; a drawback of this prover is the amount of machine time taken to compute proofs [Bowen & Gordon, 1995]. The HOL theorem prover environment has been used for analyzing Z documents as well: in this case Z code has to be translated into higher order logic (HOL) before being processed. Though HOL system has better performance, it is less automatic than Eves: more user interaction is requested to complete proofs.

Unfortunately, the effort needed to formally prove a property of a specification is far greater than the one needed to write the whole specification [Barden *et al.*, 1994]. Automatic provers usually need much help from specifier that often must suggest the way the prove should progress. Guiding a theorem prover calls for a mathematical sophistication (and investment of time) that most users do not have, so these tools have been mostly confined to experts working in areas that can afford the cost, such as safety-critical software [Jackson, 1994].

Model checking calls for no user guidance (like type checking), but like theorem proving, it can answer quite sophisticated queries. However, model checking works on only finite state machines and most software systems have infinitely many states. The rationale behind why theorem proving has been primary approach for reasoning about software systems is that software systems are, in general, infinite state machines. In [Jackson, 1994] model checking is successfully used on software systems with infinite states: the idea is to build an abstraction in which a small number of abstract models represents a large (possibly infinite) set of concrete models. For instance, the type *Int* can be abstracted to $\lbrace\!| \, neg, zero, pos \, |\!\rbrace$. Abstraction can be used to automate the proof of some simple theorems usually proved by hand.

In order to state a property that guarantees a safe behavior of the buffer, for instance "*a message produced will eventually be consumed*", we need to specify the buffer's dynamic behavior. Z declarative semantics does not permit to define a concept of "history", so a property like the one defined above cannot be defined. In fact, no existing tools for Z can perform analysis on dynamic aspects of the system being specified.

A language derived from Z, namely Object-Z [Carrington *et al.*, 1989], allows temporal properties to be specified, introducing a trace-based semantics. We are here interested in original Z, so that we can continue to use all the existing tools and techniques.

3 A CONCURRENT SPECIFICATION: A LIFT SYSTEM

In order to explain our approach we use a well known example: the Lift System [Evans, 1994b, Richardson *et al.*, 1992, Cuellar *et al.*, 1994, Garzotto *et al.*, 1987].

3.1 Informal Requirements Specification

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

3.2 Formal Requirements Specification

In [Evans, 1994b] the requirements for a lift system are specified using Z; a number of dynamic properties are stated using some Unity-like logic constructs [Chandy & Misra, 1988]. Although the use of Unity logic allows one to capture dynamic properties not directly expressible in Z, the intended operational semantic model is strongly underspecified and the analysis method is unclear. Instead, starting with a specification very similar to the original, we will formalize both an operational semantics and a logic suitable to reason on it. We start the specification introducing a number of *free types*, namely primitive types defined by enumeration.

$$
\begin{array}{l}
DIRECTION ::= up \mid down \\
STATE ::= stopped \mid moving \\
DOOR ::= open \mid closed \\
REQUEST\_TYPE ::= up\_request \mid down\_request
\end{array}
$$

A lift can be defined by its position, direction, state, and door state:

$$
\begin{array}{|l}
\hline
\textit{Lift} \\
position : \mathbb{N} \\
direction : DIRECTION \\
state : STATE \\
door : DOOR \\
\hline
\end{array}
$$

The following schema describes a lift system, which includes a lift and some records of requests:

$$
\begin{array}{|l}
\hline
\textit{Lift\_System} \\
Lift \\
requests : \mathbb{P}(REQUEST\_TYPE \times \mathbb{N}) \\
\hline
\end{array}
$$

requests is the set of requests indicating their type and floor.

The initialization operation schema is:

$$
\begin{array}{|l}
\hline
\textit{Init\_Lift\_System} \\
Lift\_System' \\
\hline
position' = 1 \\
direction' = up \\
state' = stopped \\
door' = open \\
requests' = \emptyset \\
\hline
\end{array}
$$

The operation schema *Make_Requests* adds a new request to the *requests* set.

$$
\begin{array}{|l}
\hline
\textit{Make\_Requests} \\
\Delta \textit{Lift\_System} \\
r? : \textit{REQUEST\_TYPE} \\
f? : \mathbb{N} \\
\hline
\textit{requests}' = \textit{requests} \cup \{(r?, f?)\} \\
\hline
\end{array}
$$

The schema describing the moving of the lift up (a single schema in [Evans, 1994b]) is here divided up in two schemas, namely *Move_Up_Up* and *Move_Down_Up*. In this way we avoid $\vee$ operator in schemas and obtain simpler rules in the semantic mapping (see Sect.4.2).

Schema *Move_Up_Up* describes the moving up of the lift when up requests are present above the lift's current position; it can be applied only when the lift is already moving "up".

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

Operation schema *Move_Down_Up* defines the operation of changing the direction of the moving from down to up if no down requests are present under the lift position and there is at least one request for the lift above its current position.

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

Move_Down_Down and *Move_Up_Down* are the specular schemas for the moving of the lift down, which 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 having served some requests.

3.3 Formal Analysis

Given the above formal document, do suppose we intend to analyze it, looking for possible errors or inconsistencies. We need to "interpret" the document, especially if we are interested in properties of its intended dynamics.

In order to express properties on the dynamic behavior of the system, Unity-like logic constructs are introduced. Liveness (namely "a good thing will eventually occur") or safety (namely "a bad thing never happens") properties can be expressed in this way.

Properties are predicates (as the ones in the operation schemas) built on using some logic operators ($\wedge$, $\vee$, $\neg$, $\Leftrightarrow$, $\Rightarrow$) and Unity constructs (we will formally define the new logic constructs and analyze the properties given above in Sect.4.4).

Example We state some properties about the Lift system [Evans, 1994b]

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

The first theorem states that when the door is open, the state of the lift is stopped. The second one ensures that, when the lift arrives at a floor where requests are presents, it opens its door. The third one means that when the lift is moving up and there are requests above its position, it will eventually satisfy them on arriving at the floors requested.

Question is: are these properties true or false given the Z specification? Answering this kind of questions is the goal of the formal analysis we describe in the next sections.

4 SPECIFYING THE DYNAMICS OF Z SPECIFICATIONS

The elementary components of any Z document are State schemas and Operation schemas. This means that we can see a Z specification as a pair: $< S, O >$ where S is the set of the State schemas and O the set of the Operation schemas. Semantically, a State schema s in S can be seen as the set of all its possible instantiations [Spivey, 1988a]. The standard Z semantics [Spivey, 1988a, Brien & Nicholls, 1992, Breuer & Bowen, 1994] is declarative and does not offer any formalization for concurrency. For this reason, we have defined an operational semantics based on a chemical model.

For simplicity and conciseness, here we consider a restricted version of Z; we specify such a fragment using Z itself, so following the Z tradition [Spivey, 1988a, Gardiner *et al.*, 1990]. Moreover, because of the concurrent interpretation of Z that we are going to give, we assume that all variables not explicitly mentioned in the postconditions of an operation schema need not be invariant (that is: other operations can concurrently modify them). This assumption is needed in our interpretation and allows concurrency of the operations. In some papers on analysis of Z documents the assumption is exactly the opposite: "Variables not mentioned in the schemas are considered unchanged": e.g. [Richardson *et al.*, 1992]; however, this is not standard Z either.

We remark that our assumption, together with the suppression of the invariants $Lift' = Lift$ from *Make_Requests* operation and the invariant $Requests' = Requests$ from *Move_Up_Up* and *Move_Down_Up* allows concurrency among operations acting on the *Lift* and operations acting on the *Requests*.

In order to better explain our operational semantics, now we briefly introduce the Chemical Abstract Machine that is the basis of our interpretation of Z.

4.1 The Chemical Model

In the Chemical Abstract Machine model [Berry & Boudol, 1992, Boudol, 1993] *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

  $condition(m_1, m_2..m_k) \longrightarrow m_1, m_2..m_k \Rightarrow m'_1, m'_2..m'_l$

  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

  $condition(M_1, M_2..M_k) \longrightarrow \{\!|\ M_1, M_2..M_k\ |\!\} \Rightarrow \{\!|\ M'_1, M'_2..M'_l\ |\!\}$

- **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 the Chemical Abstract Machine two rules can fire concurrently if they do not need the same molecules to react on; hence, several rules can progress simultaneously on a solution. If two rules conflict, in the sense that they "consume" the same molecules, one of them is chosen non-deterministically to react.

A CHAM has some similarities with a Z specification. The state of a Z specification is global, even if it can be seen as a composition of many sub-states, corresponding to the state schemas involved in the specification. Using the CHAM the global solution is composed of several completely independent sub-solutions. Moreover, in both Z and the CHAM the firing of actions cannot be enforced; there are only enabled or not enabled operations (or rules). In fact Z operations are enabled when their preconditions are satisfied and CHAM rules are enabled when conditions are true and molecules in the left-hand side of the rule are present.

We consider a *fair* Cham where repeatedly enabled rules will eventually be fired: this assumption helps in proving properties defined using Unity logic constructs (Sect. 4.4).

4.2 An Operational Semantics For Z

We give a CHAM interpretation of Z specifications which allows us to deal with concurrent dynamics. Intuitively, an instance of a state schema is associated to 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 premises and consequences are solutions composed of pre and post conditions of the operation.

We describe the formalization of such an interpretation using the Z language itself.

A molecule is a tuple of a name, a type and a value:

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

A solution is a bag of molecules:

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

and a rule is composed of a conditional part that define the applicability of the rule and of two solutions that indicate molecules to delete and add to the state solution:

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

We will call the first $SOLUTION$ "pretuples" and the second "posttuples" in order to avoid ambiguity.

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.

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

$$Fsem : SCHEMA\_INSTANCE \longrightarrow SOLUTION$$

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

$Fsem\_op$ associates a rule to an operation schema [1]:

$$Fsem\_op : SCHEMA\_OP \longrightarrow RULE$$

$Fsem\_Op$ associate to pre and postcondition of a schema different part of the rule:

- Every Z schema postcondition that specifies the removal of an element from a set or bag is mapped to a pretuple of the rule (molecule to be deleted).

[1] A similar function has been defined for initialization operation, where no preconditions are present.

- Every postcondition that specifies the insertion of an element in a set or bag is mapped to a posttuple of the rule (molecules to be added).

- Every Z precondition that defines a membership ($\in$, $\sqsubseteq$) is mapped to a pretuple (a removal) and also to 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 seen as a removal and reinsertion.

- Postconditions containing mathematical operators ($+$, $-$,...) on naturals are encoded deleting one molecule and adding the molecule updated.

  Example: $x' = x + 1$ is seen 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 added again as already described [2].

  Example: $x < 5$ is seen as $v < 5 \longrightarrow (x, \mathbb{N}, v) \Rightarrow (x, \mathbb{N}, v)$

Now, following the interpretation of the Chemical Machine, rules can fire concurrently when they are enabled by conditions and non-conflicting on pretuples molecules.

It is possible to define many other functions to describe, for instance, when rules, i.e. operation schemas, can fire concurrently: it usually depends on their postconditions.

Example To clarify this interpretation let us consider again the Lift system.

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

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

The state schema instance obtained by the application of the operation is the same solution presented above.

The rule associated to the operation schema *Make_Requests* has the following posttuple (no premises):

$$(requests, \mathbb{P}(REQUEST\_TYPE \times \mathbb{N}), (r?, f?))$$

The rule corresponding to the operation *Move_Up_Up* (*Move_Down_Up* is similar) has the following condition $f > pos$, as premises:

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

and as consequences:

[2]This is done following the chemical semantics where conditions can only be stated on the local molecules involved in the rule [Boudol, 1993].

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

Rules which do not act on the same molecules can react simultaneously: hence, following our interpretation, enabled operation schemas which do not modify the same variables can react in parallel.

We remark that both sets and bags can be decomposed in several molecules, thus increasing potential concurrency. For example, the set *requests* is decomposed in many molecules, one for each request: in this way it is possible to have concurrency among different applications of the operation *Make_Requests*.

4.3 An Abstract Execution Model And Its Logic

We define now an *execution model*, namely a way of abstractly executing a Z specification document, and a Unity-like logic [Chandy & Misra, 1988] to reason on properties exhibited by such a model.

The execution model is defined on the semantics just described; it represents the unfolding of the application of the rules of the operational semantics.

For every state schema s an abstract execution tree 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-like logic constructs using Z as meta-language, we introduce the specification of the execution tree:

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

where

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

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 (s, seq), there must hold the post-conditions of all the operations in the operations set applied to reach that node (sequence structure help to keep link between nodes and operations set).

4.4 A Logic For Expressing Dynamic Properties

In order to be able to reason on dynamic properties, like deadlock and starvation, we introduce our logic borrowing a few constructs from the Unity logic. Properties are expressed as predicates related by Unity logic operators; predicates now have chemical semantics and are interpreted as chemical solutions. We can state a predicate p is valid on a state solution s if all the molecules in the chemical interpretation of p are also in s.

We briefly list the logic constructs we introduce and their semantics:

- 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 the whole subtree of that node.
- **Invariant** p says that p is true forever. That is, for every node of the execution tree p is valid.
- p **ensures** q means 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 in 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 N, then in each branch through N there is a node M below N where q holds.

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

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

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.

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

The formalization of *ensures* derives from *unless*: in fact, p **ensures** q if p **unless** q and exists a set of operations that applied to a state where p is valid generates a state where q holds (that is: eventually q becomes true because of the fairness of the chemical model we have introduced).

Having described the meaning of these logic predicates on the operational model, we can define and reason on dynamic properties on the abstract concurrent interpretation of Z specifications.

Example We specify and verify dynamic requirements with the help of the new constructs. We have already defined three properties on the Lift specification in Sect.3: now we analyze the specification trying to prove them.

Theorem 1: **Invariant** $door = open \Rightarrow state = stopped$

Proof: To prove an invariant property we prove that the property holds after the initialization and that it is stable.

The first condition is true: in fact, the *Init_Lift_System* operation assigns the value *open* to the *door* variable and *stopped* to the lift state. $Stable(p)$ is p *Unless* false: that is, for every operations set applicable to a state where $p \wedge \neg$ false $= p \wedge$ true $= p$ is valid, the next state in the execution will satisfy $p \vee$ false $= p$.

In our particular case, consider p as the property $door = open \Rightarrow state = stopped$; given a state where p holds, all the operations are enabled, in fact p may be rewritten as $door = closed \vee state = stopped$. The operation *Make_Requests* does not modify the lift state while other specification operations change it accordingly to the invariant. Moreover, operations sets to be executed can contain only one operation acting on the lift system (all the operations on the lift need to modify the same molecules) plus several *Make_Requests*. Then we can conclude that the invariant holds.

Theorem 2: $(t, f) \in requests \wedge f = position \wedge door = closed$ **ensures** $door = open$

Proof: To prove p **ensures** q (where p is $(t, f) \in requests \wedge f = position \wedge door = closed$ and q is $door = open$) we must prove p **unless** q and the existence a set of operations that applied to a state where $p \wedge \neg\ q$ is valid generates a state where q holds. We now prove p **unless** q; for every enabled operations set on the solution containing molecules:

$(requests, \mathbb{P}(REQUEST\_TYPE \times \mathbb{N}), (t, f)), (position, \mathbb{N}, pos), (door, DOOR, closed)$

the application has to lead to a state where molecule $(door, DOOR, open)$ is present or the previous molecules are still in the solution (this is the **unless** formalization of our concurrent execution model).

Considering our Z specification, we notice that the only enabled operations on an instance state containing those molecules are *Make_requests* (which only adds new requests molecules without modifying the lift state) and *Open_Door* which exactly opens the lift door. Hence, p **unless** q holds.

We now prove the second part of our theorem, that is: given a state where $p \wedge \neg\ q$ holds, there exists an enabled operations set applicable, which leads to a state in which q holds. In our specification such a set is composed of operation *Open_Door* and some (or none) *Make_Requests*. Assuming the fairness condition given in Sect.4, we can state that the set will eventually be applied. This completes the proof.

Theorem 3: $(t, f) \in requests \wedge f > position \wedge direction = up$
$\wedge\ state = moving \wedge door = closed$ **leads_to** $position = f$

Proof: in order to prove a p **leads_to** q property we must ensure that at least one chain of operations sets does exist starting from a state containing the molecules:

$(requests, \mathbb{P}(REQUEST\_TYPE \times \mathbb{N}), (t, f)), (direction, \mathbb{P}\, DIRECTION, up),$
$(position, \mathbb{N}, pos), (state, \mathbb{P}\, STATE, moving), (door, \mathbb{P}\, DOOR, closed)$, with $f > pos$

leading to a state containing $(position, \mathbb{N}, f)$.

The operations sets in the chain are composed of the operation *Make_Requests* and the operation *Move_Up_Up*, which continuously move the lift up until requests are present above the lift position.

5 ANIMATING A Z SPECIFICATION

A formal requirements document provides a precise and rigorous description of the system being developed. Its aim is to define abstract properties of the system, describing *what* the system has to do, and not *how* to do it [Hayes & Jones, 1989].

We stress the fact that Z has non-executable semantics. Z specifications are declarative since the specifier is allowed to declare not computable entities, like infinite sets or not computable functions, and to specify properties and operations on them.

However, the specifier often could desire to have a prototype of the system which can be tested and whose dynamic properties can be directly inspected by either the specifier himself or even the customer. Hence, there is a problem. A notation like Z is typically used to make simple the proof of important properties of the system studied, but it cannot help to establish a very important property from a practical point of view: the executability (implementability) of the specification [Fuchs, 1992]. Moreover, specifiers who want to experiment intermediate specifications in order to gain feedback from a customer are deprived from using the computer to test and validate the current version.

5.1 Animation By Prolog

The technique of validation by *animation* has been introduced in order to overcome the difficulty of obtaining a prototype from a non-executable specification language. Specifically, several approaches have been used to transform a Z specification into an executable form, using different programming languages and different methodologies [Sterling *et al.*, 1996]. 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. Executable languages are less expressive than non-executable ones, since their functions must be computable and their domains must be finitely representable. 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 [Breuer & Bowen, 1994].

Many researchers have directed their efforts toward the development of methods for mapping Z specifications to Prolog programs. Indeed, using logic programming appears to be a natural way to animate Z documents, since the practice shows that most predicates found in Z documents have an easy implementation in terms of Prolog clauses. For instance, Knott and Krause in the SuZan project developed a basic strategy for manual translation of Z to Prolog and produced a library of Prolog predicates implementing standard Z constructs [Knott & Krause, 1992]. Such an approach is called *generate and test* because it is based on generating all values which could possibly satisfy the abstract state described in the specification document, and testing them using the specification properties and invariants. Unfortunately, it can handle any specification document, but is impractical: it suffers from a problem of combinatorial explosion of solutions and often no satisfactory answer from a query is obtained, because there are infinite answers or the program loops too much time to find at least one solution.

To overcome these difficulties we can somewhat restrict the Z notation to be used in our specification document, in order to have a subset of the language "almost directly" executable or compilable [Valentine, 1995, Doma & Nicholl, 1991]. This is the *procedural* method [Diller, 1990], which typically also defines a style and an order in which Z constructs must be used. In fact, the specification must be written bearing in mind a successive animation, such that it is possible to have a straightforward translation into procedures of the target language. The restriction imposed on the Z language are paid back in terms of efficiency of execution of the obtained prototype.

Our approach to animation can be classified as procedural since the original specification has first to be refined in a simplified notation we call *Zel*. The main novelty of our proposal lies in the use of a coordination language, Shared Prolog [Ciancarini & Gaspari, 1996], as target language for the mapping of Z specifications. Shared Prolog is an extension of Prolog with support for explicit parallelism and its semantics is also based on the chemical model [Ciancarini, 1991]. So the semantics we gave in Sect.4 for Z specifications, gives us a formal basis for the animation process which leads to the construction of a parallel/distributed prototype of the system being specified. The simplicity of the mapping from Z constructs to Prolog predicates is maintained but now the system has the capability of dealing with concurrent features of the specification.

A distributed prototype offers a more realistic model of the system we are developing, allowing one to test and inspect concurrent behaviors. Issue of efficiency is also addressed since the code obtained by the automatic translation of the refined specification can be executed over a network of workstations.

Unlike other animators that simply offer a way of executing single operations sequentially and step by step, our animator generates Shared Prolog code, which is an executable form of the Z specification able to generate a parallel trace.

5.2 A Parallel Animation Language

Shared Prolog is based on the combination of the shared dataspace coordination model (as in Linda) with logic programming computing (as in Prolog) [Ciancarini & Gaspari, 1996].

A program in Shared Prolog 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{tuple-1, ... ,tuple-n}.
```

Another special goal defines some agents which share the tuple space (we also call agents "active tuples", since they are part of the tuple space):

```
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 non-deterministically chosen and executed.

5.3 Animating A Z Specification With SP

According to the specification style described in Sect.4, 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; two (or more) operations are concurrent if they are enabled at the same time. This implies that in the animation, several operations can be executed at the same time. In fact, the animation of a requirements specification document using a concurrent language has the immediate effect of capturing and making observable the concurrent characteristics of the system being specified.

In order to simplify the translation we have restricted Z to a number of constructs (we call Zel - Z elementary - the resulting notation). In the following table is summed up what can be used in Zel:

Predefinite types	$\mathbb{N}, \mathbb{Z}$
Type constructors	$\mathbb{P}$, bag, seq, $\times$
Operators on set	$\in, \notin, \cup, \cap, \subset, \subseteq$
Operators on multiset	$\sqsubseteq, \uplus, \uminus$
Operators and functions on sequences	in, $\frown$, *head*, *tail*, *front*, *last*
Arithmetic operators	+ , - , * , div , mod
Relational operators	$=, \neq, >, <, \geq, \leq$
Logical operators	$\neg$, $\wedge$
Quantifiers	$\exists$
Schema connectives	$\vee, \wedge, \Rightarrow, \fatsemi, \gg$
Variable decorations	! , ? , $'$

The restrictions we have introduced concern types and use of variables, form of predicates, and usability of some quantifiers.

5.3.1 *Types*

Types can be defined in Zel as follows:

- *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*.

5.3.2 *Variables*

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 to 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.

5.3.3 *Predicates*

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 through 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.

5.4 From Zel To SP

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 now describe the ideas we used to build the translator.

Each simple schema not defined by a schema calculus expression in the Zel document is transformed in 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.

To each variable defined in the predicative part of a Zel schema corresponds a unique SP tuple in the blackboard, which is obtained concatenating the name of the variable and of 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.

Example Suppose that in a precondition we have a set membership predicate, and in the postcondition we update the set removing the tested element.

precondition	$v \in set$
postcondition	$set' = set \setminus \{V\}$
pattern	`{set(v)}` $\rightarrow$ ..

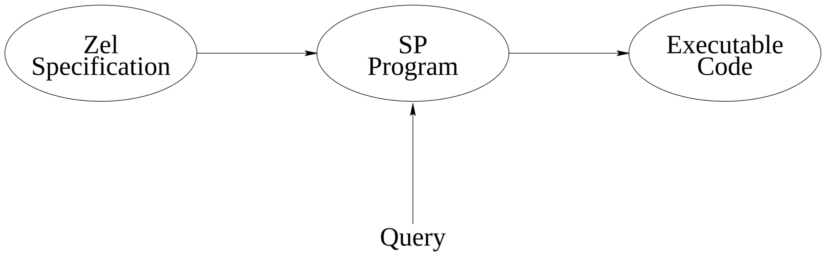
In the SP program this is equivalent to including a tuple in the in-guard of the corresponding pattern, with the meaning that it will be removed from the blackboard.

5.5 Querying The Animation

The SP program obtained through the transformation process has to be initialized before it can be executed. The initial state of the blackboard and the initial goal in fact cannot be obtained by automatic translation of the specification, but must be defined in the query.

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. Moreover, tuples representing input data for the animation must be added to the tuple space in the initial blackboard. 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.

The following diagram visualizes the animation process for a given specification:



6 ANIMATING THE LIFT SYSTEM

We describe now the animation of the lift system specification of the lift system introduced in Sect.3. The first step consists of refining the Z document. This amounts to refining and simplifying Z schemas into Zel syntax.

For instance, the following schemas describe the operation of moving the lift up, provided that there is at least one pending request above its current position either when the lift is going up or when there are no requests going down. Note how few changes have been made with respect to the original version given in Sect.3: they only concern variables defined as $\mathbb{P}\ \mathit{sometype}$ and i/o variables.

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

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad Move\_Down\_Up \quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad}} \\
\Delta LiftSystem \\
\hline
\exists\, tf : REQUESTTYPE \times \mathbb{N};\ t : REQUESTTYPE; \\
f, p1, p2 : \mathbb{N};\ d : DOOR;\ c : DIRECTION \bullet \\
\qquad (tf \in requests \land tf = (t, f) \land \\
\qquad p1 \in position \land f > p1 \land p2 = p1 + 1 \land \\
\qquad d = closed \land d \in door \land \\
\qquad c = down \land c \in direction \land \\
\qquad \neg\, (\exists\, ug : REQUESTTYPE \times \mathbb{N};\ u : REQUESTTYPE;\ g : \mathbb{N} \bullet \\
\qquad\qquad (ug \in requests \land ug = (u, g) \land g \leq p1)) \land \\
\qquad position' = position \setminus \{p1\} \cup \{p2\} \land \\
\qquad direction' = \{up\} \land \\
\qquad state' = \{moving\} \land \\
\qquad door' = \{closed\}) \\
\hline
\end{array}
$$

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)}
```

The initial tuple space is obtained from the *InitSystem_Lift* schema defined in the specification document. It can be modified to satisfy a particular query to be animated.

In the following example, we put into the tuple space a set of tuples representing a sequence of service requests for the lift, and an initial condition for the lift.

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

Moreover, at least one agent has to be activated by the initial goal:

```
initial_goal (op_liftsystem).
```

The SP program is then compiled and executed. Trace (a) below shows the report of an animation of the system for the above sequence of service requests, including the final state of the lift.

```
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
```

trace(a)

```
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
```

trace(b)

A trace helps the specifier to debug a specification. He should aim to check aspects of the system that have not been fully explored or to correct an improper behavior due to mistakes in the specification.

For instance, trace b) shows the animation report if we "forget" to include in the *Move_Up* schemas the test concerning the shutting of lift door. The trace shows that a lift can move to a new floor to satisfy a request *without closing its door*; 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 using 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 [Kemmerer, 1985]

The main advantages of the animation are in terms of accelerating the specification validation with respect to the user requirements and increasing the degree of trust the specifier can have in the system being built.

7 REFINING THE ANIMATION

Since the process of obtaining an animation is automatic, it is possible to rapidly build a prototype of the system even if the original specification is incrementally modified in order to test different or new features. The feedback coming from the execution of the animation can be used to improve or emend the original specification document.

For instance, we can refine our example to make the lift system more realistic increasing the number of lifts and specifying a scheduling policy for user requests. This can be easily accomplished by parameterizing opportunely the operations and the state of the original lift system. The modified schema looks as:

$$
\begin{array}{|l}
\hline
\textit{LiftSystem} \\
\textit{position} : \mathbb{P}(\textit{LIFT} \times \mathbb{N}) \\
\textit{direction} : \mathbb{P}(\textit{LIFT} \times \textit{DIRECTION}) \\
\textit{state} : \mathbb{P}(\textit{LIFT} \times \textit{STATE}) \\
\textit{door} : \mathbb{P}(\textit{LIFT} \times \textit{DOOR}) \\
\textit{requests} : \mathbb{P}(\textit{LIFT} \times \textit{REQUESTTYPE} \times \mathbb{N}) \\
\textit{Lifts} : \mathbb{P}\ \textit{LIFT} \\
\hline
\end{array}
$$

where $\textit{LIFT} ::= a1 \mid a2 \mid a3$ defines the available lifts.

The following schema refines the corresponding one presented in the preceding section.

$$
\begin{array}{|l}
\hline
\textit{Move\_Up\_Up} \\
\Delta \textit{LiftSystem} \\
\hline
\exists\, tf : \textit{LIFT} \times \textit{REQUESTTYPE} \times \mathbb{N}; \\
t : \textit{REQUESTTYPE};\ e : \textit{LIFT};\ f, p1, p2 : \mathbb{N};\ d : \textit{DOOR}; \\
c1, c2 : \textit{DIRECTION};\ s1, s2 : \textit{STATE} \bullet \\
(tf \in \textit{requests} \land e \in \textit{Lifts} \land tf = (e, t, f) \land \\
(e, p1) \in \textit{position} \land f > p1 \land p2 = p1 + 1 \land \\
(e, d) \in \textit{door} \land d = \textit{closed} \land \\
(e, c1) \in \textit{direction} \land c2 = \textit{up} \land \\
(e, s1) \in \textit{state} \land s2 = \textit{moving} \land \\
\textit{position}' = \textit{position} \setminus \{(e, p1)\} \cup \{(e, p2)\} \land \\
\textit{state}' = \textit{state} \setminus \{(e, s1)\} \cup \{(e, s2)\} \land \\
\textit{direction}' = \textit{direction} \setminus \{(e, c1)\} \cup \{(e, c2)\}) \\
\hline
\end{array}
$$

The operation *Move_Up_Up* is now enabled when one of the active lifts has to serve a request at a floor above its current position; state, direction, and position of the moving lift are consequently updated. User requests are scheduled by a controller which assigns each pending request to the lift nearest to the requested floor.

$$
\begin{array}{|l}
\hline
\textit{Controller} \\
\Delta \textit{LiftSystem} \\
r? : \mathbb{P}(\textit{REQUESTTYPE} \times \mathbb{N}) \\
\hline
\exists\, s : \textit{REQUESTTYPE} \times \mathbb{N};\ r1 : \textit{LIFT} \times \textit{REQUESTTYPE} \times \mathbb{N}; \\
t : \textit{REQUESTTYPE};\ e1 : \textit{LIFT};\ a, f, a1, p1 : \mathbb{N} \bullet \\
(\{s\} = r? \land s = (t, f) \land \\
(e1, p1) \in \textit{position} \land a = p1 - f \land a1 = a * a \land \\
\neg\, (\exists\, e2 : \textit{LIFT};\ b, b1, p2 : \mathbb{N} \bullet \\
((e2, p2) \in \textit{position} \land b = p2 - f \land b1 = b * b \land b1 < a1)) \land \\
r1 = (e1, t, f) \land \\
\textit{requests}' = \textit{requests} \cup \{r1\}) \\
\hline
\end{array}
$$

A change of scheduling policy of requests involves the redefinition of the above schema; the consequences over the whole system can be inspected through the execution trace. Concurrency of operations of different lifts can be appreciated since concurrently enabled operations are concurrently executed. What follows is the trace of the animation of a lift system composed of three lifts and a scheduler assigning requests as specified above. The final state of the system with a set of pending requests is also showed.

Controller	**Lift a1**	**Lift a2**	**Lift a3**
`req upRequest,1 assign a1`			
	`lift a1 open 1`		
`req upRequest,8 assign a1`			
	`lift a1 close`		
	`lift a1 up from 1 to 2`		
	`lift a1 up from 2 to 3`		
	`lift a1 up from 3 to 4`		
	`lift a1 up from 4 to 5`		
	`lift a1 up from 5 to 6`		
	`lift a1 up from 6 to 7`		
	`lift a1 up from 7 to 8`		
	`lift a1 open 8`		
`req upRequest,4 assign a2`			
		`lift a2 up from 1 to 2`	
		`lift a2 up from 2 to 3`	
		`lift a2 up from 3 to 4`	
`req upRequest,3 assign a2`			
`req upRequest,5 assign a2`			
`req upRequest,3 assign a2`			
`req upRequest,1 assign a3`			
`req upRequest,6 assign a1`			
`req upRequest,4 assign a2`			
`req upRequest,3 assign a2`			
`req upRequest,1 assign a3`			
`req upRequest,2 assign a3`			
`req upRequest,8 assign a1`			
`req upRequest,4 assign a2`			
`req upRequest,3 assign a2`			
	`lift a1 close`		
		`lift a2 open 4`	
		`lift a2 close`	
			`lift a3 open 1`
			`lift a3 close`
	`lift a1 open 8`		
	`lift a1 close`		
	`lift a1 down from 8 to 7`		
		`lift a2 open 4`	
		`lift a2 close`	
			`lift a3 open 1`
			`lift a3 close`
			`lift a3 up from 1 to 2`
		`lift a2 open 4`	
		`lift a2 close`	
			`lift a3 open 2`
	`lift a1 down from 7 to 6`		
	`lift a1 open 6`		
		`lift a2 up from 4 to 5`	
		`lift a2 open 5`	
		`lift a2 close`	
		`lift a2 down from 5 to 4`	

```
                                        lift a2 down from 4 to 3
                                        lift a2 open 3

requests [(a2,upRequest,3),(a2,upRequest,3),(a1,upRequest,6),
         (a2,upRequest,3),(a3,upRequest,2),(a2,upRequest,3)]
position [(a3,2),(a1,6),(a2,3)]
door [(a3,open),(a1,open),(a2,open)]
direction [(a3,up),(a1,down),(a2,down)]
state [(a3,stopped),(a1,stopped),(a2,stopped)]
```

The trace given above shows that the controller correctly assigns requests to the lifts which concurrently move from floor to floor serving the formerly scheduled requests. Note how the arrival of a request to the floor where a lift has already stopped causes the re-opening of the doors. This and other strange behaviors, if found, can be corrected by furtherly refining the specification.

8 CONCLUSIONS AND FUTURE WORK

When formal requirements specification is the first phase in a software development process based on formal methods, the possibility of analyzing and testing relevant properties of requirements (including the implementability/executability property) early in the process avoids to carry costly mistakes over the next phases.

We have introduced two methods for the analysis of specification dynamics: the first one is more abstract, and it is able to deal with global behavioral components, whereas the second is more concrete, and able to produce a high level prototype. The prototype itself can be used to study the specification behavior, producing test cases which can be used in subsequent phases.

The main novelties of our approach are the introduction of a chemical operational semantics for Z, which offers the formal basis to analyze dynamic properties of reactive systems, and the automatic development of a concurrent animation using a prototyping language able to highlight concurrent behaviors.

Our work can be closely compared with the one exposed in the classic paper [Kemmerer, 1985]. The main difference is that our method aims clearly at the requirements specification document, whereas Kemmerer was apparently more interested in design. We believe it is important to clearly distinguish requirements specification from design specification [Ciaccia *et al.*, 1996], and to introduce and study methods and tools for the analysis, testing, and verification of the requirements specification document.

Some formal approaches integrate Z with other notations; for instance, Benjamin integrates Z schemas with CSP notation [Benjamin, 1989]. CSP is used to specify abstract operations, while Z is used for detailing some design features of abstract state and data. The integration is very low level and not formally specified. In [He, 1995] Petri Nets are used to formalize control flows, causal relations, and dynamic behavior of systems statically specified using Z; there is no formalization of the interaction between the two notations.

[Evans, 1994a] offers 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 visualize Z specified systems, yet we think this approach is not giving a formal semantics basis for Z but only a visualizing method.

In [Richardson *et al.*, 1992] a formalism based on temporal logic is used to integrate Z schemas with dynamic properties. The use of temporal logic offers good starting points to the study of the dynamics of Z specifications, however the integration is not supported by a formal semantics.

In this sense something more formal has been done for Object-Z [Carrington *et al.*, 1989]: 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 [Lamport, 1994]. A simpler approach has been suggested by Evans in [Evans, 1994b]. He uses a Unity like logic [Chandy & Misra, 1988] to formalize properties on the behavior of systems; an interleaving model with atomic operation interpretation is given but not formalized.

Our approach shows how the simplicity and conciseness of Unity logic constructs fit quite well with the operational semantics based on CHAM.

We have been using the notations and tools described in this paper in an undergraduate course on requirements engineering. Several student projects have been completed, and now we are analyzing such an experience to evaluate the overall approach. The major shortcomings our users have found concern the compiler tool: it lacks support for incremental debugging. Another aspect we are investigating is the formal relationship between the operational semantics and the original declarative semantics of Z. We are studying correctness and completeness of our semantics with respect to the original one.

Acknowledgments. The authors would especially like to thank Paolo Ciaccia for his 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

[Barden *et al.*, 1994] Barden, R., Stepney, S., & Cooper, D. (1994). *Z in Practice.* Prentice-Hall.

[Baumann & Lermer, 1995] Baumann, P. & Lermer, K. (1995). In: *Proc. 15th Conference on Foundation of Software Technology and Theoretical Computer Science*, (Thiagarajan, P., ed) volume 1026 of *Lecture Notes in Computer Science* pp. 62–79, Bangalore, India: Springer-Verlag, Berlin.

[Benjamin, 1989] Benjamin, M. (1989). In: *Proc. 4th Z Users Workshop (ZUM89)*, (Nicholls, J., ed) Workshops in Computing pp. 221–228, Oxford: Springer-Verlag, Berlin.

[Berry & Boudol, 1992] Berry, G. & Boudol, G. (1992). *Theoretical Computer Science,* **96**, 217–248.

[Bloesch *et al.*, 1995] Bloesch, A., Kazmierczak, E., Kearney, P., & Traynor, O. (1995). *Int. Journal on Software Engineering and Knowledge Engineering,* **5** (4), 599–618.

[Boudol, 1993] Boudol, G. (1993). In: *A Decade of Concurrency*, (deBakker, J., deRoever, W., & Rozenberg, G., eds) volume 803 of *Lecture Notes in Computer Science* pp. 92–123, Springer-Verlag, Berlin.

[Bowen & Gordon, 1994] Bowen, J. & Gordon, M. (1994). In: *Proc. 8th Z Users Workshop (ZUM94)*, (Bowen, J. & Hall, J., eds) Workshops in Computing pp. 141–167, Cambridge: Springer-Verlag, Berlin.

[Bowen & Gordon, 1995] Bowen, J. & Gordon, M. (1995). *Information and Software Technology,* **37** (5-6), 269–276.

[Breuer & Bowen, 1994] Breuer, P. & Bowen, J. (1994). In: *Proc. 8th Z Users Workshop (ZUM94)*, (Bowen, J. & Hall, J., eds) Workshops in Computing pp. 185–212, Cambridge: Springer-Verlag, Berlin.

[Brien & Nicholls, 1992] Brien, S. & Nicholls, J. (1992). Programming Research Group.

[Carrington *et al.*, 1989] Carrington, D., Duke, D., Duke, R., King, P., Rose, G., & Smith, G. (1989). In: *Formal Description Techniques (FORTE 89)* pp. 281–296, North-Holland.

[Chandy & Misra, 1988] Chandy, K. M. & Misra, J. (1988). *Parallel Programming Design.* Addison-Wesley.

[Ciaccia *et al.*, 1996] Ciaccia, P., Ciancarini, P., & Penzo, W. (1996). *Int. Journal on Software Engineering and Knowledge Engineering,* **(to appear)**.

[Ciancarini, 1991] Ciancarini, P. (1991). In: *Research Directions in High-Level Parallel Programming Languages*, (Banatre, J. & LeMetayer, D., eds) volume 574 of *Lecture Notes in Computer Science* pp. 110–125, Mont Saint-Michel, France: Springer-Verlag, Berlin.

[Ciancarini & Gaspari, 1996] Ciancarini, P. & Gaspari, M. (1996). *Computer Languages,* **(to appear)**.

[Cuellar *et al.*, 1994] Cuellar, J., Wildgruber, I., & Barnard, D. (1994). In: *FME'94: Industrial Benefit of Formal Methods* volume 873 of *Lecture Notes in Computer Science* pp. 639–658, Barcelona, Spain: Springer-Verlag, Berlin.

[Damon & Jackson, 1996] Damon, C. & Jackson, D. (1996). In: *Proc. TACAS '96*, (Margaria, T. & Steffen, B., eds) volume 1055 of *Lecture Notes in Computer Science* pp. 70–86, Springer-Verlag, Berlin.

[Dick *et al.*, 1989] Dick, A., Krause, P., & Cozens, J. (1989). In: *Proc. 4th Z Users Workshop*, (Nicholls, J., ed) Workshops in Computing pp. 71–85, Oxford: Springer-Verlag, Berlin.

[Diller, 1990] Diller, A. (1990). *Z: An Introduction to Formal Methods.* Wiley.

[Doma & Nicholl, 1991] Doma, V. & Nicholl, R. (1991). In: *VDM 91: Formal Software Development Methods*, (Prehn, S. & Toetenel, W., eds) volume 551 of *Lecture Notes in Computer Science* pp. 189–203, Noordwijkerhout: Springer-Verlag, Berlin.

[Evans, 1994a] Evans, A. (1994a). In: *Proc. 8th Z Users Workshop (ZUM94)*, (Bowen, J. & Hall, J., eds) Workshops in Computing pp. 269–281, Cambridge: Springer-Verlag, Berlin.

[Evans, 1994b] Evans, A. (1994b). In: *Proc. FME'94 Industrial Benefits of Formal Methods*, (Bertran, M., Denvir, T., & Naftalin, M., eds) volume 873 of *Lecture Notes in Computer Science* pp. 366–380, Springer-Verlag, Berlin.

[Fuchs, 1992] Fuchs, N. (1992). *IEE Software Engineering Journal,* **7** (5), 323–334.

[Gardiner *et al.*, 1990] Gardiner, P., Lupton, P., & Woodcock, J. (1990). In: *Proc. 5th Z Users Workshop*, (Nicholls, J., ed) Workshops in Computing pp. 3–11, Oxford: Springer-Verlag, Berlin.

[Garzotto *et al.*, 1987] Garzotto, F., Ghezzi, C., Mandrioli, D., & Morzenti, A. (1987). In: *Proc. 1st European Software Eng. Conf. (ESEC 87)* volume 289 of *Lecture Notes in Computer Science* pp. 180–190, Springer-Verlag, Berlin.

[Hayes, 1993] Hayes, I. (1993). *Specification Case Studies.* Prentice-Hall, 2 edition.

[Hayes & Jones, 1989] Hayes, I. & Jones, C. (1989). *IEE Software Engineering Journal,* **4** (6), 330–338.

[He, 1995] He, X. (1995). In: *Proc. 7th Int. Conf. on Software Engineering and Knowledge Engineering* pp. 173–180, Rockville, Maryland: Knowledge Systems Institute.

[Inverardi & Wolf, 1995] Inverardi, P. & Wolf, A. (1995). *IEEE Transactions on Software Engineering,* **21** (4), 373–386.

[Jackson, 1994] Jackson, D. (1994). In: *Proc. 2nd Int. Symp. of Formal Methods Europe (FME),* (Naftalin, M., Denvir, T., & Bertran, M., eds) volume 873 of *Lecture Notes in Computer Science* pp. 519–531, Barcelona, Spain: Springer-Verlag, Berlin.

[Jia, 1994] Jia, X. (1994). Institute of Software Engineering.

[Kemmerer, 1985] Kemmerer, R. (1985). *IEEE Transactions on Software Engineering,* **11** (1), 32–43.

[Knott & Krause, 1992] Knott, R. & Krause, P. (1992). In: *The Unified Computation Laboratory,* (Rattray, C. & Clark, R., eds) volume 35 of *IMA Conference Series* pp. 207–220, Oxford, UK: Clarendon Press.

[Lamport, 1994] Lamport, L. (1994). In: *Proc. 8th Z Users Workshop (ZUM94),* (Bowen, J. & Hall, J., eds) Workshops in Computing pp. 267–268, Cambridge: Springer-Verlag, Berlin.

[Richardson *et al.*, 1992] Richardson, D., Aha, S., & O'Malley, T. (1992). In: *Proc. 14th IEEE Int. Conf. on Software Engineering* pp. 105–118, Melbourne, Australia:.

[Saaltnik, 1989] Saaltnik, M. (1989). In: *Proc. Z User Workshop,* (Nicholls, J., ed) Workshops in Computing pp. 223–242, Oxford, UK: Springer-Verlag, Berlin.

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

[Spivey, 1988b] Spivey, J. (1988b). *The* fuzz *Manual.*

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

[Sterling *et al.*, 1996] Sterling, L., Ciancarini, P., & Turnidge, T. (1996). *Int. Journal on Software Engineering and Knowledge Engineering,* **6** (1), 63–88.

[Valentine, 1995] Valentine, S. (1995). *Information and Software Technology,* **37** (5-6), 293–302.

[vanHee *et al.*, 1991] vanHee, K., Somers, L., & Voorhoeve (1991). In: *Proc. VDM 91: Formal Software Development Methods,* (Prehn, S. & Toetenel, W., eds) volume 551 pp. 204–219, Springer-Verlag, Berlin.

[Wordsworth, 1992] Wordsworth, J. (1992). *Software Development with Z.* Addison-Wesley.