Engineering formal requirements: analysis and testing

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

Abstract

We introduce a method for formal analysis and symbolic testing of behavioral aspects of Z specifications. We start defining a (chemical) operational semantics, which supports an abstract execution model and some new constructs to allow the verification of dynamic properties. Moreover, using such a semantics, we have built a parallel animator of Z specifications which automatically constructs distributed prototypes directly from a (refined) specification: such a tool makes effectively observable concurrent behaviors of the Z requirements specification.

1 Introduction

Formal methods like Z [14] can play an important role in the engineering of reliable software systems. Requirements specification documents written with a formal notation can be mechanically checked; their careful analysis and early validation can be the starting point of a software development process completely based on formal methods, in which the quality of both software products and processes can be enhanced [5].

The Z notation is currently used as a specification language to formally describe and analyze the requirements and the architectures of a wide range of hardware and software systems. However, Z has been more successfully used for the specification of sequential systems. Since it is an extremely abstract specification language, the specifiers of concurrent or distributed systems have very poor support and need to pay special attention to behavioral aspects of their formal documents. A number of researchers have attempted to overcome these difficulties, typically combining Z with some other formalism (Petri Nets, Temporal Logic or TLA [10, 4, 1]) well suited to specify dynamic properties. Usually no tool is given to support the engineering of formal requirements written in these hybrid notations, except maybe a syntax checker based on a LaTeX-like concrete syntax (eg. fuzz).

The approach we present here offers the specifier two complementing approaches for validating and testing his specification. The former allows a formal analysis of a Z document: its dynamics is formally expressed and studied on an *execution model*, to be considered a sort of abstract symbolic execution in the sense of [11]. In addition to the abstract, declarative semantics given in [14], we introduce a new operational semantics based on the *chemical metaphor* embedded in the notation of the Chemical Abstract Machine [2]. We define on such a semantics a logic offering a number of constructs which can be used to define and analyze dynamic properties.

The latter approach is based on a tool for automatic generation of a distributed prototype of the system which makes practical animating the specification in order to debug it from errors, inconsistencies, and ambiguities in a truly concurrent framework. The tool is a parallel animator: it consists of a source-to-source compiler for the automatic translation of a Z specification into a program written in the concurrent language Shared Prolog, which has a chemical semantics as well. The use of a combination of logic programming with coordination mechanisms has the effect that the resulting animation can be executed in a truly distributed environment to make observable the concurrent behaviors of the system being specified.

2 A concurrent specification

In order to explain our approach we use an example widely discussed when dealing with concurrent dynamics of formal specifications: the Lift System [8, 12].

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

2.1 Formal requirements specification

In [8] the requirements of such a lift system are specified using Z; a number of liveness and safety proper-

ties are stated using some Unity like logic constructs. Although the use of Unity logic allows to capture dynamic properties not directly expressible in Z, the intended operational semantic model is strongly underspecified and the analysis method is unclear.

We elaborate a simpler version of the lift problem with respect to [8]. Although our specification looks very similar to the original, the interpretation we associate to it is completely different.

$$DIRECTION ::= up \mid down$$
$$STATE ::= stopped \mid moving$$
$$DOOR ::= open \mid closed$$
$$REQUEST_TYPE ::= up_request \mid down_request$$

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

Lift

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

The following schema describes the system:

Lift_System

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

requests is the set of requests indicating their type and floor. The initialization operation schema is:

Init_Lift_System

$$Lift_System'$$

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

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

Make_Requests

$$\Delta Lift_System$$
$$r? : REQUEST_TYPE$$
$$f? : \mathbb{N}$$

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

The schema describing the moving of the lift up (a single schema in [8]) is here split in two operation schemas, *Move_Up_Up* and *Move_Down_Up*: in this way we avoid $\vee$ operator in schemas and make simpler the rules for the semantics mapping (see Sect.3). We only specify *Move_Up_Up* operation: *Move_Down_Up* is very similar, it specifies the changing of direction. The schema *Move_Up_Up* defines the operation of moving the lift up in case up requests are presents above the lift.

Move_Up_Up

$$\Delta Lift_System$$

$$door = closed$$
$$(\exists f : \mathbb{N} \bullet (up_request, f) \in requests \wedge f > position)$$
$$direction = up$$
$$position' = position + 1$$
$$direction' = up$$
$$state' = moving$$
$$door' = closed$$

Move_Down_Down and *Move_Up_Down* are the schemas for the moving of the lift down: we do not report here for conciseness. The other schemas are *Open_Door* and *Close_Door* which 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 An Operational Semantics

The standard Z semantics [13, 3] is declarative and does not offer any formalization for concurrency. Thus, we have defined an operational semantics based on a chemical model. For simplicity and conciseness, here we consider a restricted version of Z.

Before giving semantics to Z constructs, we define the relevant syntactical elements of a Z specification. The elementary components of any Z document are State schemas and Operation schemas.

Moreover, because of the concurrent interpretation of Z we are going to give, we make the following assumption: all variables not explicitly mentioned in the postconditions of an operation schema need not to 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. [12]; however, this is not standard Z too.

A Chemical Abstract Machine [2] 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. In the Chemical Abstract Machine (CHAM) 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, 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 capture eventual executions.

We study a CHAM interpretation of Z specifications which allows us to deal with concurrent dynamics.

Intuitively, an instance *inst* of a state schema s is associated to a solution ($inst \longrightarrow \mathbb{P}(C)$) 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 conditions, pretuples, and posttuples are solutions composed of pre and post conditions of the operation.

A molecule is a tuple made of a name, a type, and a value, while a solution is a bag of molecules:

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

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

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

A rule is composed of a conditional part, that defines the applicability of the rule, and two solutions to indicate molecules to delete and add to the state solution. We call the first *SOLUTION* "pretuples" and the second "posttuples" 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 matching the pretuples of the rule.

The semantics associates a solution to a schema_instance:

$SCHEMA_INSTANCE \longrightarrow SOLUTION$

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

A rule is associated to an operation schema:

$SCHEMA_OP \longrightarrow RULE$.

Pre and postcondition of an operation schema are mapped on different parts 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 (molecules 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 ($\in$, $\sqsubseteq$) is mapped on a pretuple (a removal) and also on a posttuple (reinsertion) if Z postcondition does not contain an indication of removal of that element: in other words a membership test is seen as a removal and reinsertion.
- Postconditions containing mathematical operators ($+$, $-$,..) on naturals are encoded deleting one molecule and adding the updated molecule. 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 readded as already described [1]. 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.

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.

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

The initialization operation *Init_System* is mapped on a chemical rule with no pretuples:

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

The state schema instance obtained after 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 pretuples):

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

The rule corresponding to the operation *Move_Up_Up* (rule corresponding to *Move_Down_Up* is very 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:

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

4 An abstract execution model

We define now an *execution model*, namely a way of abstractly executing a Z specification document, and

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

a Unity-like logic to reason on properties exhibited by such a model.

The execution model is defined on the semantics just described and 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 there can be several different applicable operation sets (chosen in the set of all the enabled operations on that node) depending on the non determinism of the choice of the operations being in conflict. Each branch corresponds to the application of a group of enabled operations that could be applied without conflicts, as dictated by the CHAM model [2].

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

$$\begin{aligned} TREE &::= Void_tree \\ &\mid fork\langle\langle PAIR \times \mathrm{seq}\, TREE\rangle\rangle \\ PAIR &== SCHEMA_INSTANCE \times \\ &\quad \mathrm{seq}\,\mathbb{P}\, SCHEMA_OP \end{aligned}$$

Every state schema is mapped on an execution tree with specific properties; 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 operations in the sets belonging to the sequence seq must be enabled on s;
- all operations in the sets belonging to the sequence seq must act on the state schema of which s is instance;
- each set member of the sequence seq must contain operations that can concur (that is without conflicts);
- for every s' in a pair (s', seq'), label of one of the children of the node labeled (s, seq), there must hold the postconditions of all the operations applied to reach that node (sequence structure helps to link nodes and operations set).

4.1 A logic for expressing dynamic properties

In order to be able to reason on dynamic properties, like deadlocks and starvation, we borrow a few constructs from the Unity logic. Liveness (namely "a good thing will eventually occur") or safety (namely "a bad thing never happens") properties can be expressed through predicates (as the ones in the operation schemas) built using some logic operators ($\wedge$, $\vee$, $\neg$, $\Leftrightarrow$, $\Rightarrow$) and Unity constructs stated out of every operation schema. Properties have a chemical interpretation as well, so that we can analyze their truth on the execution model.

- p *unless* q means that whenever p is true during the execution, surely 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 either q will be true or still p holds.
- The meaning of p *ensures* q is that when p becomes true than eventually q will hold and before that moment p is still valid. That is, if p is true on a node, q will eventually hold on a node in all its subtree and before that moment p is valid on all the visited nodes.

The other constructs are **Stable**, **Invariant**, **Leads_to** which are defined in a similar way (further details in [6]).

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

4.2 Formal analysis of the lift system

It is now possible to introduce and verify dynamic requirements of the Lift system with the help of the constructs defined on the chemical execution model.

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

The first theorem states that in every moment the door is open if and only if the lift is stopped. The second one ensures that when the lift arrives to a floor where requests are present 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 arriving at the requested floors.

We now prove **Theorem 2** (the other two theorems can be proved in a similar way).

Proof of Theorem 2: If p **ensures** q (where p is $(t, f) \in requests \wedge f = position \wedge door = closed$ and q is $door = open$) must be valid, first p **unless** q has to hold (see the formalization). That means that for every enabled operations set on the solution containing molecules $(requests, \mathbb{P}(REQUEST_TYPE \times$

$\mathbb{N}), (t, f))$, $(position, \mathbb{N}, pos)$ and $(door, DOOR, closed)$, the application leads to a state where either molecule $(door, DOOR, open)$ 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 the only enabled operations on an instance state containing those molecules are *Make_requests* (that only adds new requests molecules without modifying the lift state) and *Open_Door* that exactly opens the lift door. Hence, p **unless** q holds.

To prove p **ensures** q is now necessary to ensure that, given a state where $p \wedge \neg q$ holds, there is an enabled operations set applicable, which leads to a state in which q holds. In our specification such a set is composed of the operation *Open_Door* and some (or none) *Make_Requests*. This completes the proof.

5 Animating a specification

A formal requirements document should provide a precise and rigorous description of a 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 [9]. Z is not executable because it allows to declare not computable entities, like infinite sets or non computable functions, and to express properties and operations on them.

However, it is often desirable to have a prototype of the system which can be tested and whose dynamic properties can be directly inspected by either the specifier or even the customer. The technique of *animation* has been introduced in order to overcome the difficulty of obtaining a prototype from a non executable specification language like Z. Several approaches have been used to animate a Z specification, using different methodologies and languages; a short review can be found in [15]. All the proposed solutions have 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 [3].

We propose here a method for the distributed animation of specification documents written in Z by translating them into programs written in the coordination language Shared Prolog [7], an extension of Prolog with support for parallelism.

Our approach consists of refining the original specification to obtain a more procedural version directly translatable or executable by generate-and-test clauses and queries. We define a subset of Z in which specifications must be refined before the animation process is carried on. The translation from such a refined specification is automatically performed by a source-to-source translator which produces a correct distributed program in Shared Prolog. The code obtained is finally compiled by the SP compiler and executed over a network of workstations.

5.1 A Coordination Language

Shared Prolog is a coordination language based on the combination of the shared dataspace coordination model (as in Linda) with logic programming computing (as in Prolog) [7].

A SP program consists of a set of modules called *theories*, an *initial goal* and a 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 the special goal *tuple_space*, while *initial_goal* defines some agents which share the tuple space (we also call them "active tuples", since they are part of the tuple space):

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

An agent represents a process executing a "theory" and accessing the tuple space for reading/consuming/producing tuples. A *theory* is a Prolog program extended with mechanisms for coordination through the tuple space.

Each theory is composed of a set of *patterns* and a *knowledge base*. Syntactically a theory looks as follows:

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

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

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

The *in_guard* and the *read_guard* are evaluated trying to unify a multiset of tuples in the blackboard with those contained in the guard. The body of a pattern contains predicates defined in the knowledge base of the theory. Semantically, each pattern consists of two components named *preactivation* and *postactivation*. The former defines an activation condition consisting of reading and/or deleting some tuples from the blackboard. The latter consists of a Prolog goal (*body*) and two multisets of tuples: one to be added to the blackboard in case of successful evaluation of the

goal (*out_set*), and the other to be added in case of body failure (*failure_set*).

Initially every pattern of a theory is examined to check the satisfiability of its own guard. If a guard is satisfied, the corresponding body is evaluated and tuples are added to the blackboard. If several patterns satisfy their preactivation, one is nondeterministically chosen and executed.

5.2 Animating Z specifications with SP

Operation schemas in Z 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.

Consistently with the Chemical sematics for Z, an operation is enabled if the current state of the system satisfies its preconditions and two (or more) operations can concur if they are enabled at the same time and not conflicting. The translation of a specification into a concurrent language with chemical semantics (like SP) has the immediate effect of capturing and making observable the concurrent characteristics of the system being specified.

In order to simplify its translation we have restricted Z to a number of admissible constructs (we call Zel - Z elementary - the resulting notation). The limitations we have introduced concern types and use of variables, form of predicates, and usability of some quantifiers. No restrictions are imposed on schemas construction and connections.

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

Variables in the declarative part of a schema must be defined as $\mathbb{P}$, bag or seq of another type X, because they are mapped into SP tuples, as we will see later. 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.

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 (more details in [6]).

The order of the predicates is important and influences the SP program obtained from the animation. One must pay attention to define predicates without unbound variables, so that the corresponding SP program does not evaluate non ground terms.

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

5.3 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 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.

Each variable defined in the predicative part of a Zel schema corresponds to a unique SP tuple in the blackboard, obtained concatenating the name of the variable and the schema in which it is defined. Input variables are removed from the blackboard, while output variables are added. For each schema, the system tries to determine a couple of animatable predicates in the list of precondition and postconditions, and the corresponding action is added to the pattern.

5.4 Querying the animation

The SP program obtained from the transformation process has to be initialized before it can be executed. In fact, the initial state of the blackboard and the initial goal 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.

6 Animating the Lift System

We describe now the animation of the specification of the lift system introduced in Section 2. The first step

consists of refining the Z document. This amounts to refining Z schemas to Zel and making some simplifications.

For instance, the following schema describes the operation of moving the lift up, provided that there is at least one pending request above its current position when the lift is going up.

Note how few changes have been made (they only concern variables defined as $\mathbb{P}\, sometype$ and i/o variables).

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

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* reported above.

```
{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 for the SP program 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. Also at least an agent has to be activated by the initial goal:

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

initial_goal (op_liftsystem).
```

The SP program is then compiled and executed. In the figure (b) is reported the result of the animation of the system for the above sequence of service requests, with the final state of the lift.

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

The part (a) of the figure shows, for example, the animation report if we "forget" to include in the *Move_Up_Up* and *Move_Down_Up* schemas the test concerning the shutting of lift door:

```
req upRequest,3
up from 1 to 2
up from 2 to 3
open 3
req upRequest,4
close[]
up from 3 to 4
open 4
req upRequest,5
up from 4 to 5
open 5
close[]
down from 5 to 4
open 4
final set:[(upRequest,4)]
final pos.:fl 4 door:open
   (a)
```

```
req upRequest,3
req upRequest,1
close[]
up from 1 to 2
up from 2 to 3
open 3
req upRequest,4
close[]
up from 3 to 4
open 4
final set:[(upRequest,4)]
final pos.:fl 4 door:open
   (b)
```

The above 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 giving in input different test cases that are instances of a particular functional requirement and observing the resulting animation. An unsuccessful case shows that the functional requirements tested is not valid, but only if all instances of a functional requirement are tested satisfiability is demonstrated [11]. The main advantages of the animation are in terms of accelerating the validation of the specification with respect to the user requirements and increasing the degree of trust of the specifier in the system being built. As result an early detection of errors can be accomplished, saving development time and reducing overall costs.

7 Conclusions and future works

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 abstract and is able to deal with global behavioral components, whereas the second is concrete offering a rapid prototype for the study of system behavior using some specific test cases. The main novelties of our approach are the introduction of a Z chemical semantics to analyze operational properties of reactive systems and the integration of parallelism into the prototyping language to highlight concurrent behaviors.

Our work can be closely compared with the one exposed in the classic paper [11]. 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 [5], and to introduce and study methods and tools for the analysis, testing, and verification of the requirements specification document.

The notation and tool described here have been used in our undergraduate course on software engineering in several student projects. 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. We are studying correctness and completeness of our semantics with respect to the original one, and the possibility to extend either Z coverage (i.e. Zel notation) or SP language.

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] P. Baumann and K. Lermer. A Framework for the Specification of Reactive and Concurrent Systems in Z. In P. Thiagarajan, editor, *Proc. 15th Conference on Foundation of Software Technology and Theoretical Computer Science*, volume 1026 of *Lecture Notes in Computer Science*, pages 62–79, Bangalore, India, 1995. Springer-Verlag, Berlin.

[2] G. Berry and G. Boudol. The Chemical Abstract Machine. *Theoretical Computer Science*, 96:217–248, 1992.

[3] 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.

[4] 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.

[5] P. Ciaccia, P. Ciancarini, and W. Penzo. From Formal Requirements to Formal Design. In *Proc. 7th Int. Conf. on Software Engineering and Knowledge Engineering*, pages 23–30, Rockville, Maryland, 1995. Knowledge Systems Institute.

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

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

[8] 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.

[9] I. Hayes and C. Jones. Specifications are not (necessarily) executable. *IEE Software Engineering Journal*, 4(6):330–338, November 1989.

[10] 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.

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

[12] 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.

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

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

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