Comput Syst Sci & Eng (2005) 1: 61–72
© 2005 CRL Publishing Ltd

International Journal of
Computer Systems
Science & Engineering

Early requirements validation with 3D worlds

Alfredo Raúl Teyseyre and Marcelo Campo*

ISISTAN Research Institute, UNICEN University, Campus Universitario, B7001BBO Tandil, Argentina. Email:{teyseyre, mcampo}@ exa.unicen.edu.ar

It is a well-known fact the real significance of correctly determining requirements of a system at the very beginning of the development process. Indeed, experience demonstrates that the incorrect definition of requirements leads to development of deficient systems, increases the cost of its development or even causes projects to fail. Thus, it is crucial for clients to verify that the planned system satisfies their needs. In order to help users in the process of requirements understanding and validation this work proposes using 3D visualization techniques. The use of these techniques can reduce the communication gap between clients and developers resulting in a much more effective process of requirements validation. The approach tries to take advantage of the benefits of the 3D visualization, complementing this with the advantages of formal specifications. The approach proposes the use of formal specifications in a lighter way. This means that no formal reasoning (theorem proving) is carried out to check the properties of the specified system and the emphasis is focused on the execution and animation of the specification for early validation. A prototype tool that materializes the proposal was developed. The tool allows specifying the requirements in the formal language Z, defining a graphical representation of them and creating a 3D animated visualization of their execution through which the users can validate them.

Keywords: Requirements, Visualization, Requirements Visualization, 3D Graphics, Formal Specifications, Lightweight Formal Methods

1. INTRODUCTION

Meeting user requirements of a software system is a major challenge to software developers. The experience gained in several large projects reveals that a very large percentage of errors were consequence of the imprecision in the earlier stages of the development process [27]. Therefore, it is a well-accepted fact that it is crucial to express user requirements as completely, correctly and unambiguously as possible. Indeed, forgoing requirements completely would be like trying to navigate a complex route without a road map [3]. In consequence, it is vital for the customers to be able to confirm that the planned system meets their needs, and this means that the system must be described in a way that they can understand it [28]. In fact, users when asked to specify requirements generally claim, "I don't know how to tell you, but I'll know it when I see it" [3].

Many conventional approaches have been applied to validate requirements, but most of them, fail in detecting errors [18]. In contrast, formal approaches give precision at specification time. In that sense, formal specifications enable us to denote unambiguously the meaning of a requirements specification document due to their formal syntax and semantics. However, except in safety-critical work, the cost of full verification is prohibitive [15]. Besides, formal specifications often fail in the user validation process since they are based on formal notations not always understandable by users. Therefore, in order to overcome these difficulties visualization techniques appear as an interesting alternative to explore.

Visualization is a method to comprehend information by the use of diagrams to represent it. Data are transformed into geometric representations that help users in the

*Also at CONICET

understanding process. In general, graphical representations provide a closer match to the mental model of users than textual representations and take advantage of their perception capabilities.

In spite of their success in numerous computing areas, little research has been reported in the area of requirements visualization. The previous approaches enable developers to validate visually the specification of a system with the user, but their poor expressive graphics make difficult the validation. Moreover, neither of the works makes use of current 3D graphics capabilities in order to present animations that are more realistic. However, 3D visualization techniques can be a powerful tool to help in the analysis and understanding of requirements. The use of visualization techniques could reduce the communication gap between the customer and developer resulting in a more effective requirements validation process [26]. In this context, the main objective of this work is using 3D visualization and animation techniques to validate requirements with the user.

A tool, called REQVIZ3D that materializes the proposal was developed. This tool allows specifying the requirements in the formal language Z [34], defining a graphical representation of them and creating a 3D animated visualization of their execution through which the users can validate them, as Figure 1 shows. For instance, this figure shows a visual prototype of a lift system, developed with REQVIZ3D, which enables the client to experience the system as formally specified.

This paper is organized as follows. Section 2 surveys current efforts towards requirements validation. Section 3 describes the approach and section 4 presents a case study. Section 5 presents a brief description of the prototype tool ReqViZ3D. Finally, section 6 outlines some preliminary conclusions and future work.

2. RELATED WORK

Clearly, the straightforward choice to capture requirements of a software system is natural language. However, natural languages specifications have been one of the main sources of ambiguity due to its rich vocabulary and its expressiveness [24]. As an alternative formal specification languages have been proposed. Formal specification languages have a formal syntax and semantics, which make it possible to denote unambiguously the meaning of the requirements. The best-known formal specifications languages are Z [34], B [20] and VDM [16] among others.

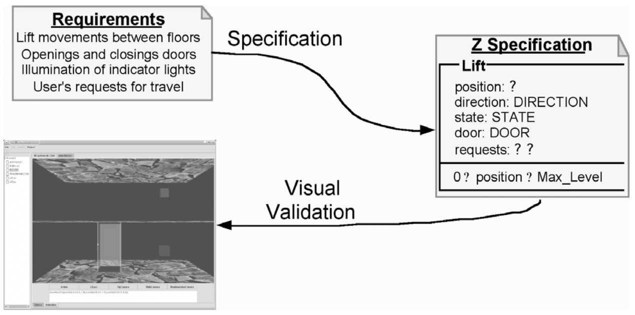


Figure 1 Requirements validation

Even though formal specification languages are precise, concise and unambiguous, they fail in the validation process with a customer: it is difficult for a customer to understand formal specifications because they are based on mathematical foundations and notations. However, having formalized a system, automated support is available for validating the model by execution.

Many have proposed the use of executable formal specifications for the construction of prototypes to validate software requirements with users at an early stage through feedback [9]. Techniques like execution have been introduced to overcome the difficulty of using a non-executable specification language, allowing the specifier to either test or rapidly implement his/her specification document. Several researchers have reported success in executing subsets of Z translating them to languages such as PROLOG or LISP [26, 12].

Although specification execution can provide direct feedback during the process of making a specification and decrease errors made at earlier stages of the development process, it seems to be more helpful to developers rather than to users. This is mainly because the execution is still based on the underlying specification notations and the system is still described in a way that users cannot understand.

Visualization techniques have been used in many computing areas. However, in spite of their success, little research has been reported in the area of requirements visualization. Most of the reported works are oriented towards the validation of requirements on specific domains, as for example real-time systems (IPTES [29] and ENVISAGER [10]), and do not address a wide range of problems as formal specification methods do. Moreover, there is only one fixed graphic representation of requirements, for example nets, limiting in consequence the expressive power of the visual presentations. This could lead to poor expressive presentations that make difficult understanding.

Among the few works reported, two of them can be remarked: VIZ [26] and POSSUM [12]. Both systems enable the developer to visually validate specifications in Z. Technology provided by VIZ allows software developers to choose an appropriate representation of objects used in an executable formal specification and create animations of these objects in an interactive fashion. However, the system only supports the construction of simple presentations. On the other side, POSSUM facilitates the construction of complex presentations using TCL/TK, but it does not provide assistance in the construction of the presentations. Moreover, both systems only support 2D presentations and do not take advantage of current 3D graphics technologies. Table 1 presents a brief comparison of the two systems.

Table 1 VIZ and POSSUM compared

Tools		ViZ	Possum
Formal Specification Language		Z	SUM (Z extension)
Execution Language		Lisp	Mercury
Visualization	**Based on States**	Z states	Z states
	Construction	Interactive	Scripts (Tcl/Tk)
	Construction degree of complexity	Simple	Tcl/Tk Knowledge
	Kind of Visualizations	Simple	Complex
	3D	No	No

3. THE APPROACH

Our main objective is the visualization and animation of requirements to achieve a more effective requirements validation process. The approach proposes the use of visualization, as well as, of formal specifications. Before describing the approach lets state, what a validation means [20]:

"*Validation of a description D against a description C means checking that D satisfies the properties specified in C, where C is the informal or semi-formal description.*"

In the context of requirements validation, the check consists of ensuring that the specified system (D) is the system that the client wants, where C is an informal set of the client expectations. Figure 2 resumes the key ideas behind this project. After writing an informal specification, we express requirements formally in Z. A formal specification makes it possible to denote unambiguously the meaning of the system requirements. In parallel, we build a graphic presentation suitable for validating specification concepts with the user. It may be possible to present abstract visual forms of the system to be designed or even to build a visual prototype of the actual system, displaying the physical properties and behavior of the main system aspects. After linking the graphic objects with the specification objects, requirements are validated by means of visualizations. Therefore, knowing that the requirements specification conforms to the user needs, it is a much more reliable base for developing the system. However, if problems are detected, specifications must be corrected.

The formal approach adopted can be classified as a light one, in the sense, that no formal reasoning (correctness proofs[1]) is carried out to check if the properties of the specified system respond to the informal requirements and the emphasis is focused on the execution of the specification [17, 13]. Using formal methods in a lighter way is both a key to using them on large-scale applications and a way of penetrating fields outside the safety-critical area, where formal methods are mainly used and a detailed application can be justified because of the danger of loss of life [17].

We have decided to formalize requirements in Z. This is mainly because the experience gained in the past years from case studies has proven that a large variety of specification problems may be successfully addressed in Z and set theory forms an adequate basis for building the more complex data structures which are needed in specifications [27]. However, it should be noted that also other formal specification languages could have been selected.

In order to present an animated presentation, to validate requirements, formal specifications are executed. The execution of the specification allows the user to walk through a specification using different scenarios that are shown by visual presentations. The animation displays the behavior of the specified system and provides a means of dynamic testing. As a result of this approach, it is intended that:

- Misunderstandings between clients and developers are detected.

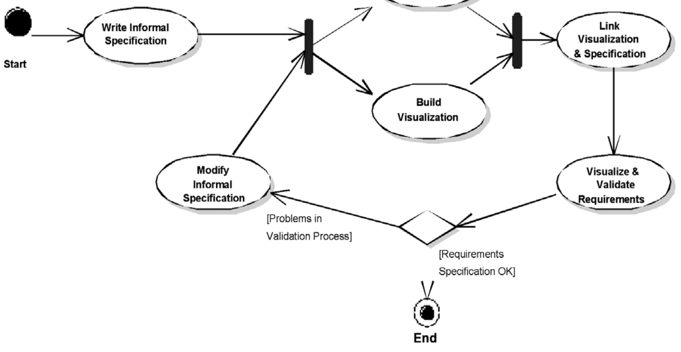

Start
Write Informal Specification
Write Formal Specification
Build Visualization
Link Visualization & Specification
Visualize & Validate Requirements
Modify Informal Specification
[Problems in Validation Process]
[Requirements Specification OK]
End

Figure 2 Requirements validation process

- New services arise and obscure ones are clarified.
- Inconsistencies in specifications are detected.
- The developed system is much closer to the needed system.
- The development effort is reduced.

The approach is in part similar to prototyping [32], where an experimental system is developed as a basis for formulating requirements. A prototype is an initial version of a system, used to demonstrate concepts and try out design options. A prototype can be used in the requirements engineering process to help with requirements elicitation and validation. The executable model of the system is used to check requirements and then is usually thrown away when the system specification has been agreed.

In the next subsections, each aspect of the approach is discussed in more detail: from execution of specifications to visualization and validation of requirements using a simple example.

3.1 Execution of formal specifications

Formal specification languages, such as Z, have been developed to define precisely and concisely the characteristics and specifications of a software system. However, formal specification languages fail in establishing a very important property for an immediate reflection of the consequences of the specifications and for an early validation: the executability of a specification [9].

Z was not conceived for execution, since its aim is to define the abstract properties of the system being built and not the design decisions or the implementation details of the system. Z specifications are declarative and the developer can declare non-computational entities, such as infinite sets or non-computable functions and specify properties and operations on them.

Therefore, in order to execute a formal specification in Z, the notation of Z must be restricted to a subset almost directly executable. This means that Z is restricted forbidding the declaration of non-computational entities and adapting it to the capacities of executable languages that, on the other side, are less expressive than non-executable ones, because their functions must be computable and their domains must be finite.

1 A proof consists of the sequence of well-defined formulae $F_1, F_2, ..., F_n$ in the language in which each formula F_i is either an axiom or derivable by an inference rule from previous formulae in the sequence. The last formula F_n in the sequence is said to be proven

Several problems arise according to the chosen method of translation and the target language. Usually these problems derive from trying to match different levels of abstraction. 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 reasonable efficient execution to test the specification [4].

Due to mathematical and logical foundations of formal languages, declarative or functional languages seem to be the most suitable ones. A logic programming language is a very interesting choice for translating a specification language as Z, which is based on first order logic. The conceptual gap between a logic programming language (which is a subset of a first order logic) and a specification based on logic is significatively less than the gap between a specification based on logic and an imperative language. For instance, a straightforward way to animate Z documents seems to be mapping Z specifications into PROLOG because practice shows that most Z predicates are easily implemented in terms of PROLOG clauses. Figure 3 illustrates the translation and execution of Z specifications: the Z specification document is translated to PROLOG and then executed by a PROLOG interpreter using a library of basic Z predicates in PROLOG. In addition, this figure shows for instance the implementation of a PROLOG predicate *mem*, which defines de membership of an element to a given set.

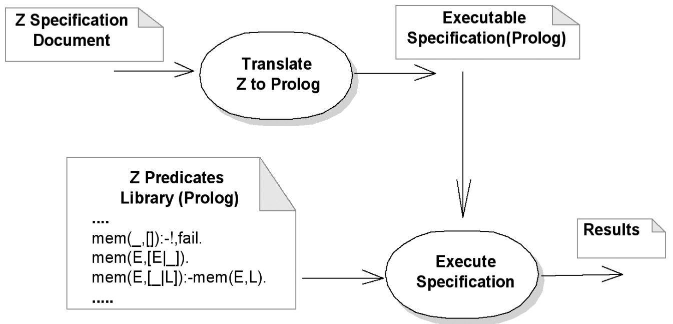
In fact, it is possible to take a subset of Z for generating PROLOG code. This point of view is compatible with the assertion that a considerable part of Z has executable semantics [4]. In particular, the approach we adopted is similar to the approach proposed by Sterling [36]:

- The semantics of the subset of Z is clear.
- The translation of the subset to PROLOG is practically direct.
- The power of the subset is sufficient for many applications.

Although these claims need a deeper argument they will not be discussed, as the focus, is the application of logic programming in requirements engineering. Instead, we show how the transformation can be done by using an example.

3.1.1. Translating Z to Prolog

The main construction in Z is the schema. A schema enables us to decompose a specification into small pieces. In Z, schemas are used to describe both static and dynamic aspects



```
....
mem(_,[]):-!,fail.
mem(E,[E|_]).
mem(E,[_|L]):-mem(E,L).
.....
```

Figure 3 Translation and execution of Z specifications

of systems. In order to show the transformation a simple and widely discussed LIFT SYSTEM [8] example is presented: "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."

First, we introduce a schema to describe the system state, which corresponds, to the static part of the system. The static aspects include the states it can occupy and the invariant relationships that are maintained as the system moves from state to state.

$$\begin{array}{|l}
\textit{Lift} \\
\hline
\textit{position} : \mathbb{N} \\
\textit{direction} : \textit{DIRECTION} \\
\textit{state} : \textit{STATE} \\
\textit{door} : \textit{DOOR} \\
\textit{requests} : \mathbb{P}\,\mathbb{N} \\
\hline
0 \leq \textit{position} \leq \textit{MaxLevel} \\
\hline
\end{array}$$

MaxLevel == 5

DIRECTION ::= up | down

STATE ::= moving | stopped

DOOR ::= open | closed

The lift can be defined by its position, direction, state, door state and pending requests. The direction of the lift can be up or down, while the state indicates if the lift is moving or stopped. The lift door opens when the lift arrives at a floor and it is closed while the lift is moving. Possible requests are up or down requests. The invariant states that the movement of the lift is restricted to an interval of valid floors.

In order to translate a state schema, a PROLOG clause is created, whose name is the same of the schema, the arguments of the clause are the state variables and the invariant of the schema is the body of the clause. An additional argument is also added for storing global declarations of the specification. For example, Figure 4 shows the translation of the Lift schema. The clause getContain, which is used to access values of global declarations, enables us to obtain the value of the constant Max_Level.

We can now start defining the system operations, that is, the dynamic aspects of the system. The dynamic aspects include operations that are possible, relationship between their inputs and outputs and changes of state that happen. In an operation schema, we can identify a declaration part and a predicate part. The declaration part defines the inputs and the outputs of the operation as well the system state schemas over which it operates:

$$\begin{array}{|l}
\textit{MakeRequests} \\
\hline
\Delta \textit{Lift} \\
\textit{floor?} : \mathbb{N} \\
\hline
\textit{requests}' = \textit{requests} \cup \{\textit{floor?}\} \\
\hline
\end{array}$$

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

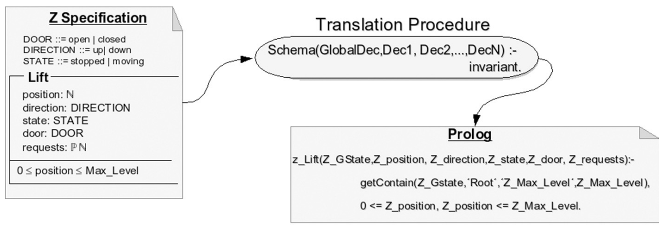
$$\begin{array}{|l}
\textit{OpenDoor} \\
\hline
\Delta \textit{Lift} \\
\hline
\textit{door} = \textit{closed} \\
(\exists f : \mathbb{N} \bullet f \in \textit{requests} \land f = \textit{position}) \\
\textit{state}' = \textit{stopped} \\
\textit{door}' = \textit{open} \\
\textit{requests}' = \textit{requests} \setminus \{F\} \\
\hline
\end{array}$$

MakeRequests schema adds a new request to the requests set. The declaration (Lift alerts us to the fact that the schema is describing a state change in Lift schema. The declaration floor? defines the input of the schema, by convention names of inputs end in a question mark. The part of the schema below the line is the predicate part. The other operation schema MoveUpUp defines the operation of moving the lift up if up requests are present above the lift (in addition, similar operations are defined for the other directions and for closing the door, not reported for conciseness).

For translating an operation schema a PROLOG clause is created, whose name, is the same name of the schema, and the inputs and outputs of the operations are the arguments of the clause (Figure 5 shows the translation procedure). Also, two additional parameters are needed (PROLOG structures) for holding the state of the system before and after the execution of the operation. Other two parameters are included for maintaining global declarations, logging and tracing the execution of the operations (information used to animate visualizations). Finally, the body of the clause is composed by the assertions of the Z schema, that is, pre and post conditions of the schema operations. Also, after the execution of an operation schema the invariant must still remain true, so in order to verify that fact a call to the PROLOG clause of the state schema is done. Figure 5 illustrates the translation procedure of the schema MakeRequests. The operation addChangeOp registers in the global state that the operation was executed.

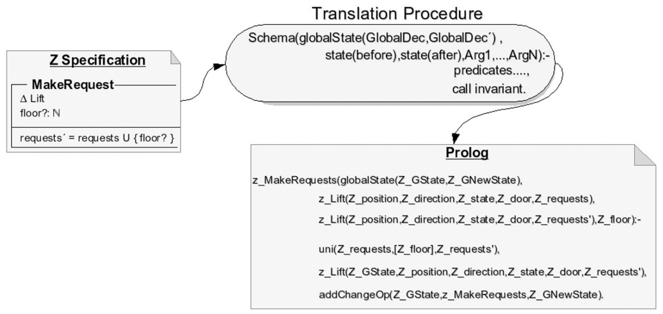
3.1.2. Object based extension

Z makes use of schemas to structure specifications. However, this is not sufficient for structuring large specifications. Many researchers remark the need for better structuring in specifications to improve readability and allow modular verification and refinement. In that sense, many different approaches proposed providing Z with an object-oriented structuring mechanism [35]. These approaches have involved the extension of Z with object-oriented concepts [7,1,21] or attempts to use Z in a more object-oriented style [11,38].



```
z_Lift(Z_GState,Z_position, Z_direction,Z_state,Z_door, Z_requests):-
          getContain(Z_Gstate,´Root´,´Z_Max_Level´,Z_Max_Level),
          0 <= Z_position, Z_position <= Z_Max_Level.
```

Figure 4 Z state schema and its translation procedure


Z Specification
MakeRequest
Δ Lift
floor?: ℕ
requests´ = requests U { floor? }
Translation Procedure
Schema(globalState(GlobalDec,GlobalDec´) ,
state(before),state(after),Arg1,...,ArgN):-
predicates....,
call invariant.
Prolog

```
z_MakeRequests(globalState(Z_GState,Z_GNewState),
          z_Lift(Z_position,Z_direction,Z_state,Z_door,Z_requests),
          z_Lift(Z_position,Z_direction,Z_state,Z_door,Z_requests'),Z_floor):-

          uni(Z_requests,[Z_floor],Z_requests'),
          z_Lift(Z_GState,Z_position,Z_direction,Z_state,Z_door,Z_requests'),
          addChangeOp(Z_GState,z_MakeRequests,Z_GNewState).
```

Figure 5 Translation of an operation schema

In consequence, the approach adopted, makes use of Z in an object-oriented style. The operations on a particular state are grouped together. Consequently, the schema can be viewed as a class, that is a template, from which objects may be created by 'new' operations:

```
lift = new Lift
```

Objects of the same class have common operations and therefore uniform behavior. A message is sent to an object in order to execute a schema operation:

```
lift.MakeRequests(4)
```

However, the extension proposed just support objects, that is, the approach is not object-oriented, it is just object-based because it does not support class hierarchies defined by an inheritance mechanism [37].

In order to support the object-based extension the execution core for animating specifications chosen was JAVALOG [2]. JAVALOG is a PROLOG interpreter written in JAVA designed to allow easy integration between JAVA and PROLOG mixing LOGIC/OO paradigms. Each object of the specification is instantiated as Z-Executor JAVA object, that encapsulates PROLOG clauses.

JAVALOG enables the creation and usage of JAVA objects in PROLOG programs, mixing LOGIC/OO paradigms and preprocesses JAVA methods with embedded PROLOG enabling the common use of local variables in both paradigms. In JAVALOG, the elements manipulated by the logic paradigm are mapped to modules. Logic modules are defined as a set of HORN clauses.

The class diagram presented in figure 6 shows the core classes that support the execution of Z specifications. JAVALOG defines an object KnowledgeLogic for representing logic modules and a simple object called PlEngine that acts as a logic language interpreter:

```
PlEngine engine=new PlEngine();
engine.answerQuery(''append([1,2,3],[4,5],Res)'');
```

In addition, we have defined a class SharedPlEngine that encapsulates an instance of PlEngine and enables us to share the interpreter between several clients. The class ZExecutor adds all the behavior needed for executing Z specifications and uses a SharedPlEngine object. As well, a pool class PlEnginePool was defined, which is useful for minimizing

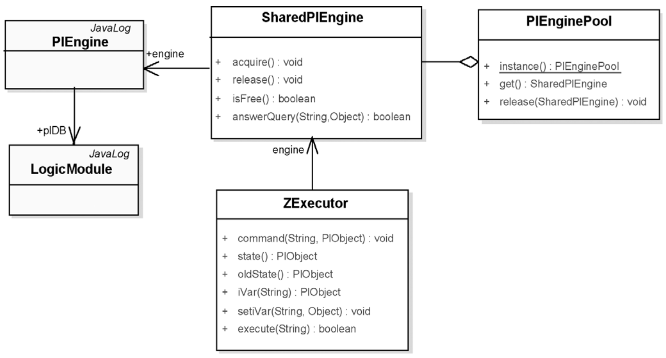

JavaLog
PlEngine
+engine
+plDB
JavaLog
LogicModule
SharedPlEngine
+ acquire() : void
+ release() : void
+ isFree() : boolean
+ answerQuery(String,Object) : boolean
engine
PlEnginePool
+ instance() : PlEnginePool
+ get() : SharedPlEngine
+ release(SharedPlEngine) : void
ZExecutor
+ command(String, PlObject) : void
+ state() : PlObject
+ oldState() : PlObject
+ iVar(String) : PlObject
+ setiVar(String, Object) : void
+ execute(String) : boolean

Figure 6 Class diagram

PROLOG engines creation overhead.

The translation of the Z specification uses the JAVALOG *new* clause for creating objects and the *send* clause for sending messages to an object. The following example illustrates the translation:

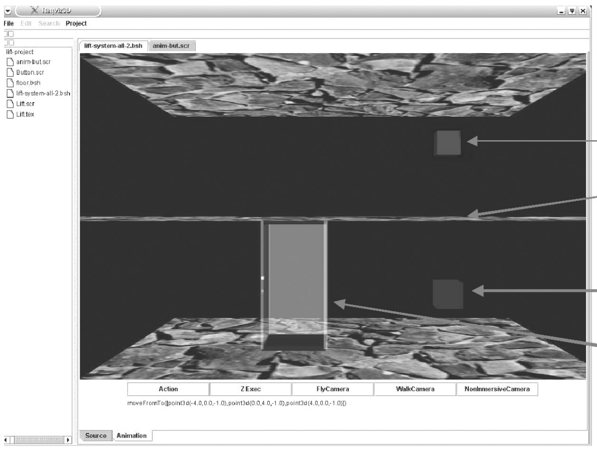
```
ALift = new Lift
ALift.MakeRequests(4)
⇓
newInstance('ZExecutor',['Lift'],ALift),
send(ALift,execute,[makeRequests,[4]],_)
```

An instance of ZExecutor is created. The argument Lift indicates the specification to be instantiated. After the creation of this instance, a message is sent to it in order to execute the schema operation makeRequests with four as argument.

3.2 3D visualization

Let us first state the notion of visualization, which is defined by Card [6] as follows: "the use of computer-supported, interactive, visual representations of data to amplify cognition", where cognition is the acquisition or use of knowledge. Visualization is a powerful tool to facilitate the analysis and understanding of complex information such as software requirements. This is mainly because it provides a closer match to the mental model of the users than textual representations and also reduces the communication gap between customers and developers.

At the beginning most of visualization systems display 2D graphics, but nowadays, more and more applications use 3D graphics in their visual presentations. Using this kind of presentations provides a greater information density than two-dimensional presentations, as a consequence, of a bigger physical space [30]. Also, an additional dimension helps to have a clear perception of relations between objects by integration of local views with global views [22] and composition of multiples 2D views in a single 3D view [19]. Besides, their similitude with the real world enables us to represent it in a more natural way. This means that the representation of the objects can be done according to its associated real concept, the interactions can be more powerful (ranging from immersive navigation to different manipulation techniques) and the animations can be even more realistic.



```
moveFromTo([point3d(-4.0,0.0,-1.0),point3d(0.0,4.0,-1.0),point3d(4.0,0.0,-1.0)])
```

Figure 7 Lift system visualization

On the other hand, several problems arise, such as intensive computation and more complex implementation than two-dimensional interfaces. These problems can be lighted using powerful and specialized hardware and several tools like 3D toolkits as JAVA3D [33] or 3D modeling languages such as VRML [14].

Therefore, in order to make the construction of the 3D graphics presentations easier, REQVIZ3D provides a graphics specification language for defining the geometry and behavior of the 3D graphics objects, which is described in the following section.

3.2.1. Building visualizations

Once the system is specified in Z, the developer defines the graphical representation of the requirements for visualizing and animating the specification concepts in a 3D world (as Figure 7 shows), and so validated by the users. In this example, the user can press the buttons of each floor and see how the lift services user requests going up or down. When the user presses a button, it is lighted on and when the lift services the request, it is lighted off. For building the visualization, REQVIZ3D provides a graphical specification language. A graphical object specification is composed by three main parts: geometry definition, actions and recognized events.

The geometry section defines the different shapes that can be used to present a graphic object. For example, the next script defines the lift graphic object:

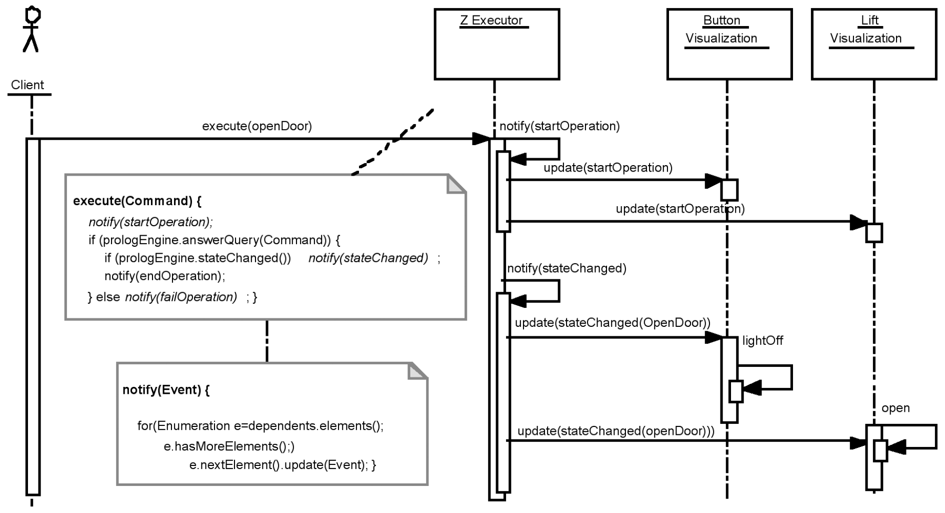
```
geometry([def(open,file('models/DOOROPEN.3DS')),
          def(closed,file('models/DOORCLOSED.3DS'))]).
```

This example defines the geometries of the lift, that is one when the lift is closed and another when it is opened. In addition, each geometry can be named in order to be identified and accessed.

The next section defines the behavior of the lift (open, close and goto). These actions are defined in terms of a set of predefined actions (translate, move...). For instance, the action open calls the switch action in order to show a graphic of the lift opened:

```
action(open, [ switch(open) ]).
action(close,[ switch(closed) ]).
action(goto(Floor,From),  [ call(Time is abs((From-Floor))),
                  moveTo(time(0,Time),[point3d(0.0,Floor,0.0)])]).
```

Finally, the events section defines the reactions of the lift in response to changes in the execution of a Z specification, using a change-propagation mechanism that ensures consistency between the specifications and visualizations based on implicit invocation [5]. The mechanism maintains a registry of the dependent components. The Z model takes the role of a publisher. The visualizations, that are the components that depend on changes of the publisher, are its subscribers. Changes in the state of the model trigger events that are propagated to the visualizations. Using this mechanism, the Z executor object announces different events about the



```
execute(Command) {
   notify(startOperation);
   if (prologEngine.answerQuery(Command)) {
      if (prologEngine.stateChanged())   notify(stateChanged) ;
      notify(endOperation);
   } else notify(failOperation) ; }
```

```
notify(Event) {

   for(Enumeration e=dependents.elements();
         e.hasMoreElements();)
               e.nextElement().update(Event); }
```

Figure 8 Events

execution state of the specification, as the method execute of figure 8 shows. The first event that is propagated when an operation is executed is the start operation event. It may be possible that the execution of the operation fails, so the fail operation event is announced. In contrast, if the operation is successfully executed an end operation event is propagated. At last, if the operation changes the state of the system a state changed event is triggered. In order to announce any change the ZExecutor invokes the notify operation which sends the update message to all dependent visualizations.

For example, Figure 8 shows the messages propagated when the Z operation openDoor is executed. When the door is opened, the presentation of the lift must be updated and the button switched off:

```
event(stateChanged,va(position), [  value(position,Pos),
        oldValue(position,OPos),goto(Pos,OPos)]).
event(stateChanged,op(openDoor),[open]).
event(stateChanged,op(closeDoor),[close]).
```

The following code defines the look and feel of the buttons used to call the lift from each floor. A button is represented by a box:

```
geometry([def(file('examples/lift/box.bsh'))]).
```

Two actions define the behavior of the buttons, lightOn and lightOff, that switch on or off the light of the button indicating that exists or not a request from that floor:

```
action(lighton,[colorFromTo(time(0,1000.0),
        color3f(0,0,0),color3f(1,0,0))]).
action(lightoff,[colorFromTo(time(0.0,1000.0),
        color3f(1,0,0),color3f(0,0,0))]).
```

Finally, the last section defines the events processed by a button. When the lift process a request the light of the button is turned off, and when the user clicks a button the light is turned on and a request is made by calling the schema operation MakeRequests:

```
event(stateChanged,op(openDoor),[getval(floor,Pos),
        value(lift,position,Pos),lightoff]).
event(oneShot,none,[getval(floor,Pos),
        callZ(z_MakeRequests(Pos)),lighton]).
```

We have described some of the graphics elements of the visualization; however, in order to make a complete scene, they must be composed and linked with a Z Executor object. For doing so BEANSHELL [25] was used. BEANSHELL is a small, free, embeddable, JAVA source interpreter with object scripting language features, written in JAVA. BEANSHELL executes standard JAVA statements and expressions, in addition to obvious scripting commands. BEANSHELL supports scripted objects as simple method closures like those in PERL and JAVASCRIPT. You can call BEANSHELL from your JAVA applications to execute JAVA code dynamically at run-time or to provide scripting extensibility for your applications:

- Creates a Z executor object for the lift specification:

```
ZAnimat z= new ZExecutor("lift.tex.pl");
```

- Composes the graphic objects making a 3D scene:

```
J3DCompositeComponent world=
    new J3DCompositeComponent();
J3DComponent lift=Factory3D.instance().
    createViewForZ("lift.scr",z);
world.addComponent(lift);
for(int i=0; i < 5; i++) {                //Creates each floor
     J3DComponent button=Factory3D.instance().
         createViewForZ("button.scr",z);
     J3DComponent floor=new J3DSimpleComponent();
     floor.builder(newBeanBuild3D("floor.bsh"));
      ......
     world.addComponent(button);
    world.addComponent(floor);
}
return world;
```

In consequence, each scripting language is useful for different tasks. The graphics specification language is helpful

Table 2 Scripting commands

Operation Type	Command	Description
Geometry Operations	geometry(geometries)	Defines different graphics representations for an object
	switch(geometry)	Changes actual representation
	morphFromTo(time(Begin,Duration), geometries)	Morphs from a shape to another one
Basic Operations	translate(X,Y,Z)	Translates (X,Y,Z) points from actual position
	rotate(X,Y,Z,A)	Rotates an angle A based on vector (X,Y,Z)
	scale(X,Y,Z)	Scales an object by a factor (X,Y,Z) in each dimension
	scale(S)	Scales an object by a factor S
	moveFromTo(time(Begin,Duration), Points)	Moves an object following a path
	moveTo(time(Begin,Duration), Points)	Moves an object following a path from the actual position
	colorFromTo(time(Begin,Duration),Colors)	Changes the color to another one
Z Related Operations	value(Variable,Value)	Returns the value of a variable of a Z State Schema
	oldvalue(Variable,Value)	Returns the previous value of a variable of a Z State
	callZ(Operation)	Executes a schema operation

for defining, at a high level of abstraction, the look and feel of the graphics objects and it provides a simple way to integrate visualizations with Z execution. Table 2 resumes the main commands of this language. On the other side, a language as BEANSHELL, is useful for interactively experimenting building general-purpose JAVA scripts.

3.3 Requirements validation

For example, as a consequence of visualizing the lift system, the following problem was detected: the lift door was still opened while the lift was moving. Although the modeled system was very simple, it seems to be easier to detect this problem in the visualization than on the textual specification itself.

In order to correct this problem, the invariant now additionally asserts that while the lift is moving the door must be closed. Moreover, the operation schemas that move the lift now verify before moving the lift if the door is closed:

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{Lift}} \\ \hline
\mathit{position} : \mathbb{N} \\
\mathit{direction} : \mathit{DIRECTION} \\
\mathit{state} : \mathit{STATE} \\
\mathit{door} : \mathit{DOOR} \\
\mathit{requests} : \mathbb{P}\mathbb{N} \\ \hline
0 \leq \mathit{position} \leq \mathit{MaxLevel} \\
\mathbf{state = moving \Rightarrow door = closed} \\ \hline
\end{array}$$

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

4. A CASE STUDY: AN AUTOMATIC TELLER MACHINE

This section presents a brief description of a case study developed with REQVIZ3D. An automatic teller machine which provides these basic services: deposit, withdraw, transfer, balance and user authentication.

In order to use the ATM machine, the user inserts its card and is prompted for a password. The password is validated and if it is correct, the client can make transactions by touching the screen buttons and entering values using a keyboard. The user receives a ticket for each operation and, in the case of extracting money, takes it.

The state of the system is defined by two functions (*bal* and *pass*) that hold the balance and the password of each account respectively, the state variable *log* that registers all the account transactions, and the variable *user* that represents the current user of the machine:

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{ATM}} \\ \hline
\mathit{bal} : \mathbb{N} \twoheadrightarrow\!\!\!\!\!\!\!\;+\; \mathbb{R} \\
\mathit{pass} : \mathbb{N} \twoheadrightarrow\!\!\!\!\!\!\!\;+\; \mathbb{N} \\
\mathit{log} : \mathbb{P}(\mathbb{N} \times \mathit{TRANSACTION} \times \mathbb{R}) \\
\mathit{user} : \mathbb{N} \\ \hline
\end{array}$$

$$\mathit{init\_BankServer} \mathrel{\widehat{=}} [\, \mathit{BankServer}' \mid \mathit{bal}' = \{1 \mapsto 25, 2 \mapsto 5\} \wedge \mathit{pass}' = \{1 \mapsto 1111, 2 \mapsto 2222\} \wedge \mathit{log}' = \{(1, \mathit{deposit}, 15), (1, \mathit{deposit}, 10), (2, \mathit{deposit}, 5)\} \,]$$

To illustrate the functionality of the ATM the following schemas define the balance and credit operations. The first one informs the available money and the another deposits a certain amount of money. The last operation increments the balance by changing the function *bal* by means of the operator $\oplus$ that updates the function with the new balance:

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{Credit}} \\ \hline
\Delta \mathit{ATM} \\
\mathit{amount}? : \mathbb{R} \\
\mathit{dst}? : \mathbb{N}_1 \\
\mathit{report}! : \mathit{MESSAGE} \\ \hline
\mathit{bal}' = \mathit{bal} \oplus \{\mathit{dst}? \mapsto \mathit{bal}(\mathit{dst}?) + \mathit{amount}?\} \\
\mathit{log}' = \mathit{log} \cup \{(\mathit{dst}?, \mathit{deposit}, \mathit{amount}?)\} \\
\mathit{report}! = \mathit{ok} \\ \hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{Balance}} \\ \hline
\Xi \mathit{ATM} \\
\mathit{src}? : \mathbb{N}_1 \\
\mathit{amount}! : \mathbb{R} \\ \hline
\mathit{amount}! = \mathit{bal}(\mathit{src}?) \\ \hline
\end{array}$$

In order to validate requirements with the user, a visualization was developed, as Figure 9 shows. The scene is composed by different graphics objects such as money, ticket and ATM among others. The following scripts define the main components of the visualization:

- *Money*: When the client makes an extraction gets the money.

```
geometry(file('money.bsh')).
action(money,[moveFromTo(time(0,2),[point3d(-0.5,1,0.3),
    point3d(-0.5,1,2)])]).
event(stateChanged,op('z_WidthdrawOk'),[money]).
```

- *Ticket*: When the user makes an operation gets a ticket.

```
geometry(file(ticket.bsh')).
```

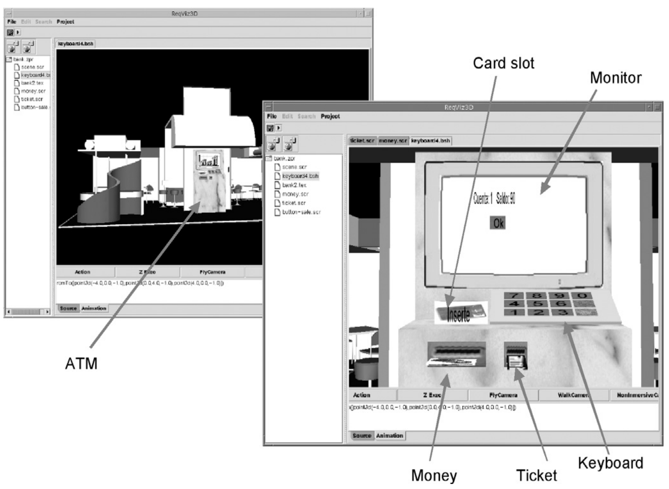


Figure 9 ATM

```
 action(ticket,[moveFromTo(time(0,2),[point3d(0.0,1.0,0.3),
     point3d(0.0,1.0,2.0)])]).
event(stateChanged,op('z_Balance'),[ticket]).
```

Finally, a bsh script integrates all graphic objects. Also, other graphic objects are created using JAVA3D:

- Creates a Z Executor object to animate the Z specification:

```
 ZAnimat Bank= new ZAnimat("bank.tex.pl");
```

- Composes the scene 3D:

```
J3DCompositeComponent World=
    new J3DCompositeComponent();
J3DCompositeComponent ATM=
    new J3DCompositeComponent();
J3DComponent money=Factory3D.instance().
    createViewForZ("money.scr",Bank);
ATM.addComponent(money);
J3DComponent ticket=Factory3D.instance().
    createViewForZ("ticket.scr",Bank);
ATM.addComponent(ticket);
.....
return World;
```

4.1 An architectural refinement

We have seen how to use Z to precisely capture software requirements and validate them using visualization techniques. However, in order to build the final system these specifications can be refined. In fact, this section illustrates a specification of the ATM at an architectural level, which in addition, can replace the initial specification and can be still validated using the visualization previously developed.

The basic building blocks of an architectural description are *components* and *connectors*. Components are computational entities of a system, which make tasks through internal computation and external communication with the rest of the system, while connectors represent the architectural interaction between components. For example, Figure 10 shows an architectural refinement of the ATM system, which includes a pipeline subsystem, a specialization of the style pipes and filters [31]. A filter (component) reads stream of data on its input and produces streams of data on its output. The pipes (connectors) perform as channels for the transmission of the data produced by the filters.

In order to connect components and connectors, both types of elements define a collection of interaction points: the *ports*. A port is an individual connection point, which defines a service that the element either offers or expects.

The complexity of a component may be high; it could be for example an entire subsystem. This means, that a component could be disaggregated in a collection of other components that interact by means of connector instances. This decomposition constitutes a *configuration*. Therefore, an architectural description is defined as hierarchical configurations of components and connectors as Figure 10 illustrates.

In order to animate specifications the architectural model is based on the notion of an *event*. An event is initiated in a port of a component and propagated to the port of the other element. A Z operation schema models an event. An event has a type, an instance that indicates the element that initiated it, input parameters for passing information about the event and output parameters for returning information.

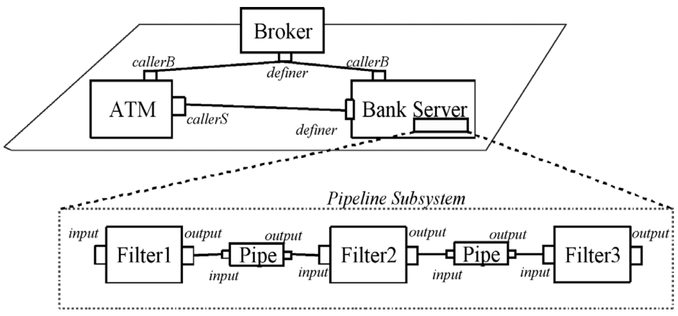


Figure 10 ATM architectural refinement

The refined specification of the ATM addresses new issues such as distribution. The system consists now in decoupled components that interact by remote services interaction (Broker Architectural Style [5]). The main components are shown in Figure 10. The *Bank Server* implements the services for managing accounts and making transactions. This component registers with the *Broker* indicating the services it provides. The *Broker* stores the name of the providers and is responsible for coordinating communication between clients and providers. The ATM implements user functionality: deposit, withdraw, transfer, balance and user authentication. The *Bank Server* provides transaction and accounting support. It is a complex component that is internally composed of other components. One of its components is a pipeline subsystem that implements transaction filtering and sorting operations.

The *Broker* provides support for registering services. The *Bank Server* registers itself with the *Broker* component via a registry event. The *Broker* stores the name of the provider and is responsible for coordinating communication between clients and providers.

$$MESSAGE ::= fail \mid ok$$

$$\begin{array}{|l}
\multicolumn{1}{l}{LookUp} \\ \hline
\Xi Broker \\
event? : EVENT \\
instance? : ELEMENT \\
inputP? : Name \\
outputP! : Component \\ \hline
event? = lookUp \\
outputP! = services(inputP?) \\ \hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{Broker} \\ \hline
definer : Port \\
services : Name \pfun Component \\ \hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{Registry} \\ \hline
\Delta Broker \\
event? : EVENT \\
instance? : ELEMENT \\
inputP? : Name \times Component \\
outputP! : MESSAGE \\ \hline
event? = registry \\
services' = services \oplus \{inputP!\} \\
outputP! = ok \\ \hline
\end{array}$$

When a client requests a service via a *lookUp* event, the broker locates the appropriate server. This implementation of the Broker realizes the Direct Communication Broker System variant, that is, clients can communicate with servers directly.

The ATM provides basic transactions support. In order to use the ATM machine, the user inserts his/her card and is prompted for a password. The password is validated and if it is correct, the client can make different transactions. In order to find an appropriate server it sends a *lookUp* event to the *Broker* by calling the *Initiate* schema operation. After that, the sentence *callerS.setConnection* links the *callerS* port with the *Bank Server* port, setting a channel for making requests.

$$\begin{array}{|l}
\textit{Connect}\\
\Delta ATM\\
\hline
callerB.Initiate(lookUp, This, bankserver, Res)\\
callerS.setConnection(Res.definer)\\
\hline
\end{array}$$

The ATM calls *BankServer* services for actually performing transactions. For example, the next Z schema models a *Credit* operation. The *ATM* calls bank services using *callerS* port. Using this port the *ATM* sends a *credit* event to the *Bank Server*:

$$\begin{array}{|l}
\textit{Credit}\\
\Delta ATM\\
amount? : \mathbb{R}\\
dst? : \mathbb{N}\\
report! : MESSAGE\\
\hline
callerS.Initiate(credit, This, amount?, dst?, report!)\\
\hline
\end{array}$$

The *BankServer* implements basic transaction and log operations. The *BankServer* registers with the *Broker* with the *InitService* schema. The following schemas model this component:

$$\begin{array}{|l}
\textit{InitService}\\
\Delta BankServer\\
\hline
callerB.Initiate(registry, This, bankserver \mapsto This, Res)\\
\hline
\end{array}$$

$$\begin{array}{|l}
\textit{BankServer}\\
callerB : Port\\
definer : Port\\
pipeline : Port\\
bal : \mathbb{N} \nrightarrow \mathbb{R}\\
pass : \mathbb{N} \nrightarrow \mathbb{N}\\
log : \mathbb{P}(\mathbb{N} \times TRANSACTION \times \mathbb{R})\\
\hline
\end{array}$$

$$\begin{array}{|l}
\textit{Credit}\\
\Delta BankServer\\
event : EVENT\\
instance : ELEMENT\\
amount? : \mathbb{R}\\
dst? : \mathbb{N}\\
report! : MESSAGEINIT\\
\hline
event = credit\\
bal' = bal \oplus \{dst? \mapsto bal(dst?) + amount?\}\\
log' = log \cup \{(dst?, deposit, amount?)\}\\
report! = ok\\
\hline
\end{array}$$

As a result of modeling the system at an architectural level, we provide a way to reason about software systems at an adequate level of abstraction. In fact, it has been recognized that finding an appropriate architectural design for a system is a key element of its long-term success.

5. THE TOOL

In essence, REQVIZ3D takes a specification as input and generates a visualization as output, through which users can validate requirements. Firstly, the system requirements are specified in Z using an editor as Figure 11 shows. After that, their respective graphical representations are also built with ReqViZ3D. Finally, the requirements are animated and visualized, and so validated by the user. For example, Figure 11 also presents the visualization of a vending machine. A client inserts coins in the coin slot. After that, if the required amount of money was inserted, the client obtains a can by pressing the eject button. The machine can dispense a limited number of cans. The vending machine has two buttons: one for inserting coins and the other for ejecting the can. When the correct amount of money is inserted the eject button is lit on, then the user can press it and the can is ejected and the button is lit off.

REQVIZ3D was developed in JAVA. In order to animate a Z specification, it is translated to PROLOG and executed.

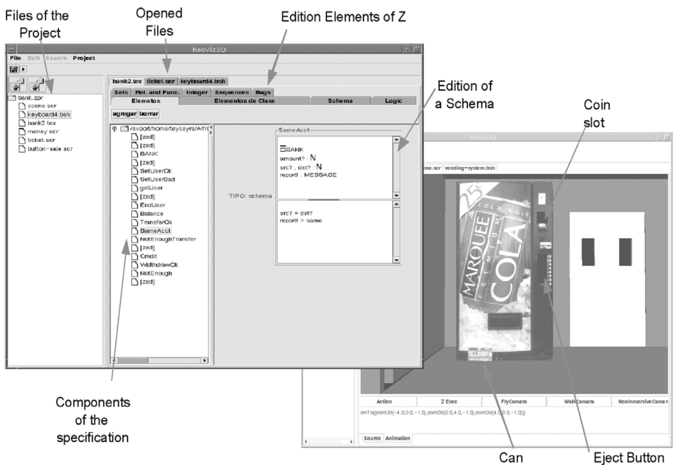


Figure 11 REQVIZ3D tool

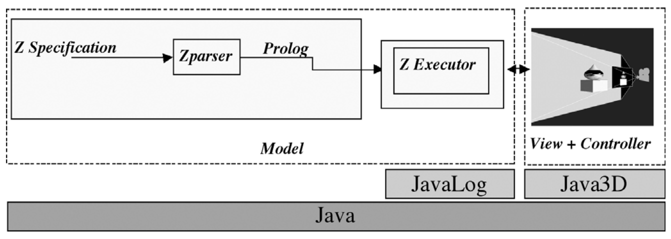


Figure 12 Global system view

As we developed REQVIZ3D in JAVA, a way to integrate JAVA and PROLOG was needed. As we have seen this integration was done using JAVALOG. Also, trying to take advantage of 3D visualizations we developed the View subsystem on top of JAVA3D. Figure 12 presents a global system view of REQVIZ3D that defines a blueprint of the overall structure of the application and corresponds to the architectural model Model-View-Controller [5]. This model prescribes the division of an interactive application in three parts, the Model that represents the application functionality, the View responsible for the output interface and the Controller responsible for the input handling.

6. CONCLUSIONS

The main contribution of this work is the utilization of 3D visualization techniques to reduce the communication gap between the customer and the developer resulting in a much more effective process of requirements validation: the system is described in a way that users can understand. Therefore, as a consequence of validating requirements in the earlier stages of the development process, the total effort to develop a system is reduced.

In addition, a prototype tool to visualize requirements was developed. This tool assists the developer in several stages in the development process: from requirements specification in Z and definition of graphical objects, to animation and execution of requirements in a 3D world.

Several examples were presented showing that the use of visualization techniques were very useful in analyzing their dynamics. In addition, we have presented an architectural refinement of a case study, which can replace the initial system specification and can be still validated using the visualization initially developed.

Three-dimensional graphics were used in the construction of the visualizations. Their similitude with the real world enables us to represent it in a more natural way than 2D. This means that the representation of the objects can be done according to its associated real concept. However, the construction of 3D graphics presentations is a difficult and time-consuming task, besides it requires specific knowledge and even artistic skills. Therefore, a future extension includes the automatization of the presentation extending the ideas presented in several works about the automatic generation of presentations [23,39].

At last, the work combines an informal approach (visualization) with a formal light one, resulting in a more effective technique. In that sense, a light application of formal methods can be an economical way to improve the quality of specifications without using formal proofs.
Others future extensions include supporting OBJECTZ as specification language and provide a basic library of 3D graphics components.

REFERENCES

1 **Antonio Alencar and Joseph Goguen.** OOZE. In *Object Orientation in Z*, pages 79–94. Susan Stepney, Rosalind Barden, and David Cooper editors, Springer, 1992.

2 **A. Amandi, A. Zunino, and R. Iturregui.** Multi-paradigm languages supporting multi-agent development. In *MAAMAW'99*, pages 128–139, 1999.

3 **Barry Boehm.** Requirements that handle ikiwisi, cots, and rapid change. *Computer*, 33(7):99–102, 2000.

4 **P. Breuer and J. Bowen.** Towards Correct Executable Semantics for Z. In *Proc. 8th Z Users Workshop (ZUM)*, pages 185–212. Springer-Verlag, 1994.

5 **Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal.** *Pattern-Oriented Software Architecture: A System of Patterns*. John Wiley & Sons, 1996.

6 **Stuart Card, Jock MacKinlay, and Ben Shneiderman, editors.** *Readings in Information Visualization: Using Vision to Think*. Morgan Kaufmann, 1998.

7 **Roger Duke, Paul King, Gordon Rose, and Graeme Smith.** The Object-Z specification language, version 1. *Technical Report* 91–1, Software Verification Research Centre, Department of Computer Science, The University of Queensland, Australia, 1991.

8 **A. S. Evans.** Specifying and verifying concurrent systems using Z. *Lecture Notes in Computer Science*, 873, 1994.

9 **N. Fuchs.** Specifications are (preferably) executable. *IEEE Software Engineering Journal*, 7(5):323–334, September 1992.

10 **J. P. Diaz Gonzalez and J. E. Urban.** Language aspects of ENVISAGER. an object-oriented environment for the specification of real-time systems. *Computer Languages*, 16(1):19–37, 1991.

11 **Anthony Hall.** Using z as a specification calculus for object-oriented systems. In Dines Bjørner, C. A. R. Hoare, and H. Langmaack, editors, *VDM'90: VDM and Z-Formal Methods in Software Development*, volume 428 of *Lecture Notes in Computer Science*, pages 290–318. Springer-Verlag, 1990.

12 **D. Hazel, P. Strooper, and O. Traynor.** Possum: An animator for the SUM specification language. In *Proceedings: 4th Asia-Pacific Software Engineering and International Computer Science Conference*, pages 42–51. IEEE Computer Society Press, 1997.

13 **Johann Hörl and Bernhard K. Aichernig.** Validating voice communication requirements using lightweight formal methods. *IEEE Software*, 17(3):21–27, MayJune 2000.

14 ISO. Vrml97, international specification. *Technical report*, ISO, 1997.

15 **Daniel Jackson and Jeanette Wing.** Lightweight Formal Methods. *IEEE Computer*, 29(4):22–23, April 1996.

16 **Cliff B. Jones.** *Systematic Software Development Using VDM*. Prentice-Hall International, Englewood Cliffs, New Jersey, second edition, 1990. ISBN 0-13-880733-7.

17 **Cliff B. Jones.** Formal methods light: A rigorous approach to formal methods. *Computer*, 29(4):20–21, April 1996.

18 **John C. Kelly, Joseph S. Sherif, and Jonathan Hops.** An analysis of defect densities found during software inspections. *The Journal of Systems and Software*, 17(2):111–117, February 1992.

19 **Hideki Koike.** The role of another spatial dimension in software visualization. *ACM Transactions on Information Systems*, 11(3):266–286, 1993.

20 **Kevin Lano.** The B Language and Method: *A guide to Practical Formal Development*. Springer Verlag London Ltd., 1996.

21 **Kevin C. Lano.** Z++. In Susan Stepney, Rosalind Barden, and David Cooper, editors, *Object Orientation in Z, Workshops in Computing*, pages 106–112. Springer-Verlag, Cambridge CB2 1LQ, UK, 1992.

22 **J. D. Mackinlay, G. G. Robertson, and S.K. Card.** The perspective wall: Detail and context smoothly integrated. In *Proceedings of ACM CHI'91*, pages 173–179, 1991.

23 **Jock Mackinlay.** Automating the design of graphical presentations of relational information. *ACM Transactions on Graphics,* 5(2):110–141, April 1986.

24 **Bertrand Meyer.** On formalism in specifications. *IEEE Software*, 2(1):6–26, January 1995.

25 **Pat Niemeyer.** BeanShell. http://www.beanshell.org, 1999.

26 **M. B. Ozcan, P. W. Parry, I. C. Morrey, and J. I. Siddiqi.** Visualisation of executable formal specifications for user validation. *Lecture Notes in Computer Science*, 1385, 1998.

27 **Ben Potter, Jane Sinclair, and David Till.** *An Introduction to Formal Specification and Z.* Prentice Hall, New York, 1991.

28 **C. Potts.** Expediency and appropriate technology: An agenda for requirements engineering research in the 1990s. *Lecture Notes in Computer Science*, 550, 1991.

29 **P. Pulli, R. Elmstrom, G. Leon, and de la Puente.** IPTES – incremental prototyping technology for embedded real-time systems. *Technical report*, ESPRIT, 1991.

30 **Manojit Sarkar and Marc H. Brown.** Graphical fisheye views. *Technical Report* CS-93-40, Department of Computer Science, Brown University Providence, September 1993.

31 **M. Shaw and D. Garlan.** *Software Architecture: Perspective on an Emerging Discipline*. Prentice-Hall, 1996.

32 **I. Sommerville.** *Software Engineering*, 5/e. Addison-Wesley Publishing Company, 1996.

33 **H. Sowizral, K. Rushforth, and M. Deering.** *The Java 3D API Specification*. Addison-Wesley, 1998.

34 **J. M. Spivey.** *The Z Notation. Prentice Hall International*, UK, 2 edition, 1992.

35 **S. Stepney, R. Barden, and D. Cooper, editors.** *Object Orientation in Z. Workshops in Computing*. Springer-Verlag, 1992.

36 **L. Sterling, P. Ciancarini, and T. Turnidge.** On the Animation of Not Executable Specifications by Prolog. *Int. Journal on SE and KE*, 6(1):6--88, 1996.

37 **Peter Wegner.** Dimensions of object-based language design. In *Conference proceedings on Object-oriented programming systems, languages and applications,* pages 168–182. ACM Press, 1987.

38 **P. J. Whysall and J. A. McDermid.** *An approach to object-oriented specification using* Z. In J. E. Nicholls, editor, Z User

Workshop, Oxford 1990, *Workshops in Computing*, pages 193–215. Springer-Verlag, 1991.

39 **Michelle X. Zhou.** Automated visual discourse synthesis: Coherence, versatility, and interactivity. In *Proceedings of ACM CHI 98 Conference on Human Factors in Computing Systems (Summary)*, volume 2 of Doctoral Consortium, pages 76–77, 1998.