Scenario-Based Techniques for Supporting the Elaboration and the Validation of Formal Requirements

PATRICK HEYMANS

Institut d'Informatique, University of Namur
Rue Grandgagnage 21, B-5000 Namur, Belgium
Patrick.Heymans@info.fundp.ac.be

ERIC DUBOIS

S.W.I.F.T,
Avenue Adèle 1, B-1310 La Hulpe, Belgium
Eric.Dubois@swift.com

CREWS report 98-30
Technical Report of the University of Namur
October 1998

This is a draft version of a paper published in the Requirements Engineering Journal (1998) 3:202-218, Springer-Verlag .

Abstract

Developing complex, safety critical systems requires precise, unambiguous specification of requirements. A formal specification language is thus well suited to this task. Formal specification languages require, but also exacerbate, the need for tools. In particular, tools should support the elaboration (how to build the formal specification?) and the validation (how to check the adequacy of the specification towards the informal needs of the various stakeholders?).

This paper focuses on the language Albert II, a formal language designed for the purpose of expressing requirements for distributed real-time systems. It presents two contributions supporting its use. The first contribution aims at improving the elaboration process by providing a method for constructing an Albert II description from scenarios expressing the stakeholders' requirements. These are represented through Message Sequence Charts extended to deal with composite systems. The second contribution takes the form of a requirements validation tool (a so-called animator) that the stakeholders can use interactively and cooperatively in order to explore different possible behaviours (or instance-level scenarios) of the future system. These behaviours are automatically checked against the formal requirements specification.

***Keywords** : Scenarios, Formal Methods, Animation, Message Sequence Charts.*

1 Introduction

In this paper, we study two possible roles played by scenarios[a] [35,4] in the context of producing Software Requirements Documents (SRD) for real-time distributed systems. More specifically, we will focus on the functional part of the SRD and will use a formal requirements language for expressing its content.

The SRD is the central output of the Requirements Engineering (RE) process. A large variety of stakeholders with different backgrounds (users, customers, domain experts, designers, maintainers, etc.), are involved in this process. To ensure readability by each of them, this document is written in natural language (sometimes complemented with diagrams and figures). Several standards (see [39] for example) and authors (see, e.g., Meyer [27]) have produced recommendations aimed at improving the quality of the produced document. Besides usual qualities like completeness and consistency, there are some rules related to the structure of the natural language statements (e.g., like 'no more than one verb per sentence'). Of particular interest also are recent recommendations about the exact nature of a requirement. For example, Jackson and Zave [44,23] make clear that (i) requirements have to be grounded in the reality of the problem environment and that (ii) it should be possible in the SRD to make a clear distinction between *indicative* statements related to the behaviour of the environment and *optative* statements (that is, the requirements) associated with the desired system to be installed.

Throughout the paper, we will illustrate our ideas by exemplifying them using the *Lift System* case study inspired by that presented at IWSSD'87 (we refer to [40] for an informal introduction). In Figure 1, we present a *context diagram* (an old idea from structured analysis recently reinstated by Jackson [23]) to keep track of the interactions of the system (the *Controller*) with its environment. Below, a set of optative statements associated with the *Controller* are presented. Such requirements are carefully structured and written in English :

- The controller can send a request to any floor to activate the door-opening mechanism.
- Resetting a button has to happen simultaneously with the request for opening a door.
- The controller sends requests for opening a door at the floor where the door is located.

[a] Our research on scenarios and animation is supported by the EU-funded ESPRIT LTR research project 21.903 CREWS (Cooperative Requirements Engineering With Scenarios) and by the Wallon Region's CAT project.

▪ …

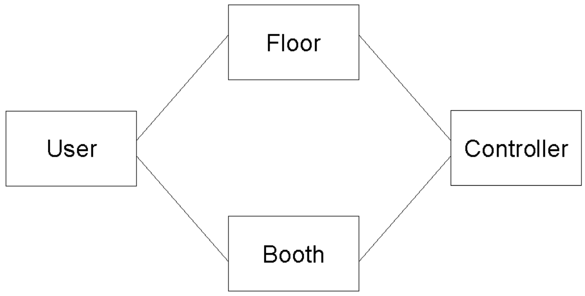

Figure 1 : Context diagram of the lift system.

Writing the SRD is a complex activity where the analyst should start from various, redundant, incomplete and fragmented sources of information coming from the different stakeholders. To support his task, various techniques, relying on suitable ontologies of concepts, have been proposed. They help in the clarification and the structuring of the different knowledge pieces given by the stakeholders. For example, the use of an Entity-Relationship model has proved useful for modelling the information aspects (e.g., see Wieringa [42] for an example of the use of these techniques at the RE level). For more complex systems (like safety-critical systems), more formal techniques are useful to analyse properties of the critical part of the system behaviour and validate this part with a high degree of certainty. In this paper, because we are concerned with application domains pertaining to real-time *composite systems* [11], we will consider the Albert II language, a formal requirements language that we designed in 1992 and which is the topic of the development of several tools [10] and of practical real-size industrial experiences [43].

Albert II was designed with *naturalness* in mind; i.e., it aims at preserving the structure of the informal requirements expressed by the stakeholders as far as compatible with a formal semantics. This helps in maintaining traceability links between the SRD and the formalised SRD (see Figure 2), and in the validation by stakeholders. In Section 2, we introduce the Albert II language and illustrate its use.

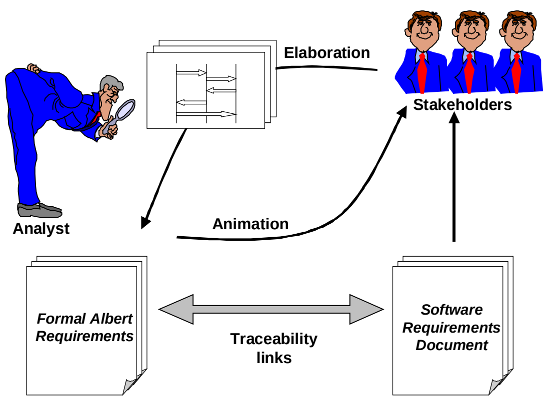

Figure 2 : Suggested requirements engineering process.

Producing the formal Albert II requirements document is not an easier activity than producing the informal SRD from scratch. However, because the language relies on formal semantics and on a suitable ontology of concepts, more guidance can be provided to the analyst. In this paper, we will focus on the support provided by the language for the analyst when interacting with the stakeholders. In particular, we will suggest how a *scenario*-based approach can enrich these interactions.

First of all, it is clear that the elaboration of an Albert II requirements document cannot be made in one shot. Several iterations are required during which various sources of information (see Figure 2) are analyzed, discussed and exploited. To address this problem, in the recent past, we studied the possibility of using (diagrammatic) semi-formal notations before producing the formal document [43]. In particular, MSCs (Message Sequence Charts [41]) are of particular interest for specifying reactive composite systems. Therefore, we have decided to adopt them and integrate them with the Albert II language. At this level, integration means to give a definition to MSCs which is compatible with the one underlying Albert II. For example, in Albert II, because actions can have an external effect, we need to enrich classical MSCs. Another advantage coming from this conceptual proximity is the possibility to express rules and heuristics guiding the elaboration of Albert II requirements from the MSCs produced. The result of this work is presented in Section 3.

Another basic problem of formal notation is readability. In particular, at the RE level, it is clear that we cannot expect our various stakeholders to read mathematical formulae. As the reader will discover in Section 2, an Albert II specification is also difficult to read. However, thanks to the traceability property indicated above, informal statements can be produced in a semi-automated way from the formal statements and integrated in the SRD. But the SRD is probably not sufficient for the RE validation activity where all stakeholders have to agree on the behaviour of the system described in the SRD. For complex systems, reading the SRD document carefully (even several times) does not suffice for stakeholders to get a precise idea of the system's behaviour in all situations (in particular, the abnormal situations). This is why we are developing a so-called *animator* tool which allows stakeholders to cooperatively explore different possible behaviours of the future system (as allowed by the formal Albert II description). The purpose of the tool comes down to testing if a given scenario proposed by one or several stakeholders is compatible with the requirements specification. In Section 4, we present the functionalities offered by this tool and we briefly discuss its architecture.

2 The Albert II language

Albert II [8,9] is a formal requirements specification language based on a real-time temporal logic [5]. The language has been designed in 1992 and from that time has been validated (and revised) through the specification of non-trivial systems like Computer Integrated Manufacturing (CIM, see [7]), process control and telecommunications systems [43]. Besides supporting the distinction made between optative and indicative statements (see above), this language is also characterized by:

- its naturalness (see definition in previous section), where the objective is to avoid the introduction of extra elements (overspecifications) in the formal specification which do not have a counterpart in stakeholders' concepts. This naturalness property is guaranteed by the possibility to write requirements by adopting an *operational* and/or a *declarative* style of specification;
- the existence of various *templates* associated with specific categories of statements. These templates provide methodological guidelines to the analyst in eliciting and structuring Albert II specifications.

Such a specification is made of (i) a graphical specification component in which the vocabulary of the specification is *declared* and of (ii) a textual specification component in which the admissible behaviours of agents are *constrained* through temporal logic formulae organized in terms of the templates introduced above.

Albert II organizes its specification around the agents identified in the environment and in the system. An agent is an autonomous entity that can perform or suffer *actions* which change or maintain its *state* (either physical state or state of knowledge about the external world) and/or the states of other agents. Actions are performed by agents to discharge contractual obligations expressed in terms of *local constraints*, applicable to the agent itself, and *cooperation constraints*, that apply to the interactions between agents.

2.1 Graphical declarations

Hereafter, we illustrate the concepts of the language by applying them to the specification of the lift system case study introduced in Section 1. Figure 3 contains the graphical declaration of the *LiftComplex* (made of the *Controller* and its environment) according to the Albert II conventions. For lack of space, we have omitted the *TopFloor* and *GroundFloor* agents. Note that this declaration provides a refinement of the context diagram of Figure 1.

Each agent is represented by an oval and multiplicity is indicated by shadowing the ovals. Figure 3 also declares the internal structure of the *Controller* agent. It declares the state structure and the actions that may happen during the lifetime of an agent and which may change the state of the agent (and possibly of other agents). State components are represented by (normal) rectangles and actions are represented by rectangles with rounded corners. We also note that state components are typed and that actions can have typed arguments.

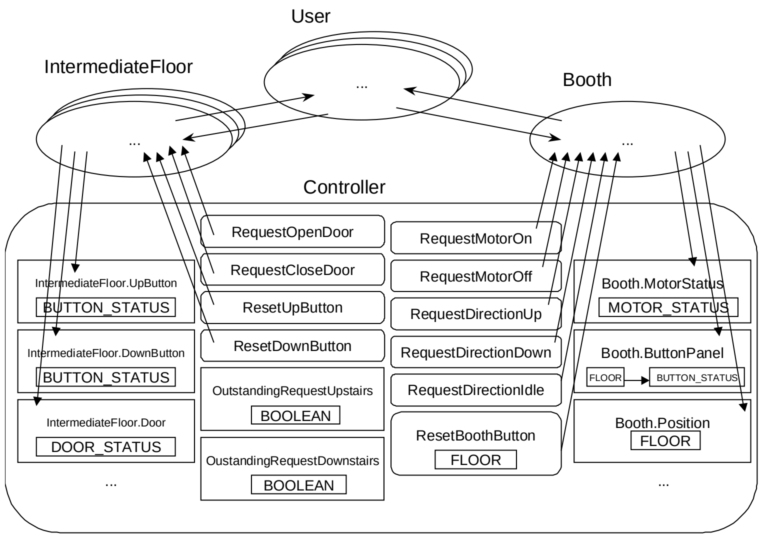

Figure 3 : Graphical declarations of LiftComplex.

Types may vary from simple data types to complex data types (recursively built using the usual data types constructors like *set*, *sequence*, *table*, etc.). The structure of types is not defined graphically and appears in the *Data Type* portion of an Albert II specification. For example, *FLOOR* is a user-defined type associated with the identity of the different floors (that is, the *TopFloor*, the *GroundFloor*, and all the *IntermediateFloor*s) and on which are defined a *Next* function (access to the floor directly above) and the *Higher* and *Lower* predicates which test if a floor is above or below another floor.

The information provided in Figure 3 is informally rephrased in the first part (*Declarations*) of the specification (examples are given further in this section). Informal descriptions of declarations play the role of *designations* [23]. From graphical conventions used in Figure 3, we understand that *Button-*

Panel is a table indexed on *FLOOR* while *Position* is a single time-varying component of type *FLOOR*. Finally, the boolean value of the *OutstandingRequestUpstairs* component is derived from the values of other components (these derivation links have been omitted in the figure for clarity).

In addition, the graphical notation also expresses visibility relationships linking agents to the outside. Arrows in Figure 3 show (i) how agents make information (actions and state components) visible to other agents (e.g., the *RequestMotorOn* action is made visible by the *Controller* to the *Booth*) and (ii) how external agents may influence the agent's behaviour through exportation of information (the *Controller* is influenced, e.g., by the *UpButton* component of the *IntermediateFloor*).

Finally, it is important to note that, in Figure 3, all the information managed by the *Controller* system is shared by the *Controller* with its environment. This typically results from the indicative/optative statements perspective we adopted: statements associated with the *Controller* are optative and expressed in terms of information shared with the external entities[b]; indicative statements (whose values are possibly shared with the *Controller*) are expressed at the level of the agents located in the *Controller*'s environment.

2.2 Classification of properties

Besides graphical declarations, textual constraints are used for pruning the (usually infinite) set of possible lives associated with the agents of a system. As explained above, Albert II supports two styles of specification. To guide the analyst in eliciting and structuring requirements specifications, the constraints are further classified into categories for each of which a characteristic *template* is defined. The existence of these different templates results from the performance of several case studies and the identification of typical patterns associated with the textual informal requirements. Besides *Basic constraints*, *Local constraints* define the behaviour of an agent using an operational (event-condition-action) and/or declarative style of specification. Finally, *Cooperation constraints* refine the importation/exportation links introduced in the graphical declaration.

Hereafter, based on the example, we briefly describe some important templates. We also provide a fragment of the textual specification associated with the *Controller*. A more complete account of the language can be found in [9].

2.2.1 Basic constraints

Derived Components constraints express how the value of state components can be computed from other state components (see examples in the specification below).

Initial Valuation constraints allow to fix the initial value of a state component. This type of constraint can express, for example, that initially the motor is off and that there is no pressed button in the button panel:

INITIAL VALUATION

MotorStatus = Off
ButtonPanel[_] = *Idle*

2.2.2 Declarative constraints

---

[b] with the exception of *Derived Components* which just play the role of intermediate components that are used for simplifying the expression of requirements.

State Behaviour constraints express restrictions on the possible values that can be taken by the state components forming the state of an agent. These restrictions can be static (i.e., invariants which hold at any time) or dynamic (i.e., depending on time). As an example, here is an indicative property of the *Booth* agent telling that, in order to go from up to down (or down to up), the booth's motor must go through a neutral position:

STATE BEHAVIOUR

MotorDirection = *Up* Until! *MotorDirection* = *Idle*
MotorDirection = *Down* Until! *MotorDirection* = *Idle*

Duration constraints put restrictions on the duration of action occurrences. For example, in the definition of the *Floor* agent, it is said that closing the door takes between 0.8 and 1 second:

ACTION DURATION

$| \mathit{CloseDoor} | \geq 0.8\ \mathit{sec}$
$| \mathit{CloseDoor} | \leq 1\ \mathit{sec}$

Action composition constraints define how actions may be refined in terms of finer-grained component actions. They are also used to express desired temporal relationships between action occurrences (see example in the specification below).

2.2.3 Operational constraints

Effects of actions constraints express how the occurrence of an action (either from the agent or from the outside) changes the state of an agent. Since actions can have durations, Albert II makes it possible to distinguish between a *pre-effect* and a *post-effect* associated with an action occurrence. In the agent *Floor*, for example, the action *CloseDoor* has the following effects:

EFFECTS OF ACTIONS

CloseDoor : *DoorStatus* := *Closing*
[ ]
DoorStatus := *Closed*

Preconditions describe under which conditions an action can occur (see example in the specification below).

Triggering constraints express state conditions under which an action has to occur (see example in the specification below).

2.2.4 Cooperation constraints

Examples of these constraints are given in the specification below.

State/Action perception constraints define under which condition an agent is sensitive to information (part of state or occurrence of action) provided by other agents.

State/Action information constraints define under which condition an agent shows part of its state or occurrences of actions it performs, to other agents.

Agent : **LiftComplex.Controller**

DECLARATIONS

STATE COMPONENTS

OutstandingRequestUpstairs
  instance-of *BOOLEAN*
  derived-from *Booth.Position, Booth.ButtonPanel, GroundFloor.UpButton, TopFloor.DownButton, IntermediateFloor.UpButton, IntermediateFloor.DownButton*

> The controller knows that there are outstanding requests upstairs (wrt the floor where the booth is located) from the status of buttons in the booth and at the floors.

…

ACTIONS

**RequestOpenDoor* → *LiftComplex.IntermediateFloor, LiftComplex.TopFloor, LiftComplex.GroundFloor*

> The controller can send a request to some floor to activate the door opening mechanism. Such an action is instantaneous[c]

…

BASIC CONSTRAINTS

DERIVED COMPONENTS

$$OutstandingRequestUpstairs \triangleq \exists\, d : (\, Higher(d, b.Position) \wedge (\, b.ButtonPanel(d) = Pressed \vee d.UpButton = Pressed \vee d.DownButton = Pressed\,)\,)$$

> There is an outstanding request upstairs iff a button is pushed at some floor higher than the one where the booth is located or a button in the booth referring to some higher floor is pressed

…

[c] Indicated by the * symbol.

DECLARATIVE CONSTRAINTS

ACTION COMPOSITION

{*ResetUpButton, ResetDownButton, ResetBoothButton*}

$$comp \leftrightarrow \ RequestOpenDoor \quad \Leftrightarrow (\ ResetUpButton \oplus \mathsf{DAC}\ ) \quad \Leftrightarrow (\ ResetDownButton \oplus \mathsf{DAC}\ ) \quad \Leftrightarrow (\ ResetBoothButton(x) \oplus \mathsf{DAC}\ ) \ \underline{\text{with}}\ x = b.Position$$ [d]

`Resetting a button has to happen simultaneously with the request for opening a door.`

…

OPERATIONAL CONSTRAINTS

PRECONDITION

$$ResetUpButton: \quad b.Position.Door = Closed \ \wedge\ b.MotorStatus = Off \ \wedge\ b.Position.UpButton = Pressed \ \wedge\ (\ (\ b.MotorDirection = Up \wedge\ OutstandingRequestUpstairs\ ) \ \vee\ (\neg OutstandingRequestsUpstairs \wedge\ \neg OutstandingRequestsDownstairs)\ )$$

```
The controller can reset the up-button only if : the door is closed, the motor is off, the
up-button at the current floor is pressed and if one of the following is true:
-   the direction of the motor is up and there are outstanding requests upstairs,
-   there is no outstanding request at all.
```

…

TRIGGERRINGS

$b.Position.Door = Open\ /\ 10\ sec \rightarrow RequestCloseDoor$

```
The controller requests the current floor to activate the door closing mechanism when the
door has been open at this floor for ten seconds.
```

…

COOPERATION CONSTRAINTS

STATE PERCEPTION

$\mathcal{K}\ (b.Position\ /\ \mathsf{TRUE})$

`The controller always perceives the booth’s position.`

…

ACTION INFORMATION

$\mathcal{XK}\ (RequestOpenDoor.f\ \ /\ f = b.Position)$

```
The controller sends requests for opening the door at the floor where the booth is lo-
cated.
```

…

In the specification fragment above, we can see the informal comments that paraphrase each formal statement (we are currently developing a tool to help in the semi-automatic reformulation of the for-

[d] In the Albert II statement, $\leftrightarrow$ relates the composed action to its component actions. x $\Leftrightarrow$ y means that x and y have to start and end at the same times. x $\oplus$ y means that either x or y takes place. DAC means “Dummy Action”, that is, no action takes place. Finally, a sequence of action names between curly braces means that occurrences of these actions have to take place as part of a composed action occurrence.

mal statements in natural language). These comments are the basic inputs of the final SRD document (see the example requirements in the introduction).

2.3 Semantic issues

The semantics of an Albert II specification are given in two equivalent ways. The first defines which lives (models) are admissible for each agent. Agent lives are then combined to form lives of the society to which it belongs. The second semantics translates the Albert II language to a simpler real-time first-order logic called Albert$_{\mathbf{KERNEL}}$. It is important to note that at the level of a model[e] we talk about *agent instances* while, at the level of the specification, agents (classes) are declared. In a particular model and unless the agent is declared to be unique in the specification, there can be arbitrarily many instances of an agent. While the number of instances can vary from a model to another, it remains constant within a particular model.

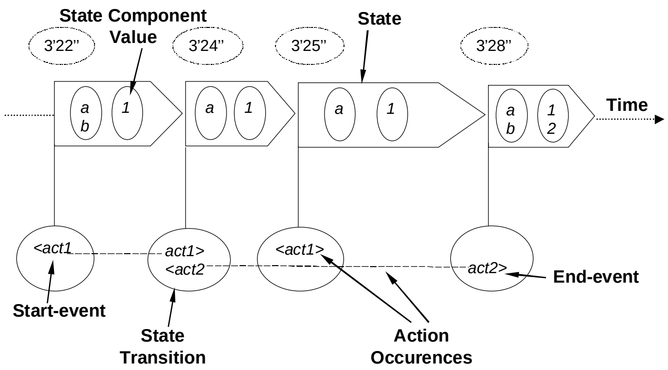

Figure 4 : Part of the life of an agent instance.

The life of an agent instance is an infinite alternate sequence of states and state transitions (see Figure 4). The sequence is indexed by a real-time value which increases throughout the sequence.

A *state* of an agent instance represents the value of all its state components in a time interval during which they remain unchanged. A *state transition* groups together all the events that affect an agent instance at a given point in time. The Albert$_{\mathbf{KERNEL}}$ notion of *event* is necessary to give a semantics to the Albert II notion of action. In fact, action occurrences can have a duration (e.g., *act2*) or can be instantaneous (as the second occurrence of *act1*). Actions have therefore been associated with events : each action has a *start*- and an *end-event* which, in the case of instantaneous action occurrences, happen at the same time.

Finally, a model (or *admissible life*, or *behaviour*) of the specification is built by combining the lives of several agent instances (at least one per class and exactly one per single agent), i.e. putting them on a common time line by adding states (if needed) and checking compatibility wrt cooperation constraints. Such a model defines an admissible behaviour of a composite system.

[e] To be understood as 'a model of a society'.

3 Supporting the elaboration of an Albert II specification with ASCs

In the previous section, we have shown fragments of an Albert II specification as it results from performing a complete RE process. It is obvious that such a process is complex and past experiences have shown that the analyst could not produce the final Albert II output from scratch without considering intermediate products.

In this respect, we experienced [43] that semi-formal techniques are very useful during the initial stages of the specification's elaboration. For example, a context diagram (like the one presented in Figure 1) helps in identifying the various agents of the system while a class-relationship diagram (like an ERA diagram) can be used to discover the various static elements of the problem (agents, state components and data types in the final Albert II specification). Both these notations are helpful mainly because each of them focuses on one aspect of the problem at a time, in this case agents identification and information modelling.

Another key aspect of the problem is of course behaviour. From our point of view, as we want to provide efficient ways to deal with complex systems, we feel an exacerbated need for a specific intermediate representation of behaviour which would (i) allow representation of knowledge expressed by stakeholders and (ii) serve as a starting point for elaboration of the final specification. It was clear to us that such a notation had to be based on the notion of scenario. Scenarios are now becoming widely recognised as a very useful way to achieve a better understanding of stakeholders' requirements. One of their main advantages is that each of them is related to the modelling of a very restricted part of the system's behaviour at a time. Therefore scenarios avoid the analyst getting lost in the complexity of tackling the entire behaviour. But, on the other hand, the pieces of information conveyed by the scenarios have to be consolidated to produce the final abstract specification.

In this section, we propose to further detail the behaviour issue by suggesting (i) how we have extended some usual scenario notation (viz. MSCs) to make it closer to the concepts required for Albert II and (ii) what kinds of information from the scenarios can be exploited in the elaboration of a specification. In the following subsections, we suggest a progressive abstraction process made of three main steps called (i) Scenario Definition, (ii) Scenario Analysis and (iii) Scenario Consolidation (see Figure 5). The process relies on a scenario notation that is defined in the first subsection.

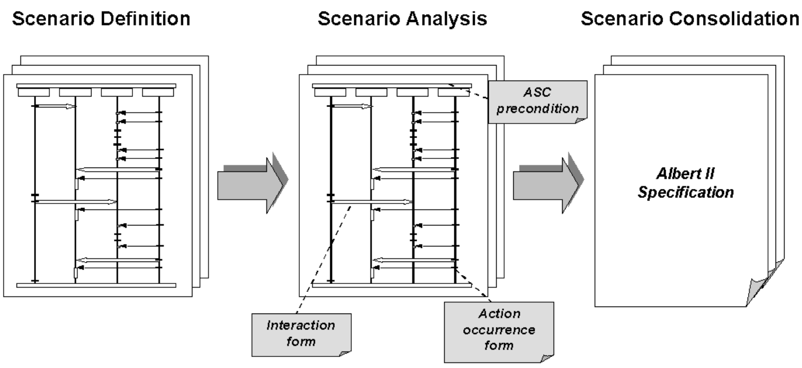

Figure 5 : Scenario-based elaboration of an Albert specification.

3.1 Scenario Definition

Our contribution to support Scenario Definition mainly resides at the product level. We provide a notation for the analyst to write scenarios no matter how he chooses to elicit them. As a process, he could use, for example, goal-oriented scenario authoring techniques [3] or elicitation based on real-world scenes [17][f] and which are compatible with the notation we propose. This notation is based on MSCs. The reasons for this are that :

- MSC-like notations are intuitive and therefore facilitate communication and understanding between the analysts and the stakeholders;
- MSC-like notations are particularly well suited for expressing interactions (as opposed to internal actions) which is precisely what one should focus on at the RE activity level (adopting a black-box view of the system).

Action Sequence Charts (ASCs) are a home-made variant of MSCs modified in order to deal with descriptions of composite systems. ASCs focus on the description of possible sequences of action occurrences performed by agent instances in order to fulfil some goal or functionality. The major difference between ASCs and MSCs is that, using MSCs (such as those that can be found in [41,33,2,24]), one can only describe sequences of messages and internal actions, making it impossible to model interactions other than message exchanges. This might not be a restriction when describing the behaviour of systems (like telecommunication systems) where the protocol for interacting with the environment is fixed. But this is not suitable for RE where, as we have seen in Section 2, we focus on the role of the system in terms of its environment without defining the 'how', i.e., the internals of the system and its message exchanges with the environment. At the RE level, one should be able to focus on actions (among which are physical actions such as moving, pushing a button or manufacturing a car part) and indicate their effect (with possible re-actions) on the agent that performs them and/or on other agents.

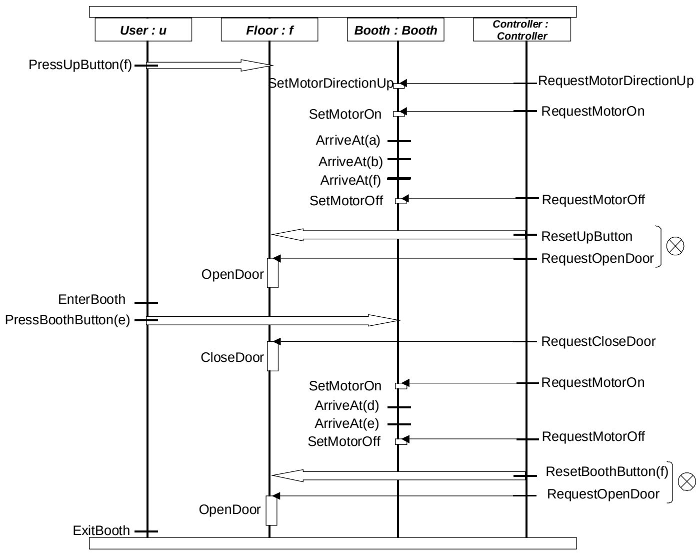

Figure 6 : Action sequence chart for a user moving to a higher floor.

In Figure 6, we illustrate the use of our ASCs by presenting a typical scenario associated with a typical flow of events allowing a user to move from a floor to a higher floor. In an ASC, an action is represented by a dash if it is instantaneous (the beginning and end events happen at the same time) and by a rectangle if it has a duration (the end event is after the beginning event). An arrow between agent instances A and B, starting from the beginning and/or end[g] of an action occurrence of A, indicates whether the action occurrence is shared with B, viz. if it has an effect on B (thick arrow) or if it sends a message to B (thin arrow). We call such arrows interactions. Interactions and events which are on a

[f] The elicitation techniques given as an example are other research topics of the CREWS project with which we participate. Integration of the complementary approaches is an issue currently addressed in the project.

[g] An arrow can only depart simultaneously from the beginning and the end of an action occurrence if the action is instantaneous.

same horizontal line happen simultaneously; if they are not, they happen at different times, unless a simultaneity sign ⊗ is in the margin.

What our process suggests is to start by performing elicitation so that a first set of such scenarios is produced.

3.2 Scenario Analysis

Producing ASCs is not sufficient for the analyst to get a precise understanding of the stakeholders' requirements. In particular, ASCs do not make explicit the underlying causalities existing (i) between action / interactions and reactions (i.e., other actions / interactions) and (ii) between action / interactions and states. This knowledge being of utmost importance for writing the final requirements, our approach suggests a way to elicit such information.

Asc	<ASC_id>	
Action occurrence	<action_occurrence_id>	
	ASC-specific	*Generalised*
Duration	What is the duration of the action occurrence in the ASC?	- Are there other circumstances in which the duration would be the same? - Are there circumstances in which the duration would be different? What would be the duration?
Action composition	In the circumstances given in the ASC, what are the relationships between this action occurrence and other action occurrences?	- Are there other circumstances in which these relationships would be the same? - Are there circumstances in which these relationships would not hold?
Precondition	What, in the circumstances given in the ASC, allows the action to take place?	- Are there other circumstances in which the action should be allowed? - Are there circumstances in which the action should not be allowed?
Triggerings	What, in the circumstances given in the ASC, forces the action to take place?	- Are there other circumstances in which the action should be forced? - Are there circumstances in which the action should not be forced?
Effect		
▪ **Internal pre-effect**	What is the effect of the beginning of the action occurrence on the agent instance that performs it?	- Are there other circumstances in which the beginning of the action would have the same effect? - Are there circumstances in which the beginning of the action would have a different effect? What would be the effect?
▪ **Internal post-effect**	What is the effect of the end of the action occurrence on the agent instance that performs it?	- Are there other circumstances in which the end of the action would have the same effect? - Are there circumstances in which the end of the action would have a different effect? What would be the effect?
▪ **Condition on post-effect**	What, in the circumstances given in the ASC, allows the end of the action occurrence to have an effect on the agent instance that performed it?	- Are there other circumstances in which the end of the action should have an effect on the agent instance that performed it? - Are there circumstances in which the end of the action should not have an effect on the agent instance that performed it?

Table 1 : Action occurrence form template.

During the Scenario Analysis phase, causality-related information is added to the initial ASCs using three new types of artefacts :

- ASC forms;

- action occurrence forms;
- interaction forms.

We decided to create these artefacts as separated from the ASCs' graphical representation in order to keep simplicity and communicability of the latter.

ASC forms basically detail the circumstances under which the ASC holds, that is, its precondition in terms of the states of the participating agents. In the example, the precondition is: the motor is off, the direction is idle, there are no requests from other users, all doors are closed, the lift is located three floors under the floor *f* at which the new request is going to be made and, finally, there are two other floors above *f*, namely *d* and *e*. For conciseness, the analyst can write parts of this information using Albert II as shown below – though, as the reader will see further, there remains an intellectual abstraction activity to be performed by the analyst to reach a specification.

$$
\begin{aligned}
&\mathit{Booth.Motor} = \mathit{Off} \wedge \mathit{Booth.MotorDirection} = \mathit{Idle} \\
&\wedge\ \forall\ \mathit{fl} : (\ \mathit{fl.UpButton} = \mathit{Idle} \wedge \ \mathit{fl.DownButton} = \mathit{Idle} \wedge \mathit{Booth.ButtonPanel}(\mathit{fl}) = \mathit{Idle} \wedge \ \mathit{fl.Door} = \mathit{Closed}) \\
&\wedge\ \mathit{Next}(\mathit{Booth.Position}, a) \\
&\wedge\ \mathit{Next}(a,b) \wedge \mathit{Next}(b,f) \wedge \mathit{Next}(f,d) \wedge \mathit{Next}(d,c)
\end{aligned}
$$

The other two types of forms have to be filled in by asking details about particular elements of the ASCs, namely action occurrences and interactions. The content of each form is structured into sections corresponding to types of causalities[h]. These types of causalities are the same as those that lead to the identification of Albert II specification templates[i]. For each section there are two slots to be filled in which differ in the abstraction level of their content: the first slot requires ASC-specific information while the second requires partially generalised information. Tables 1 and 2 show the form structure and the questions to be asked to elicit the requested information.

The answers put in the first slot define causalities that appear in the the concrete circumstances of a particular point in the ASC. That is, they assume that the precondition held initially in the ASC and that all the action occurrences and interactions previous to the point in the ASC have taken place, with their effects possibly changing the conditions on states wrt the ASC's precondition. From this information, the analyst has to get more general information in order to fill in the second slot. He has to check if the causality has a wider range of applicability than the given circumstances, what are the exact elements that are part of the causality and how the relationships between the elements changes if the circumstances vary. The type of elements than can have a role in a causality depends on the type of the causality. For example, a precondition defines a dependency of an action and its arguments on state component values (possibly, sequences of these values in time).

Table 3 shows a form fragment giving precondition information on the occurrence of *ResetUpButton* in the ASC of Figure 6. By asking whether such a precondition holds in other circumstances, the analyst discovers that it is the case for *f* being any floor where the booth is located (which is captured by the value of *Booth.Position*). This results in the partially generalised version of the statement (see second slot in Table 3).

---

[h] Some types of causalities are further detailed into more precise types of causalities. For example, when dealing with effects of actions, we distinguish between pre-effect, post-effect and condition on the post-effect. A more complex decomposition exists for action composition but it is not detailed here for sake of place.

[i] The reader interested in the rationale behind the identification of the various templates and therefore of the various underlying causality types can refer to [9].

ASC	<ASC_id>	
Action occurrence	<action_occurrence_id>	
Event	{beginning | end | occurrence}	
Originating agent instance	<agent_instance_id>	
Destination agent instance	<agent_instance_id>	
	ASC-specific	*Generalised*
External effect		
▪ **External pre-effect** (not applicable for end event)	What is the effect of the beginning of the action occurrence on the destination agent instance?	- Are there other circumstances in which the beginning of the action would have the same effect on the destination agent instance? - Are there circumstances in which the beginning of the action would have a different effect on the destination agent instance? What would be the effect?
▪ **External post-effect** (not applicable for beginning event)	What is the effect of the end of the action occurrence on the destination agent instance?	- Are there other circumstances in which the end of the action would have the same effect on the destination agent instance? - Are there circumstances in which the end of the action would have a different effect on the destination agent instance? What would be the effect?
▪ **Condition on external post-effect** (not applicable for beginning event)	What, in the circumstances given in the ASC, allows the end of the action occurrence to have an effect on the destination agent instance?	- Are there other circumstances in which the end of the action should have an effect on the destination agent instance? - Are there circumstances in which the end of the action should not have an effect on the destination agent instance?
Information :	What, in the circumstances given in the ASC, makes it possible for the interaction to be shown / to affect the destination agent instance?	- Are there other circumstances that would make it possible for the interaction to be shown / to affect the destination agent instance? - Are there circumstances that would make it impossible for the interaction to be shown / to affect the destination agent instance?
Perception :	What, in the circumstances given in the ASC, makes it possible for the destination agent instance to perceive / to be affected by the interaction?	- Are there other circumstances that would make it possible for the destination agent instance to perceive / to be affected by the interaction? - Are there circumstances that would make it impossible for the destination agent instance to perceive / to be affected by the interaction?

Table 2 : Interaction form template.

The main advantage of Scenario Analysis is to provide fine-grained information related to causalities between actions and other actions and between actions and state components. Such information improves the accuracy of the scenario-related information and is of major importance for writing the requirements specification. More generally, we can summarise the potential results of Scenario Analysis as follows :

- *Identification of relationships between ASC elements and static model elements.*
- *Identification of relationships between ASC elements.*
- *Discovery of new static model elements or refinement of previous models.* This is the case if, for example, being questioned on the effects of an action occurrence, the stakeholder mentions changes in state characteristics that did not yet appear in the static model. Therefore, new state components will have to be created. This is true for all causality types.
- *Creation of new ASCs.* Adding detailed causality information to an ASC element (especially expressing condition on states) often makes one think of what might happen in different situa-

tions (where some condition changes). If these situations need to be clarified, usually elicitation of new ASCs is triggered.

- *Creation of cross-references between ASCs.* This is usually a corollary of the previous advantage. Creation of cross-references might also appear a posteriori (during the scenario consolidation phase), as we will illustrate in the next section.
- *Discovery of real-time information.*

3.3 Scenario Consolidation

The goal of this activity is to take profit of the information resulting from Scenario Analysis in order to produce an abstract Albert II specification. To achieve this, the analyst has to complete the generalisation of the information contained in action occurrence and interaction forms. It is also at this stage that inconsistencies between contents of forms can be discovered.

In order to produce Albert II constraints for a given action, the analyst has to bring together the information contained in all the forms attached to all the occurrences of the action (and their associated interactions) in all the ASCs. For example, let us assume that *ResetUpButton* appears twice in the set of ASCs. Information pertaining to all types of causalities must be consolidated for this action. We continue to illustrate our approach with preconditions. The information that we want to aggregate is the partially generalised information contained in Tables 3 and 4.

Asc	MoveToHigherFloor_1	
Action occurrence	ResetUpButton	
	ASC-specific	*Generalised*
Precondition	f.Door = Closed ∧ Booth.MotorStatus = Off ∧ f.UpButton = Pressed ∧ ¬OutstandingRequestsUpstairs ∧ ¬OutstandingRequestsDownstairs	- Are there other circumstances in which the action should be allowed? Booth.Position.Door = Closed ∧ Booth.MotorStatus = Off ∧ Booth.Position.UpButton = Pressed ∧ ¬OutstandingRequestsUpstairs ∧ ¬OutstandingRequestsDownstairs - Are there circumstances in which the action should not be allowed? All other circumstances.

Table 3 : Action occurrence form fragment.

A first thing that the analyst has to notice is that the two conditions are mutually exclusive and that each of them ignores the existence of the other one by saying that in all other circumstances the action should not be allowed. Clearly, there is an inconsistency. After having checked with the stakeholders from whom the information was obtained, the inconsistency can be cleared by admitting that the situations represented by the two conditions all allow the action to take place (and only these situation allow it). So, first of all, in both forms, the negation of any other situation allowing the action to take place is replaced by a reference to the other form. Then, the conditions can be put together in order to provide the definitive precondition of action *ResetUpButton* (see specification in Section 2).

A similar process is followed for every action and for every causality type leading, possibly via backtracks to previous phases, to the final Albert II specification.

Asc	MoveToHigherFloor_2
Action occurrence	ResetUpButton

	ASC-specific	*Generalised*
Precondition	f.Door = Closed ∧ Booth.MotorStatus = Off ∧ f.UpButton = Pressed ∧ OutstandingRequestsUpstairs ∧ Booth.MotorDirection = Up	- Are there other circumstances in which the action should be allowed? Booth.Position.Door = Closed ∧ Booth.MotorStatus = Off ∧ Booth.Position.UpButton = Pressed ∧ OutstandingRequestsUpstairs ∧ Booth.MotorDirection = Up - Are there circumstances in which the action should not be allowed? All other circumstances.

Table 4 : Action occurrence form fragment.

4 The Albert II Requirements Specification Animator

Checking the adequacy of a formal requirements specification towards the informal needs expressed by stakeholders is far from being a trivial issue. To overcome this problem, we can roughly distinguish among three approaches:

- *Analysis techniques* allow the stakeholders to test if some properties (including goals [6]) hold from the requirements specification. These properties are checked by using model-checking [19] and/or theorem-proving techniques.

- *Conversion techniques* can be used to translate the specification into a representation which is more easily understandable by some stakeholders. The result can be some graphical diagrams that can be used for explaining the specification or a paraphrasing of the formal specification into a natural language text [14,34]. Conversion techniques can be considered as 'static' techniques since they do not provide specific help to the stakeholders in exploring the different possible behaviours that the specification associates to the composite system. It is only by reading carefully the proposed translation that they can imagine the possible behaviours.

- *Behavioural techniques* support much more interactivity with stakeholders by making them able to explore the different possible behaviours of the specified system. Two main approaches exist. *Prototyping* is based on the (manual or semi-automatic) translation of the specification into a program which can be executed. Several prototyping approaches [36,15,29] are based on the use of logic (e.g. Prolog) and functional programming languages. A basic problem with this approach is that, in order to be executable, the specification often has to be transformed and, thereby, concepts that are not relevant for the stakeholders have to be introduced (e.g., intermediate predicates during a Skolemisation process of transformation into Horn clauses) and others, which are of interest for them, become hidden in the resulting executable form. In other words, the consequence is that when the stakeholders discover an unexpected behaviour during the execution of the prototype, they cannot know the requirements which cause this behaviour. Another approach is *animation*. It allows the stakeholders to interact with a tool (called the animator) which makes it possible to incrementally build possible behaviours and check them with respect to the constraints of the requirements specifications. This approach really comes down to testing if a given scenario, proposed by one or several stakeholders, is compatible with the requirements specification. Examples of animators are those associated with Statecharts [16], Troll [13], SCR [18] or Lotos [20].

4.1 The proposed approach

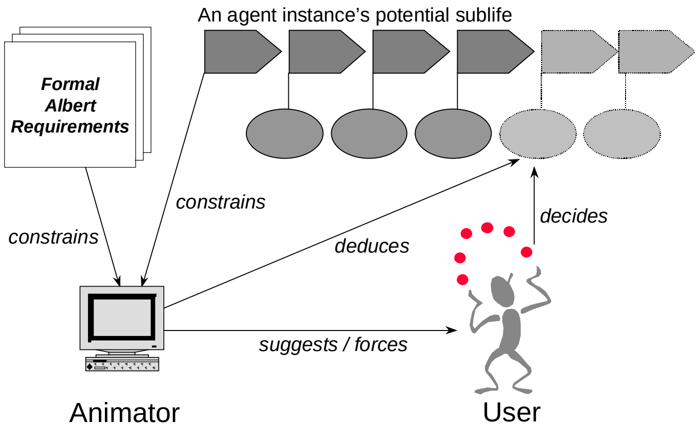

Figure 7 : Overview of animation.

For Albert II, our objective is to propose such an animator tool. However, a basic problem is that, despite some recent results related to the mapping of real-time temporal logic into automata [19,31,32], there is no existing automata technique proposing an expressiveness equivalent to the one existing in Albert II. Therefore, the approach that we have followed is to develop an interpretation algorithm taking the specification as an input and operationalising the semantic rules associated with Albert II. To justify our approach, we have to recall that the language provides templates on top of real-time first-order logic and that instantiations of a given template always map to logical statements with similar structures. As a consequence, the objective of dealing with full real-time first-order logic could be discarded. Our interpretation algorithm is made of a set of partially hard-coded[j] checks (checks for conditions to allow/refuse events proposed by the users,…) and triggers (requiring events) that take place at well-defined moments in the animation process. Computations are therefore also very efficient.

Figure 7 summarises the overall approach followed for the animation. We can see the central role played by the animator which permits to progressively build a possible system's behaviour respecting the constraints of the specification by interacting with the stakeholders. At any moment of the animation, the interpreter has to determine which events (beginning or end of actions) can take place in order to go from the current state to the next one. The identification of these candidate events is computed from the sub-behaviour already built and from the requirements specification. Some of these events have to take place because the specification's constraints force them to occur. Other events can take place or not because the specification leaves their occurrence undetermined. In such cases, the interpreter interacts with the stakeholders who decide which events have to occur. For the sake of space, we just give an example of the operationalisation performed by the animator.

Operationalising *action duration* constraints is done by the animator in the following way. When the beginning of an action is selected by a user, by examining the constraints, the animator computes

[j] We say that the checks are partially hard-coded because the temporal parts of the check are dealt with in an ad hoc way for each constraint template. On the other hand, we implemented a generic expression evaluator for the non-temporal parts.

the closest and the latest time at which the end of the action has to take place. This information is put into the animation's *obligations*. Obligations are constraints that are passed by each step of the animation to the next one until they are satisfied. The user will thus not be allowed to terminate the action occurrence if the value of the closest time in the obligations is greater than the current time. And, vice versa, he will not be allowed to continue the animation if the current time is equal to the latest occurrence time in the obligations.

Finally, we have to insist on the fact that, since the declarative properties of the specification are translated (amongst other things) into checks that are made at definite moments of the animation, there could be cases in which, at a given time during the animation process, the agent instances' sub-lives we have constructed up to the current timepoint are not admissible sub-lives wrt the constraints expressed in the specification. For example, if we have an action composition constraint imposing that each occurrence of the action *SwitchOn* of some agent is followed, within 2 minutes, by an occurrence of the *SwitchOff* action and if, at the current time in the animation, an occurrence of *SwitchOn* takes places, we will only be able to conclude, 2 minutes later, that the *SwitchOff* action can or cannot actually take place by looking at its preconditions at that moment. The reason for this is that, since non-determism is largely used in Albert II specifications, future contexts cannot be determined as they require choices to be made by the users of the animation.

4.2 The architecture of the Albert II animator

The global architecture of the tool is represented in Figure 8. At the moment, we have a distributed prototype which allows different stakeholders to cooperatively animate a specification, each of them being responsible for the animation of a part of the system/environment he is interested in.

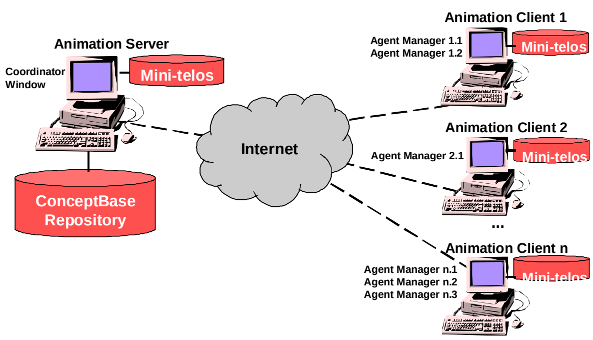

Figure 8 : Architecture of the animator.

The specifications used in the animator are produced with the support of the Albert II Editor tool (not illustrated in the figure) and are stored in the ConceptBase repository [25] using the Telos language [28]. The information in the repository is augmented with additional animation-specific information that is used throughout the animator. The server is assigned to the execution of the interpretation algorithm and is used by the *coordinator* of the animation, i.e., the person who is in charge of controlling the global flow of the animation. Lower-level operations related to the control of agent instances are performed by users of the client applications. More precisely, they are done within a*gent managers* (see next sub-section) which are windows that are created on the machines where clients are located for each agent instance of the animation. The use of the animation client itself is just to allow the users to connect to the animation server.

Animation clients only need information about agents managed locally and therefore only deal with views of the global ConceptBase repository which is managed by the server. Information exchanged among the clients and the servers are in a so-called MiniTelos (CDIF-like) format. MiniTelos is a light implementation of Telos (no deductive capabilities) that we have implemented in our university. It consists of a library of reusable Java classes. Its internal representation is made of serialisable Java objects that can be either written to a file or can be converted by MiniTelos into an ASCII file containing Telos frames that can then be used by ConceptBase.

4.3 The functionalities offered by the animator

As described above, the animation is distributed in the sense that different stakeholders animate different agent instances of the specification. For example, in the case of our lift case study, we can imagine the following configuration:

- *Stakeholder1* plays the role of the *Controller;*
- *Stakeholder2* plays the role of the *Booth;*
- *Stakeholder3* plays the role of the *TopFloor*, the *GroundFloor* and and *Floor1*, *Floor2* and *Floor3* which are instances of *IntermediateFloor*. This means that, for the purpose of the animation, we consider a specific configuration with 5 floors. This number was not fixed at the specification level (see Section 2) but needs to be fixed at the animation level because we are now working at the instances' level;
- *Stakeholder4* … *Stakeholder7* play the role of *Users*. A specific instance is called *Eric*.

As an example, let us consider the agent manager window associated with *Stakeholder3*'s client and which is in charge of controlling *Floor3* (among other things) : the first folder in the window shows the state of the agent instance while the second folder supports the building of a state transition.

Figure 9 shows the state folder of the window at some stage i of the animation. We can see the three main state components associated with any floor, viz the *UpButton*, the *DownButton* and the *Door* (in our example, there is no imported component). For each of these, we have access to its value. In the figure, we can see that, at the current stage, the value of *Door* is *Closed*. Consulting the values of the other components (not depicted in the figure) gives the values provided in the first column of Table 5.

	Stage i	*Stage i+1*
UpButton	On	Off
DownButton	Off	Off
Door	Closed	Closed

Table 5 : Values of state components of Floor3 during animation.

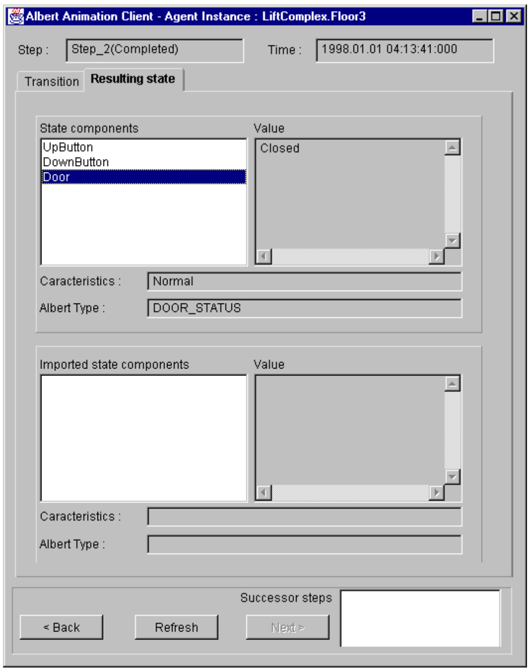

Figure 9 : The agent manager : state folder.

Let us now consider the values of the same components at next stage (i+1) of the animation (see second column of Table 5. The stakeholder has discovered a surprising behaviour where the *UpButton* has been reset to *Off* while the *Door* remains *Closed*. To avoid interrupting the animation anytime an unexpected behaviour happens; it is always possible for the stakeholder to attach a remark to any component of a state or state transition built or being built. This will then be reused when analysing the traces of the animation. In this case, the conclusion of the analysis of such traces will be that constraints have to be added in order to synchronise the occurrences of *ResetUpButton* with the occurrences of *RequestOpenDoor*. The specification fragment provided in Section 2 has already been corrected to enforce this behaviour. This is done through the *action composition* constraint[k].

Figure 10 shows the state transition folder of the window associated to the agent manager of *Eric* (the instance of *User* animated by *Stakeholder4*). It is used to construct a state transition as follows. The list of actions that have been declared for the agent *User* are displayed as a list on the window. Actions can be selected by the stakeholder and, by clicking on the "Start", "End" or "Perform" button, one can decide to add to the current transition an event representing respectively the beginning of an action, its end or both events (i.e., an instantaneous action occurrence). If the event includes the beginning of an action having parameters, a dialogue box appears for entering them. The stakeholder can decide to add any event but will receive an error message if the proposed action occurrence violates statements of the Albert II specification. In order to facilitate revisions, traceability within the animator allows the error messages to refer directly to the violated statements. For example, if we consider that we are at the stage i described above, only the *PressDownButton* and the *LeaveFloor* actions are possible because:

- there is a precondition preventing the triggering of *PressUpButton* if the button is already pressed;
- there is a precondition preventing entering the *Booth* if the *Door* is closed;
- there is a precondition making it impossible to exit the *Booth* or to press a button on the panel inside it if the *User* is not in the *Booth*.

[k] The *precondition* and *triggering* (not shown) constraints of *ResetUpButton*, which specialise the corresponding constraints of *RequestOpenDoor*, are also needed.

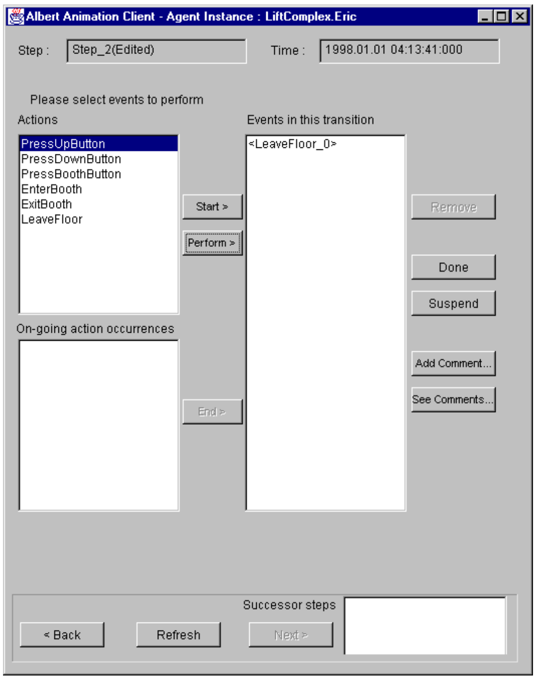

Figure 10 : The agent manager : state transition folder.

The stakeholder is referred to these statements if he tries to activate the corresponding actions.

Not shown in Figure 10, are the possibilities to reflect Albert II cooperation constraints which let the stakeholder choose (i) to show or not an action to other agent instances when no (sufficient) condition is present in the specification and (ii) to perceive or not the events that are shown by the other agent instances when no (sufficient) condition is present in the specification. In our example, nothing special has to be written since, in the requirements specification, it is said that there is a perfect and deterministic communication between the agents.

The central role played by the animator server (which supports the task of the coordinator) is to conduct the animation by centralising all the individual state transition requests issued by the different stakeholders and building the new global state of the system (made of all the local instances' states). Note that this resulting state may not be an admissible state with respect to the specification. This results in a dead-end branch which cannot be further investigated. Such states are especially marked both in the different agent manager windows and on the coordinator's display. The other possible origin of a dead-end branch is the fact that some stakeholder has decided to stop building the current state transition either because he was prevented from performing the events he intended to or because some events he performed in previous stages now have unforeseen consequences. When one stakeholder pushes on the "Done" button, all the other stakeholders can do is stop too.

Four other main functionalities are:

- When starting an animation, the coordinator is given the choice either to open an existing animation from the repository or to create a new one.
- When starting a new animation, the coordinator should fix the agent instances' configuration (see beginning of this sub-section) and distribute it among the different stakeholders. Then, the coordinator should prompt the different stakeholders to enter the values of the state components for the initial state and check that the values are valid with respect to the *Initial Valuation* constraints given in the specification (see Section 2).
- At any moment of the animation, backtracking facilities are supported in order to explore alternative (more or less normative, more or less exceptional) scenarios.

- After an animation has taken place, it is possible to consult (textual) traces of the animation. This is intended to allow further validation by stakeholders who did not directly take part in the animation. Since user comments and problems found by the animator can be included in the traces, these can serve as a basis for change requests transmitted to the analyst in order to improve the specification.

Above, we have illustrated the main functionalities offered by the animator. There are others that are not detailed in the paper for lack of space. However, the interested reader can find more information in [21].

5 Discussions

In this paper, we have reported on the use of scenarios within the context of the elaboration and validation of requirements expressed in terms of the Albert II language, a formal language designed for complex real-time systems.

At the level of the elaboration of such requirements, we have introduced an MSC-like notation (called ASCs) for capturing preliminary scenarios expressed by stakeholders for the desired system. At the RE level, scenarios have to focus on the actions being performed by the different agents and on the interactions taking place among agents. The work reported in this paper has been guided by our belief that, in a general RE context, scenarios (and in particular those expressed with MSC-like notations) are most suited for elicitation (because of their inherent communicability) than for specification. This is indicated by the fact that the addition of constructs to MSCs proposed in the literature in order to make them more abstract (that is, make them able to describe larger numbers of instance-level behaviours) tend to make MSCs complex (see [1,2]). With respect to other methods that make use of scenarios for the elicitation of formal specifications [22,12,37], we claim for a greater expressiveness and naturalness of the resulting specification even if we are not able to build it in a fully-automated way.

Not discussed in the body of the paper is the issue of extracting scenarios and structuring them into some scenario data-base. For extracting scenarios, we rely on complementary techniques elaborated within the CREWS project, like goal-based authoring [3] or elicitation from real-world scenes [17]. Results from the project also show the usefulness of validating the scenarios before they are consolidated. In this respect, tools and techniques presented in [26] help in checking and extending the coverage of exceptional situations by scenarios. Method and tool integration of the various CREWS techniques is currently in process. For structuring scenarios, which is critical for dealing with real-size problems, [33,1] provide interesting mechanisms like *episodes* (see also [30]), *services* and the well-known *use cases* (introduced by Jacobson [24]). We are currently investigating adapting them to structure our ASCs.

Other pending issues we are considering related to ASCs are as follows:

- Notations used in ASCs have been defined to be close to the Albert II concepts (actions, events, etc.). However, at the moment, no semantics is provided for ASCs. In the future, we expect to provide one with the benefit of using scenarios and the associated contextual information (like the global precondition) for formally checking that they are admissible behaviours from the Albert II requirements. Along these lines, current work is done on the definition of adapted model-checking techniques for the Albert II language [31,32].
- At the moment, traces of animation are just textual. Another study therefore concerns the obvious relationship existing between these traces and their graphical representations through ASCs. This visualisation of traces would make it easier to further interact with additional stakeholders not involved in the animation and to serve as documentation (and test cases) for the future implementation of the system.

At the level of the requirements validation, the animator allows cooperative work between stakeholders and the exploration of alternative behaviours by fixing non-deterministic events interactively. Furthermore, another feature of the tool is to keep the dialogue at the level of the vocabulary used in the specification. Future work is expected to take on the followings issues :

- For the moment, we have built a first prototype of the animator. However, the interpretation algorithm underlying its use has still to be extended in order to take a full account of all the possibilities offered by the Albert II language, like complex *state behaviour* constraints.
- Another issue is concerned with the guidance provided to the stakeholders when exploring different behaviours. Domain-dependent 'meta-knowledge' could be added in order to guide the animation by presenting normative as well as exceptional (less normative) behaviours. A classification of such 'meta-knowledge' is provided in [26].
- A final issue is related to improving the user interface of the animator for making it more intuitive. A first problem is visualisation of sequences of events and values. Again, ASCs could be useful. For sequences of state values, tables such as the one presented in Table 5 would improve presentation. A second problem is that of building a domain-specific graphical layer on top of some agent manager windows in order to keep the use of the animator even closer to users' concepts.

Evaluation of the tool is currently under way on two industrial applications. These applications have already given us some directions and priorities on how to extend the tool (especially regarding (i) additional constraint templates to be dealt with by the algorithm and (ii) the user interface). Further publications are planned to detail these results.

Finally, another issue we want to address in how our elaboration and validation techniques can complement each other. In this respect, it is clear that existing ASCs can be replayed in the animator for checking if the specification accepts them. But, more interestingly, combinations not yet apparent in the current set of ASCs but produced by generalisation can be validated. This would provide an a posteriori verification of the abstraction done by the analyst.

6 Acknowledgements

The work reported in this paper is supported by the ESPRIT Long Term Research project 21.903 CREWS (*Cooperative Requirements Engineering With Scenarios*) and by the Wallon Region's CAT project. The authors wish to thank the members of the CREWS project for fruitful discussions on the topics of scenarios and animation. They are also grateful to the present and former members of the Albert II team for collaboration and seminal work on formal specifications and animation. Finally, the authors' gratitude also goes to Pierre-Yves Schobbens for his careful proof-reading and to Christophe Bongartz for his preliminary work on linking Albert II descriptions with scenarios.

7 References

[1] Andersson M, Bergstrand J. *Formalizing Use Cases with Message Sequence Charts*. Tech. Rep. Dept. of Communication Systems. Lund University. Lund (Sweden). 1995.

[2] Ben-Abdallah H, Leue S. *Expressing and analyzing timing constraints in Message Sequence Charts specifications*. Tech. Rep. 97-04. Electrical and Computer Engineering. University of Waterloo. Ontario. Canada. April 1997.

[3] Ben Achour C. *Writing and Correcting Textual Scenarios for System Design*. In: Proceedings of the Natural Language and Information Systems (NLIS'98) Workshop. 28th August 1998. Vienna. Austria. 1998.

[4] Benner KM, Feather S, Johnson WL, Zorman LA. *Utilizing scenarios in the software development process*. In: Proc. of IFIP WG 8.1 Working Conference on Information Systems Development Process. December 1992.

[5] Chabot F, Raskin JF, Schobbens PY. *The Formal Semantic of Albert II*. Technical report. Computer Science Department. University of Namur. Namur (Belgium). May 1998.

[6] Dardenne A, Fickas S, van Lamsweerde A. *Goal-directed concept acquisition in requirements elicitation*. In: Proc. of the 6th International Workshop on Software Specification and Design - IWSSD'91. Milano (Italy). October 1991.

[7] Dubois E, Petit M. *Using a formal declarative language for specifying requirements modelled in CIMOSA*. In: Proc. of the European workshop on Integrated Manufacturing Systems Engineering (IMSE'94). François Vernadat (ed). Grenoble (France). December 12-14. 1994. INRIA Rhône-Alpes. pp. 233 -241.

[8] Du Bois P, Dubois E, Zeippen JM. *On the use of a formal RE language : the generalized railroad crossing problem*. In : Proc. of the IEEE International Symposium on Requirements Engineering (RE'97). Annapolis MD. January 6-10. 1997. pp. 128—137. IEEE Computer Society Press.

[9] Du Bois P. *The Albert II reference manual : language constructs and informal semantics*. Research Report RR-97-002. Computer Science Department. University of Namur. Namur (Belgium). July 1997. Available at ftp://ftp.info.fundp.ac.be/publications/RR/RR-97-002.ps.Z.

[10] Dubois E. *Albert : A formal language and its supporting tools for requirements engineering*. In: Formal Aspecs of Software Engineering (ETACS'98 Conference). LNCS. March 1998.

[11] Feather MS. *Language support for the specification and development of composite systems*. ACM Transactions on Programming Languages and Systems. vol. 9. no. 2. April 1987. pp. 198-234.

[12] Glinz M. *An integrated formal model of scenarios based on statecharts*. In: Proc. of ESEC '95 - 5th European Software Engineering Conference. Sitges (Spain). 1995. pp. 254—271. Springer.

[13] Grau A, Kowsari M. *A Validation System for Object-Oriented Specifications of Information Systems*. In : Manthey R, Wolfengagen V. (eds). Advances in Databases and Information Systems. Proceedings of the First East-European Symposium on Advances in Databases and Information Systems (ADBIS'97). St. Petersburg. September 2-5. 1997. Electronic Workshops in Computing. Springer.

[14] Gulla JA. *Explanation Generation in Information Systems Engineering.* PhD thesis. Trondheim. October 1993.

[15] Habra N. *A Transformational Method for Functional Prototyping.* PhD thesis. Computer Science Department. University of Namur. Namur (Belgium). September 1990.

[16] Harel D, Lachover H, Naamad A, et al. *STATEMATE: A working environment for the development of complex reactive systems*. vol. 16. pp. 403—414. April 1990.

[17] Haumer P, Pohl K, Weidenhaupt K. *Abstraction Guides: Interrelating Conceptual Models with Real World Scenes*. In: Fourth International Workshop on Requirements Engineering: Foundation for Software Quality (RESFQ). Pisa. Italy. June 8th -9th. 1998.

[18] Heitmeyer C, Labaw B, Kiskis D. *Consistency checking of SCR-style requirements specifications*. In: Second IEEE International Symposium on Requirements Engineering. March 1995. IEEE CS Press.

[19] Henzinger TA. Nicollin X. Sifakis J. Yovine S. *Symbolic model checking for real-time systems*. Information and Computation. no. 111. pp. 193-244. 1994.

[20] Hernalsteen C, de Jacquier A, Massart Th. *A toolset for the analysis of ET-LOTOS specifications*. In: Proc. of Meeting on Validation and Verification of Formal Descriptions of the Fundamental Computer Science (FNRS) Contact Group. Namur (Belgium). May 1997.

[21] Heymans P. *The ALBERT II specification animator*. Tech. Rep. CREWS report 97-13. University of Namur. 1997. Available at http://Sunsite.Informatik.RWTH-Aachen.DE/CREWS/.

[22] Hsia P. Samuel J. Gao J. Kund D. Toyoshima Y. Chen C. *Formal approach to scenario analysis*. IEEE Software. March 1994.

[23] Jackson M. *Software Requirements and Specifications: A Lexicon of Practice, Principles and Prejudices*. Addison Wesley 1995.

[24] Jacobson, Christerson, Jonsson, Overgaard. *Object-Oriented Software Engineering - A Use Case-Driven Approach*. Addison Wesley 1992.

[25] Jarke M, Gallersdorfer R, Jeusfeld MA, Staudt M, Eherer S. *ConceptBase - a deductive object base for meta data management*. Journal of Intelligent Information Systems. vol. 4. no. 2. pp. 167-192. 1995.

[26] Maiden NAM, Minocha S, Manning K, Ryan M. *SAVRE: Systematic Scenario Generation and Use.* In: International Requirements Engineering Conference (ICRE'98). Colorado Springs. Colorado. USA. April 6-10. 1998.

[27] Meyer B. *On formalism in specifications*. IEEE Software. vol. 2. no. 1. 1985. pp. 6-26.

[28] Mylopoulos J. Borgida A. Jarke M. Koubarakis M. *Telos: A language for representing knowledge about information Systems*. ACM Transactions on Information Systems. vol. 8. no. 4. pp. 325—362. October 1990.

[29] O'Neill G. *Automatic translation of VDM into standard ML programs*. The Computer Journal. March 1992.

[30] Potts C. Takahashi K. Anton A. *Inquiry-based requirements analysis*. IEEE Software. March 1994.

[31] Raskin JF, Schobbens PY. *State clock logic: A decidable real-time logic*. In: Proc. of HART'97 : Hybrid And Real-Time Systems. Grenoble. March 26-28 1997. pp. 31—47. Springer-Verlag.

[32] Raskin JF, Schobbens PY. *Real-time logics: Fictitious clock as an abstraction of dense time*. In: Proc. of TACAS'97 : Tools and Algorithms for the Construction and Analysis of Systems. Twente. April 2-4 1997. pp. 165—182. Springer-Verlag.

[33] Regnell B, Andersson M, Bergstrand J. *A hierarchical use case model with graphical representation*. In: Proc. of ECBS'96, IEEE International Symposium and Workshop on Engineering of Computer-Based Systems. IEEE. March 1996.

[34] Rolland C, Proix C. *A natural language approach for requirements engineering*. In: Proc. of the 4th Conference on Advanced Information Systems Engineering - CAiSE'92. P. Loucopoulos (ed). Manchester (UK). May 12-15. 1992. pp. 257—277. LNCS 593. Springer-Verlag.

[35] Rolland C. Ben Achour C. Cauvet C. et al. *A Proposal for a Scenario Classification Framework*. In: Requirements Engineering Journal. vol 3. no. 1. Loucopoulos P, Potts C (eds). Springer Verlag. 1998.

[36] Siddiqi JI. Morrey IC. Roast CR. Ozcan MB. *Towards quality requirements via animated formal specifications.* Annals of Software Engineering. vol. 3. 1997.

[37] Somé S. Dssouli R. Vaucher J. *Toward an automation of requirements engineering using scenarios*. Journal of Computing and Information. vol. 2. no. 1. pp. 1110—1132. 1996.

[38] Various authors. *2RARE (2 Real Applications for Requirements Engineering): Final report on user's results*. Available at http://www.info.fundp.ac.be/~phe/2rare.html. August 1996. Esprit Contract Number 20424.

[39] Various Authors. IEEE Recommended Practice for Software Requirements Specifications. Std 830-1993. IEEE. 1993.

[40] Various Authors. *Proceedings of the 4th International Workshop on Software Specification and Design - IWSSD'87.* Monterey CA. April 3-4 1987. Computer Society Press.

[41] Various authors. *ITU-T recommendation z.120 : Message sequence charts (MSC).* Telecommunication Standardization Sector of International Telecommunication Union. Geneva. Switzerland. 1996.

[42] Wieringa RJ. *Requirements Engineering: Frameworks for Understanding*. Wiley 1996.

[43] Wieringa RJ. Dubois E. *Integrating semi-formal and formal software specification techniques*. Information Systems Journal. June 1998.

[44] Zave P. Jackson M. *Four dark corners of requirements engineering*. ACM Transactions on Software Engineering and Methodology. vol.6. no. 1. 1997. pp. 1-30.