CREWS Report Series 97-04

# Some thoughts about the animation of formal specifications written in the AlbertII language

Patrick Heymans
Computer Science Department, University of Namur, Belgium.

E-mail : phe@info.fundp.ac.be

appeared in Proceedings of the Doctoral Consortium
of the third IEEE International Symposium on
Requirements Engineering (RE'97),
Annapolis, MD, USA, January 6-10, 1997

# Some thoughts about the animation of formal specifications written in the AlbertII language*

**Author** : Patrick Heymans
Computer Science Department, University of Namur,Belgium.
E-mail : phe@info.fundp.ac.be

**Supervisor** : Eric Dubois
Computer Science Department, University of Namur,Belgium.
E-mail : edu@info.fundp.ac.be

*This work is partly funded by ESPRIT LTR project 20.903 (CREWS - Cooperative Requirements Engineering with Scenarios)

## Abstract

Formally and declaratively specifying requirements on real-time composite systems requires validation by stakeholders. One way to perform such validation is allowing people to experience the dynamic properties of the system to be built by using an animation tool. In this paper, we give an overview of the issues raised by the development of such a tool for the AlbertII language with the main goal of obtaining feedback on research that is a its very early stages.

## 1 Introduction

The validation task which takes place during the Requirements Engineering (RE) of software-based systems certainly cannot be considered a solved problem yet. The most important reason for this is that the analysts in charge of writing the requirements specification and the stakeholders from whom the necessary information has to be elicited 'do not speak the same language': while the former still often make use of software-oriented modelling techniques supporting an operational style of specification, the latter would like to have a declarative statement specifying **which** of their needs will be taken into account rather than the description of **how** some of them (which ones?) are solved. Besides this need for **declarativeness** of specification languages, there is also a need for **expressiveness**, especially when people have to deal with composite systems [1] and want to express unambiguously constraints involving such things as real-time aspects and concurrent behaviours.

Orthogonally to its declarative and expressive aspects, it may be chosen to give a specification language a formal semantics i.e. an unambiguous interpretation in terms of a mathematical structure associated to the specification by the means of well-defined rules. Major advantages of **formal languages** are the possibility to make use of (semi-)automated techniques allowing to reason about the specification or to derive a prototype from it. The major drawback of such languages is the notation which is generally borrowed from mathematics and logics and therefore not often understood by anyone apart from the analyst. A rough classification of the various techniques that

[1] We call composite system a system made of heterogeneous components ranging from software and/or hardware components up to human and/or device components

can be used in support to the validation of formal requirements specifications is as follows:

**Conversion techniques** are used to convert the specification into a form which is more easily understandable by all the stakeholders (e.g. graphical representation, paraphrasing). Semantic equivalence with the original specification is anyhow hard to obtain in a fully-automated way.

**Behavioural techniques** are, in our opinion, all those which permit, in a broad sense, to observe, experience and test the dynamic properties of a specification. It involves (possibly symbolic) execution of a prototype which is, according to the level of executability of the language, the specification itself or a program derived using a transformational approach. Simulation is also part of these techniques.

**Analysis techniques** apply either to the specification (like model-checking and theorem proving) or to results delivered by the use of some behavioural techniques like the study of execution traces.

The AlbertII language [7] has been designed with both the aforementioned declarativeness and expressiveness properties in mind in order to address the problem of identifying and specifying the various components of a system. Each of these components either belongs to the already present environment (domain knowledge) or the 'machine' [9] to be installed. The other major concern when designing the language has been a **methodological** one which resulted in identifying a set of typical constraints patterns that the analyst has to fill in in order to write the specification. AlbertII is a formal language whose patterns are written on top of a real-time temporal logic for which reasoning techniques are currently under study.

The approach we intend to take for validating AlbertII specifications is actually a new one that we classify among the behavioural techniques and that we call **animation**. AlbertII is able to express **undeterminism**, which is very useful in the early stages of system development but is of little help when the goal is to build a prototype from a specification since executability is not present, as shown in [5]. Animation consists for a user or a community of users to dynamically build a behaviour made of the behaviours of the various components of the systems and its environment. This behaviour will be progressively created by interacting with a tool (called **animator**) which will check if the behaviour respects the constraints of the specification as it is introduced in the tool. Whenever undeterminism is encountered, the system asks the user to make a decision in order to remove it. The purpose of this approach really comes down to test if a given **scenario** [1, 8, 11], proposed by one or several stakeholders, is compatible with the requirements specification. Therefore, a link clearly exists between our research and the research which is currently being done around the use of scenarios in RE. Participation in the ESPRIT LTR project CREWS (**Cooperative RE with Scenarios**) [4] will help in developing this link.

In section 2, we first give an overview of the language with the help of a simple example and then try to give an intuition of what is the underlying formal semantics. Section 3 sketches the basic functioning of the animator. The major contributions we intend to bring to the tool are explained in section 4. The paper finally concludes on what has been accomplished so far and what is still left to do.

# 2 Overview of the AlbertII language

## 2.1 Syntax

The example that will be used throughout this section is a very simple one situated in the context of a library (the entire specification can be found in the Appendix of this document). The first thing to say about an AlbertII specification is that it is made of two main parts : (1) the 'static' part made of **type** and **operation** definitions and (2) the 'dynamic' part related to the specifications of **agents**. (1) is written at the beginning of the specification because it defines types and operations which will be needed at a variety of places afterwards. In the example, the only thing they contain is the definition of *BOOK* as an enumerated constructed type. (2) generally represents the main part of the specification and is explained just below.

Agents correspond to entities of the real world which appear to have some 'autonomy' (e.g. a person, a subsystem, an external machinery). The decomposition of a specification into several agent specifications allows to reduce the interdependencies between its various parts. The agents of the specification are, on the one hand, the librarian and, on the other hand, a community of users (see fig. 1). Agents are grouped into **societies**. In our example there is only one society grouping the two agent classes and that we call *Main*. The specification of an agent is itself decomposed into a **declaration** part and a part which consists of **constraints** aimed at pruning the number of admissible behaviours (called **lives** in AlbertII ) an agent can have.

The declaration part of an agent specification makes use of two main notions, the one of **state component** and the one of **action**, and has a graphical counterpart shown in fig. 1.

Agents are usually responsible for some 'data' they maintain. In this case, the librarian has to take care about the books which are on the shelf and therefore possesses a **state component** called *OnShelf* which is defined as a set of elements of type *BOOK*. Similarly, every user has a set of books (s)he has borrowed. This is represented by the a state component of the *User* agent class which is called *Borrowed* and which is defined in the same way as *OnShelf*. Agents are also able to perform actions and actions can have parameters like the action *Request* which is performed by a *User* and which has a parameter of type *BOOK*. Both actions and state components can be **exported** by their owner agent to one or several other agent(s). The syntax for declaring state components and actions is illustrated by the commented specification excerpt below (where the arrow means that a state component or an action is exported and is followed by the agent(s) to which it is exported) :

$\boxed{\textbf{User}}$

DECLARATIONS

STATE COMPONENTS

*Borrowed* <u>set-of</u> *BOOK*

`A user can possess a set of books that he`
`has borrowed`

ACTIONS

*Request(BOOK)* $\longrightarrow$ *LIBRARIAN*

`A user can request a book from the librarian`

As already mentioned, the rest of the specification of an agent is made of a series of constraints restricting the set of possible lives of the agent. There are several types of constraints and each type has an associated syntactic pattern which serves as methodological guideline for the specificier who wants to express something. The types of constraints are grouped into three families of constraints : (1) **basic** constraints, (2) **local** constraints and (3) **cooperation** constraints (see Appendix). Basic constraints are used to describe the initial state of an agent (**initial valuation** constraints) and to give the derivation rules for the derived state components (**derived components** constraints). There are five types of local constraints which relate to the internal behaviour of an agent : **state behaviour**, **effects of actions**, **capability**, **action composition** and **action duration** constraints. Finally, cooperation constraints specify how an agent interacts with its environment, i.e. how it lets the other agents know what actions it performs (**action information**), how it shows parts of its state to other agents (**state information**), how it perceives actions from other agents (**action perception**) and how it can see parts of the state of other agents

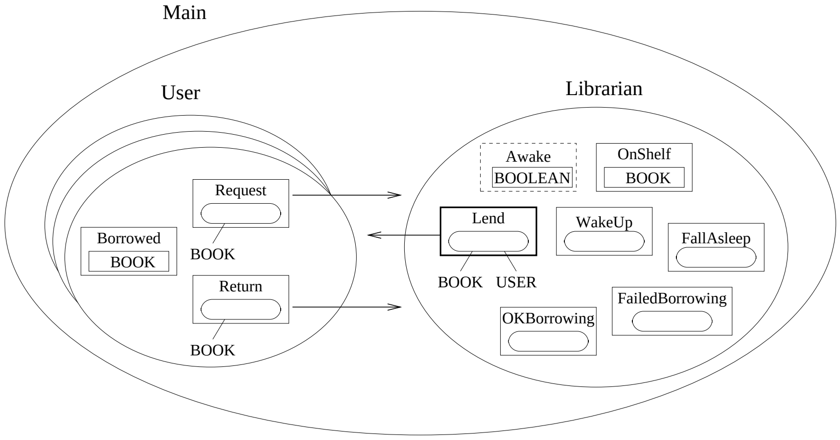


Figure 1: The library : agent declarations

(**state perception**). We will not go in the details of each type of constraint (which is provided in [7]) but rather give some examples chosen from the specification of the Appendix. The state behaviour constraint which says that users never keep books for more than a month shows how it is possible to express constraints on states in a very declarative style :

State Behaviour

b ∈ Borrowed $\Longrightarrow \Diamond_{\leq 1 month}$ ¬ b ∈ Borrowed

An operational style can also be used when necessary like in the effect of the action *Return* :

Effects of Actions

Return(b): Borrowed := Remove(Borrowed,b)

Finally, here are two examples of constraints that put restrictions on actions. The first one restricts the set of possible states from which the *Return* action can take place while the second gives the condition that must be satisfied in order for the action to be shown to the librarian (in this case there is no restriction):

Capability

$\mathcal{F}$ ( Return(b) / ¬ b ∈ Borrowed )

`A user cannot return a book he has not borrowed`

Action Information

$\mathcal{K}$ ( Return(-).l / TRUE )

`The fact that a user returns a book is`
`always shown to the librarian`

## 2.2 Semantics

The semantics of an AlbertII specification is given by mapping it to a real-time temporal logic called Albert-CORE. The set of axioms which result from the translation of an AlbertII specificication into Albert-CORE defines a set of **models** [2] of the specification. Each of these models is made of the 'sum' of the lives of the instances of agents it contains.

The life of an agent is an alternate sequence of **states** and **changes**. A state of an agent instance represents the values of all its state components during a time interval. A change groups together all the **events** that affect an agent instance at a given point in time. An event is either :

1. an instantaneous action occurrence,

[2]The term *model* is used here in the sense of a mathematical interpretation structure associated with a logical theory.

2. the beginning of an action occurrence,
3. or the end of an action occurrence.

Events are perceived by a given agent instance either because the agent instance is the one that performed it or because it had 'imported' the event from another agent instance. Since the set of axioms produced from a specification contains a **frame axiom** [2], events are the only way by which states can be modified which implies that between two different states, there is always a change.

Finally, an admissible life associated to the specification is built by combining the lives of several agent instances, i.e. putting them to a common time line by adding states (if needed) and checking compatibility wrt. cooperation constraints. Such a life (that we call **global life**) therefore defines an admissible behaviour of a composite system.

# 3 Basic functioning of the tool

A prototype was made a few years ago for the first version of AlbertII [6]. The first version of the language was much poorer than the current one both from the syntactic and semantic points of view. Anyway, the prototype laid the bases of the the functioning of the tool as they are explained below.

The interface of the animator will be window-based. A main window, besides providing menus and buttons to activate global functionalities, will display a global clock. During the animation, a window will be attached to each agent instance described in the **context** (see below) of the animator. This window will show the current value of the state components and will allow the user both to build the agent life step by step (building mode) and to consult information on its past (reviewing mode). A window should also be present to let the user observe the specification itself if he wants to have a justification of the decision made by the animator.

A session with the animator will typically be divided into several steps :

1. A **preliminary step** will be required in order to :
   (a) instantiate the agent classes [3],
   (b) initialize every agent instance's state (of course, the animator checks if state behaviour and initial valuation constraints are met),
   (c) set the value of time for the beginning of the animation.

   The above enumeration of actions aims at creating the so-called context of an animation.

2. Then, the **animation** itself can start.
   (a) In the the building mode, the user has to choose which events take place in order to go from the current state to the next one and how much time after the beginning of the current state these occur. Some events are chosen anyway by the animator if the specification says to do so. Relating this to the concepts we have presented in the section devoted to semantics of the language, it comes down to building the change which takes place between two states of an agent instance's life. The values of the components in the next state are automatically calculated by the animator.
   (b) The reviewing mode just consists in passively exploring parts of the life which has already been built.

# 4 Expected contributions

In the present section we tend to clarify our goals and, since our work has only just started, we can only give an idea of how we intend to achieve these goals in our work :

**Formal reasoning.** The animator will have to use as much as possible of the formal reasoning

[3]The agent population being static in AlbertII, instantiation can take place before the animation itself

capabilites offered by the logic underlying Albert II. This point will be the major improvement wrt. the previous prototype which had mainly an intuitive approach to reasoning. More rigourous computations performed on Albert-CORE formulae will be useful (1) for checking whether the whole behaviour matches the specification or (2) for performing more 'local' checks and computations. (2) involve such things as checking if the initial state is a reachable one, calculating which actions have to/can/cannot take place at given moment and computing the next state. (1) will be reformulated into checking that $M_B \subseteq M_S$ provided that $M_S = \{m \mid m \models S\}$ is the set of models allowed by the specification S and $M_B$ is the set of models the behaviour B can be part of.

**Symbolic animation.** In order to treat scenarios at the level of abstraction which is relevant for users, we plan to make symbolic animation which is a way to deal with several equivalent behaviours at the same time by allowing the use of 'placeholders' instead of values for state components and/or action arguments. These placeholders would provide an abstraction when it is of little interest to refer to concrete values. To give an example, let us suppose that one wants to test what is the reaction of the librarian if a user requests a book which is not on the shelf, no matter which is the book. If one wants to be exhaustive, (s)he will have to make the test for every existing book which is not on the shelf. Choosing the action *Request* with a placeholder representing the whole class of borrowed books as parameter would solve the problem in a much more elegant and efficient way. Nevertheless, symbolic animation poses problems for the computations that the animator has to do (evaluating the satisfaction of a constraint, computing the next state, etc.) but we think that these problems can, at least to a certain extent, be addressed using Abstract Interpretation techniques [3].

**Backtracking** is also one thing we want the tool to provide. By alternating building and reviewing mode, the user will be able to do backtracking : after having explored a life up to a certain point, the user may wonder what the life would have looked like if (s)he had made another decision in some past state. What (s)he would do then is to activate the reviewing mode to go back to the intended past state and then reactivate the building mode to construct a new change and thereby start constructing part of another life (or several other lives if the backtracking mechanism is applied again).

**Production of traces.** We already know that the Albert II animator's contribution to the RE activity focuses on the validation task but, at this point, we have to be a little more precise and distinguish among two possible forms of validation by scenarios the animator can be used for:

- in the first one, some stakeholders are proposing scenarios to be tested against a specification. The opportunity is then given to them to almost directly experience the behaviour they want to test by playing with the animator;
- the second possible use does not put all the stakeholders in direct interaction with the animator but just lets some of them play with it and produce traces of the scenarios they have tested. These traces will then be submitted to other stakeholders for further validation.

The RE process supported by the animator is basically the one shown in figure 2 where the specification task is a black-box which in fact includes the analysis, modelling, elication, verification and validation subtasks not represented here.

The way how the first use of scenarios takes places has been extensively described in section 3. Now, we will develop the second use which

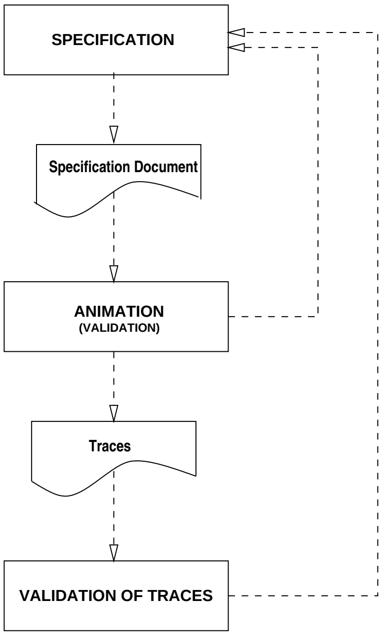


Figure 2: Animation as part of the RE process

poses the problem of the representation that will be given to the traces of the animation as scenarios. To give an idea of the kind of information that such traces should be able to contain, in Appendix B we give two small scenario descriptions in natural language. (These scenarios are derived from the library example – see Appendix A – and instances of the agent *User* are named Charles and Diana.) In order to be able to incorporate such characteristics as the ones emphasized in these scenarios, we have completed, in joint work with the other CREWS participants, a first version of a core scenario meta-model to which is associated an 'event-trace'-like representation. Hopefully, such a graphical representation will be generated automatically and will be used for visualizing traces of animations. Besides information needed for scenarios extracted from Albert II specifications, this meta-model is also able to express interesting features like exceptions, interrupts, views on scenarios, etc.

**Extensibility of the animator's repository.** A global life usually being infinite, what the users of the animator will be able to construct in a finite time is a **sublife** or, more precisely, a set of sublives if backtracking is used. In order to provide the functionalities sketched above, the animator has to store and maintain information contained in the created sublives (i.e. the states and changes explored during the animation). A first model of this knowledge has been written using ConceptBase [10]. Such meta-modelling language and tool were required because of a need for flexibility and extensibility. First, a link had to be made between the animator-related information and the Albert II language meta-model which already existed. The new meta-model extends the previous one by making an explicit link to parts of its semantics (i.e. some behaviours) which have been explored using the animator. The concepts it contains are the ones that were very briefly presented in section 2.2 of this document. Then, flexibility of the knowledge base is also required since, as mentioned in the introduction, it is our desire to integrate our research on animation with research that has the greater ambition to study how scenarios can help in RE. Links to tools and models built within the CREWS project will be greatly facilitated if a common flexible knowledge base is used. One great advantage of this has already been shown with the production of traces. But, as suggested in the second scenario example (see Appendix B), benefits are also expected to come from the integration of Albert II with other languages aimed at capturing requirements at other 'levels' than the strictly functional one (which is the main scope of Albert II). In particular, relating Albert II's concepts with the ones of a language like *i** [12], will help justify scenarios presented to stakeholders by including the organisational goals behind the behaviours they describe.

**Distributed application.** The prototype that was made for the initial version of the language has shown that it is not so easy to deal with several agents at once on the same screen. Therefore, it seems to be a good idea to split the windows corresponding to agents on different terminals. At each terminal, every stakeholder can have control on the agent(s) he is interested in.

## 5 Conclusion

Our goal is to continue research which was performed a few years ago on an animator for AlbertII specifications by focusing on its formalization and on extending its functionalities mainly with symbolic animation, backtracking and production of traces. As can be noticed from the above presentation, our contribution is still very small since our research has just started. Nevertheless, some work has been accomplished regarding the modelling of the animator's knowledge and the link with AlbertII's meta-model. A first version of a core scenario meta-model and its associated graphical representation have also been achieved in collaboration with the other CREWS participants. The tasks that we hope to accomplish in the near future are (1) to start working on the formalization of the computations made by the animator, (2) to become familiar with Abstract Interpretation techniques in order to see what is feasible in terms of symbolic animation, (3) to refine the animator's repository structure and the graphical representation for animation traces and (4) to start designing the tool in detail.

## 6 Acknowledgements

I would like to thank my supervisor, Eric Dubois, the members of the CREWS projects and the current and former members of the AlbertII team. A special thank also goes to Jean-François Raskin and Vincent Englebert for their interest in the 'formal part' of the work. All of them were of great help for me in producing this document.

## References

[1] Benner, K.M., Feather, S., Johnson, W.L. and Zorman, L.A. (1992) Utilizing Scenarios in the Software Development Process. *IFIP WG 8.1 Working Conference on Information Systems Development Process*, December 1992.

[2] Borgida, A., Mylopoulos, J. and Reiter, R. ...and nothing else changes : The frame problem in procedure specification. Technical Report DCS-TR-281, Dept. of Computre Science, Rutgers University, 1992.

[3] Cousot, P. and Cousot, R. Abstract Interpretation Frameworks. *Journal of Logic and Computation*, 2(4):511-547, 1992.

[4] CREWS (Cooperative Requirements Engineering with Scenarios), Esprit Project Programme (contract number 21.903), August 1996. Information available at `http://SunSite.Informatik.RWTH-Aachen.DE/CREWS/`

[5] Dubru, Frédéric (1993) Prototypage de Spécifications Formelles des Besoins: d'ALBERT vers OBLOG. Master thesis, Computer Science Department, University of Namur, Namur (Belgique), June 1993.

[6] Eric Dubois, Philippe Du Bois and Frédéric Dubru, Animating Formal Requirements Specifications of Cooperative Information Systems. In *Proc. of the Second International Conference on Cooperative Information Systems – CoopIS-94*, Toronto (Canada), May 17-20, 1994.

[7] Philippe Du Bois (1995) *The Albert II Language: On the Design and the Use of a Formal Specification Language for Requirements Analysis.* PhD thesis, Computer Science Department, University of Namur, Namur (Belgique), September 1995.

[8] Hsia, P. et al. Formal Approach to Scenario Analysis *IEEE Software*, pp. 33–41, 1994.

[9] Michael Jackson and Pamela Zave Deriving Specifications From Requirements: An Example. In *Proc. of the 17th International Conference on Software Engineering – ICSE'95*, Seattle WA, April 1995

[10] Jarke, M. and Gallersdörfer, R. and Jeusfeld, M.A. and Staudt M. and Eherer S. (1995) ConceptBase - a deductive object base for meta data management. *Journal of Intelligent Information Systems, Special Issue on Advances in Deductive Object-Oriented Databases*, 4(2):167-192.

[11] Lalioti, V. and Theodoulidis, B. Use of Scenarios for the Validation of Conceptual Specifications In *Proc. of the Sixth Workshop on the Next Generation CASE Tools*, Jyvaskyla, Finland, June 1995.

[12] Eric Yu, Philippe Du Bois, Eric Dubois and John Mylopoulos (1995) From organization models to system requirements - a "cooperating agents" approach. In *Proc. of the Third International Conference on Cooperative Information Systems – CoopIS-95*, Vienna (Austria), May 9-12, 1995. University of Toronto Press inc.

# Appendix A : Albert II specification of a (very) simple library

## Data Types

### Constructed Types

*BOOK* = ENUM *[ < "Albert II for Dummies",Dubois > ,*
*< "UML's formal semantics",Rumbaugh > ,*
*< "Specifying with C++",Gates > ]*

## Main

### Society

*(Librarian)*
*(User)))*

`The society is composed of a librarian and several users`

## Librarian

### DECLARATIONS

#### State Components

*OnShelf* set-of *BOOK*

`The books on the shelf are the ones which are not lent`

*Awake* instance-of *BOOLEAN*

`The librarian can be either awake or asleep`

#### Actions

$*$*Lend(BOOK,USER)* $\longrightarrow$ *USER*

`The librarian can lend a book to a user`

*WakeUp*

`The librarian can wake up`

*FallAsleep*

`The librarian can fall asleep`

*OKBorrowing*

`A borrowing (composed action) can succeed`

*FailedBorrowing*

`A borrowing (composed action) can fail`

### BASIC CONSTRAINTS

#### Initial Valuation

$Card(OnShelf) \geq 100$

Initially, the number of books on the shelf is 100 or more

## LOCAL CONSTRAINTS

### State Behaviour

$\neg$ ( $Card(OnShelf) < 10$ )

The number of books on the shelf can never go under 10

### Effects of Actions

$Lend(b,\text{-})$: $OnShelf := Remove(OnShelf,b)$

When the librarian lends a book, the book is taken out from the shelf

$WakeUp$: $Awake :=$ TRUE

As soon as the librarian wakes up, he is awake

$FallAsleep$: $Awake :=$ FALSE

As soon as the librarian falls asleep, he is not awake anymore

$u.Return(b)$: $OnShelf := Add(OnShelf,b)$

When a user returns a book, the book is put back on the shelf

### Capability

$\mathcal{F}$ ( $Lend(\text{-},\text{-})$ / $Awake =$ FALSE $\vee \neg\, b \in OnShelf$ )

The librarian cannot lend a book if he is asleep or if the book is not on the shelf

$\mathcal{F}$ ( $WakeUp$ / $Awake =$ TRUE )

The librarian cannot wake up if he is already awake

$\mathcal{F}$ ( $FallAsleep$ / $Awake =$ FALSE )

The librarian cannot fall asleep if he is already asleep

### Action Composition

$OKBorrowing \leftrightarrow u.Request(b)\ ;_{<5min}\ Lend(b,u)$

In a borrowing which succeeds, the request made by the user is followed, within 5 minutes, by the lending of the requested book to the user

$FailedBorrowing \leftrightarrow u.Request(b)$ ; DAC

In a borrowing which fails, the request made by the user is not followed by any reaction from the librarian

## COOPERATION CONSTRAINTS

### Action Perception

$\mathcal{I}$ ( $u.Request(\text{-})$ / $Awake =$ FALSE )

When he is asleep, the librarian ignores the users' requests

$\mathcal{K}$ ( $u.Return(\text{-})$ / TRUE )

The fact that a user returns a book is always perceived by the librarian

### Action Information

$\mathcal{K}$ ( $Lend(\text{-},u1).u2$ / $u1 = u2$ )

A librarian only lends to the intended user

## User

# DECLARATIONS

### State Components

*Borrowed* set-of *BOOK*

▌ `A user can possess a set of books that he has borrowed`

### Actions

*Request(BOOK)* $\longrightarrow$ *LIBRARIAN*$*$

▌ `A user can request a book from the librarian`

*Return(BOOK)* $\longrightarrow$ *LIBRARIAN*

▌ `A user can return a book to the librarian`

# BASIC CONSTRAINTS

### Initial Valuation

$Borrowed = \{\}$

▌ `Initially, a user has not borrowed any book`

# LOCAL CONSTRAINTS

### State Behaviour

$b \in Borrowed \Longrightarrow \Diamond_{\leq 1month}\ \neg\, b \in Borrowed$

▌ `A user does not keep a book for more than a month`

### Effects of Actions

*Return(b): Borrowed := Remove(Borrowed,b)*

▌ `Once a user has returned a book, he does not possess this book anymore`

*l.Lend(b,-): Borrowed := Add(Borrowed,b)*

▌ `Once the librarian has lent a book to a user, the books comes into the user's possession`

### Capability

$\mathcal{F}$ ( *Return(b)* / $\neg\, b \in Borrowed$ )

▌ `A user cannot return a book he has not borrowed`

# COOPERATION CONSTRAINTS

### Action Perception

$\mathcal{K}$ ( *l.Lend(-,-)* / TRUE )

▌ `A user always perceives he is lent a book`

### Action Information

$\mathcal{K}$ ( *Request(-).l* / TRUE )

▌ `The fact that a user requests a book is always shown to the librarian`

$\mathcal{K}$ ( *Return(-).l* / TRUE )

▌ `The fact that a user returns a book is always shown to the librarian`

# Appendix B : Scenarios derived from the specification

## Scenario 1

Charles requests "Specifying with C++" by Gates. 2 minutes and 31 seconds[a] after the request has been formulated, the librarian understands its meaning and starts acting. The book is on the shelf[b] and it takes the librarian 25 seconds[c] to lend it to Charles. Charles then keeps the book for some time (less than 1 month)[d] before he returns it.

Noticeable characteristics :

- a. real-time value between actions
- b. value of state components allowing actions to take place
- c. real-time value for action duration
- d. implicit assumptions derived from the specification

## Scenario 2

November the 2nd 1996, at 10:30 PM, Diana makes two book requests at the same time[e,f] : one for "UML's formal semantics" by Rumbaugh and the other for "Albert II for dummies" by Dubois. The librarian being particularly attentive, he manages to get both the request[g]. Unfortunately though, one of them can't be satisfied : "UML's formal semantics" is currently lent to another user[h]. Nevertheless, "Albert II for dummies" is borrowed and returned 2 days later. After he has worked hard, the librarian has to rest a bit[i] and, therefore, falls asleep.

Noticeable characteristics :

- e. fully instantiated action occurrence (including absolute time)
- f. simultaneous actions
- g. perception of actions performed by other agents
- h. value of state components preventing actions to take place
- i. goals and intentions