CREWS Report Series 97-13

The Albert II Specification Animator

Patrick Heymans

August 12th, 1997

Computer Science Department,
University of Namur

This paper reports on work that has been and is currently accomplished within the ESPRIT IV LTR project CREWS (*Cooperative Requirements Engineering With Scenarios*)

The Albert II Specification Animator[1]

Patrick Heymans, Computer Science Dept., University of Namur

Abstract

Writing requirements specifications of complex composite systems makes it necessary to have a language which is both formal to allow reasoning on specifications and declarative enough to allow the analyst to express himself in a natural way. **Albert II** is a language that tries to achieve these goals and, besides that, provides templates that guide the analyst in writing the specification. However, because of its formality, the resulting specification cannot in general be read by the various stakeholders. Validation tools are therefore required, among which an animator. In this paper we aim at giving an overview of the functionnalities and the architecture of the animator for **Albert II** specifications.

1. Introduction

The validation task which takes place during the Requirements Engineering (RE) of software-based systems certainly cannot be considered as 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[2] and want to unambiguously express constraints involving such things as real-time aspects, undeterminism 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 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 from it. Simulation is also part of these techniques.

[1] This work is funded by the European Community within the framework of the ESPRIT IV LTR project CREWS (*Cooperative Requirements Engineering With Scenarios*).

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

- *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 **Albert II** language [DuB95, DuB97] 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 exisiting environment or the 'machine' (system) [Jac95] 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. **Albert II** 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 take for validating **Albert II** specifications can be classified among the behavioural techniques and we call it *animation* [Dub93, Dub94,Hey97]. 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 system 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. In other terms, our approach really comes down to test if a given *scenario* [Rol96], proposed by one or several stakeholders, is compatible with the requirements specification.

The main challenges we want adress in our animation tool are the following :

- to provide a *distributed* tool which allows different stakeholders to cooperatively animate a specification, each of them being responsible of animating the part of the system / environment he is interested in ;

- to provide *backtracking* facilities in order to explore alternative (more or less normative, more or less exceptional) scenarios ;

- to provide feedback of the animation in a vocabulary that can be understood by its users. At least, this vocabulary is the one used in the specification and it reflects the application domain. In any case, we want to hide, as much as we can, the technical details of the semantics which underlies the specification language ;

- to deal appropriately with *undeterminism*. As already mentioned, **Albert II** 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 animate a specification since executability is not present, as shown in [Dub93]. Whenever undeterminism is encountered, the animator asks the user to make a decision in order to remove it. Not overloading the user with too many and/or too technical questions is therefore a major goal.

In section 2, we introduce the Aditec production cell case study which we will reuse for illustration purposes at various places afterwards (this case study is also the one used in the project for the purpose of demonstrating the results of the cooperation between Aachen and Namur teams. Section 3 gives an overview of the constructs of language with the help of the example. There, we also try to give a sufficient intuition of the concepts underlying the formal semantics of the language so that the main concepts of the animation be understood. Section 4 presents our approach and situates it wrt the main approaches usually proposed for animation. Section 5 sketches the architecture of the tool and section 6 introduces how it will have to be used. The paper finally concludes on what has been accomplished so far and what is still left to do.

2. The Aditec Production Cell Example

Overview

The example we will use throughout this paper is the one of a manufacturing production cell which is close to the production cell we could observe at the Aditec factory in Aachen. Aditec is a test factory which builds gear boxes for cars and whose goal is not to make profit but to serve as a laboratory for new technologies coming from research before they are applied in industry.

Production management is a complex task. It involves three computer-assisted tasks to be performed : production planning (PP), detailed scheduling (DS) and shopfloor data collection (SDC). Production itself is concentrated at the shopfloor level. Customer orders are first processed by PP which performs coarse-grained association of required parts (and sub-parts) to machine groups. It is also able to provide time boundaries for the order to be accomplished. All this information is then passed to DS which has a fine-grained vision of how production is organized from the point of view of machines. Taking into account the processing of other orders, DS tries to optimize the machines' productivity. Detailed scheduling information is then put into the SDC's database. A foreman handles this information and dispatches it to workers in charge of specific machines. The information consists of the machine order to be performed at the current time and is displayed on the terminal associated with the concerned machine. Workers, besides controlling the machines, are invited to give feedback on the progress of the order they are performing at the terminal. The information thereby given feeds back PP and DS so that rescheduling and supervision can happen at higher level.

The production cell

For our purposes, we decided to restrict our attention to the shopfloor level of the factory. The use of the **Albert II** language and animator will therefore be illustrated with the help of a specification we have written and which models a generic production cell. Rather than trying to model a particular production cell present at Aditec (with all its set of specific details), we have tried to abstract the main elements from the cells we could observe in order to keep the example largely understandable by anyone and still be able to illustrate the main concepts of our language and tool. Nevertheless, we want to insist on the fact that having written an *abstract* specification is not only a trick we used to keep the example simple enough for our illustration purposes, this is also a technique which is often used in real life situations. For example, if one wants to describe a system working in some environment, he might be interested in specifying the system's components in detail but will be satisfed with a very loose definition of the environment components, just concentrating of what has an impact on the system's requirements. Another use of a loose style of specification might be to avoid making design decisions that should be made at later stages of system development. Both these uses will be illustrated in the specification.

The production cell is composed of three main elements : (i) a machine, (ii) a worker who operates and supervises the machine and (iii) a terminal on which the worker can observe the orders he has to process and which is also the device he uses to keep the scheduling system informed of what is actually being done at the shopfloor level. Now, we will, in turn, characterize more precisely each of the components of the production cell.

The worker

The behaviour of the worker is highly undeterministic in the sense that it is not possible (nor even recommended) to model it as a strict sequencing of actions. Anyway, for identifying the basic actions he is able to perform we have thought it was a good idea to start from two typical scenarios :

Scenario 1 : Manual mode
(1) The worker asks the terminal to display the current order. (2) He reads it and (3) signals the terminal he starts setting up the machine in order to process it. (4) He loads the approriate tools on both machine's slots and (5) asks the machine to load the requested NC-program[3]. (6) He signals the terminal the machine is starting to processing the order.(7) He puts a few parts in the input buffer until set-up is complete and then (8) asks for one part to be produced. While it is produced, he (9) puts additional parts in the input buffer. (8) and (9)

[3] NC-program stands for "Numerical Control program" which is a kind of program that guides the operation that are performed by machine-tools in manufactures.

are performed several times up to the time all the necessary input parts have been introduced. Then (9) is performed a few times again and, while the machine is working, (10) he takes produced parts from the output buffer. (11) He finally signals the terminal that the processing of the order is finished.

Scenario 2 : Batch mode
Steps (1) to (7) are the same as in the first scenario. (8) The worker asks the machine to perform production in batch mode. During this time (9) he adds parts in the input buffer and (10) gets produced parts from the output buffer. (11) He finally signals the terminal that the processing of the order is finished.

We insist that these two scenarios are only a starting point to identify actions performed by the worker. The set of all the possible behaviours that we want to model in the specification contains much more that the ones depicted above. Many deviations can be considered : batch mode is interrupted because the worker as not put enough parts in the input buffer, the worker does not report on the progress of the order's processing, tools and/or NC-program do not have to be loaded because they were the same as for the previous order processed, the machine can produce junk parts, etc... Furthermore, we have to note that, even if we restrict ourselves to the two scenarios above, we would have undeterminism anyway since it can be noticed that the sequencing and number of occurrences of actions is not at all defined in every situation.

The machine

The machine we consider here *transforms* parts (as opposed to *(dis)assembling* them). The transformation the parts have to undergo is recorded by an NC-program which has to be loaded into the machine before any transformation process can start. The transformation inside the machine is performed by a set of tools which have to be mounted on specific locations and which are also necessary for a transformation process to be started. We will consider that there are two such locations on the machine. The machine also has an input and an output stock. We do not have to suppose that these are *physically* differentiated, we just consider them *logically* differentiated.

The behavior of the machine can be represented by the state-transition diagram shown in *fig.1* which is quite dependent from the way the machine has been specified in **Albert II** but which, nevertheless, gives a good idea of its required behaviour. We use the convention that the start of an action *act* is represented by *<act* and its end by *act>* ; an instantaneous action is represented by *<act>*. Requests from the worker are instantaneous actions and have the prefix *w.*

Other comments that are not trivial from this state-transition diagram are that :

- the NC-program is supposed to perform some not explicitely modeled initialization operations just after having been loaded. This lasts 3 minutes and makes the production impossible during this period ;
- when the worker requests for *Batch mode* production, the machine puts itself in a state (*WaitForPart*) in which input parts are processed iff the following conditions are satisfied : (1) a part must be available in the input buffer and (2) the output buffer must not be full. If production cannot take place during 1 minute, the machine automatically returns in *StandBy* state.

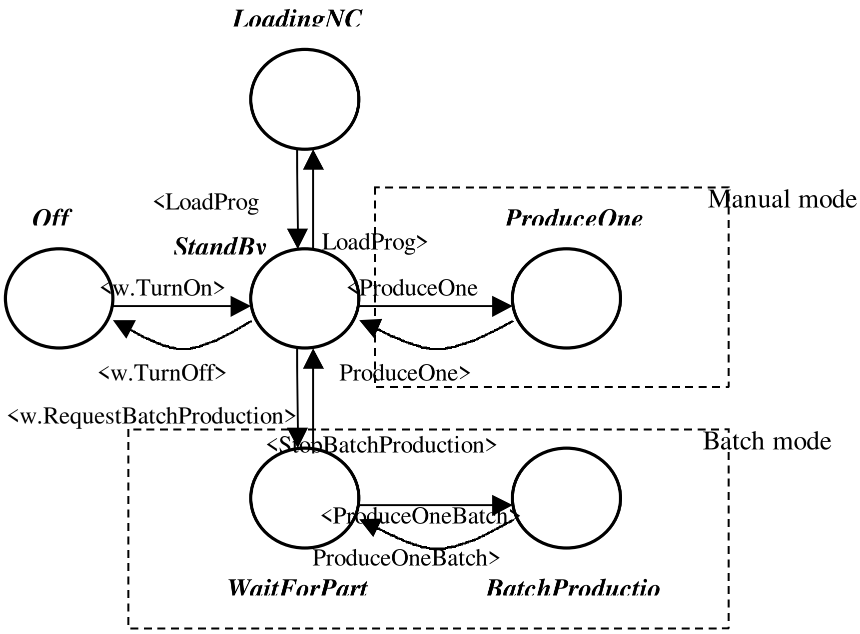


Figure 1 : State-transition diagram of the machine

The terminal

In order not to take into account the scheduling system, the functionning of the terminal (including part of the actual shopfloor data collection) has been drastically simplified. The orders that have to be fulfilled are just put into a queue whose content is not allowed to be changed dynamically (which is the consequence of the rescheduling that happens in the real system). The worker is only requested to process another order if he has completed the following sequence of tasks for the previous order : report on set-up, report that the machine has started production for the order, report that the order has been completely processed.

The only thing the terminal does is therefore to display the current order. The terminal is thus assumed to be working according to the very simple state-transition diagram shown in *fig.2* and where *int-act* represents any internal action performed by the terminal in response to an action of the worker. There is a one-to-one correspondence between the actions of the terminal and the worker's requests which have already been explained in the subsection devoted to the worker and that we will therefore not detail more.

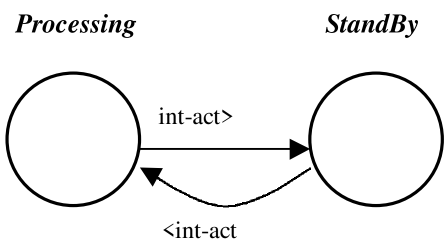


Figure 2 : State-transition diagram of the terminal

Some excerpts of the specification that resulted from this analysis is shown in section 3 below where we present the constructs of the language. The full **Albert II** specification of the production cell can be found in the Appendix.

3. Overview of the **Albert II** language

Syntax

Albert II specifications are made of two main parts : (1) the 'static' part, made of *type* and *operation* definitions and (2) the 'dynamic' part related to the specification of *agents*. The 'static' part is written at the beginning of the specification because it defines types and operations which will be needed at a variety of places afterwards. An example is the type ORDER which is a constructed type defined as a cartesian product with four fields : the first denotes the type of the required input parts, the second specified the quantity to be produced, the third, the NC-program needed by the machine and the fourth, the status of the order (Waiting, Started, Done, etc...).

> *ORDER=CP [inputType : INPUT_PART_TYPE , quantity : POSITIVE_INTEGER , ncProg : NC_PROG , status : ORDER_STATUS]*

Facilities to build types and operations are present in the language. First of all, there exists a set of predefined types (*INTEGER,STRING,RATIONAL,...*) and associated operations. Type constructors (as *CP,SET,...*) are also available together with a series of operations on parametrized types (e.g. *Empty ? : SET (#) -> #*).

The 'dynamic' part generally represents the main part of the specification. Agents correspond to entities of the real world that appear to have some 'autonomy' (e.g. a person, a subsystem, an external machinery). The decomposition of the specification into several agent specifications allows to reduce the dependencies between the various parts. The agents of the specification of the cell are the *Worker*, the *Machine* and the *Terminal*. Agents are grouped into *societies*. In our example, there is a single society grouping all the agent and we have called it *Cell*. In general, societies can form a arbitrarily complex hierarchy whose agent represent the terminal nodes. The specification of an agent is itself decomposed into a *declaration* part and a part which consists of *constraints*. This part aims at pruning the number of admissible *behaviours* (or *lives*) an agent can have. Society are not further specified since their behaviour in fact is just the 'sum' of the behaviour of the agent they contain : they are just used to structure the specification more clearly. In our example, the society structure is as follows :

SOCIETY CELL

> *(Machine)*
>
> *(Terminal)*
>
> *(Worker)*

Agents are usually responsible for some 'data' they maintain. In this case, the terminal (representing also part of the shopfloor data collection) has to record information about the orders which have to be communicated to the worker in order to be processed. Therefore, the terminal possesses a *state component* called *Orders* which is defined as a table whose domain elements are of type *ORDER* (see above). Similarly, every machine has two slots where tools can be mounted in order to perform machining operations : *ToolLocation1* and *ToolLocation2* which can take either a value of type *TOOL* or the special *UNDEF* value which here denotes that no tool is mounted in the slot (adding *UNDEF* to the set of values defined by a type is done by adding a * after the type name).

STATE COMPONENTS

> *ToolLocation1* <u>instance-of</u> *TOOL** → *Cell.Worker*
>
> *ToolLocation2* <u>instance-of</u> *TOOL** → *Cell.Worker*

STATE COMPONENTS

Orders table-of *ORDER* indexed-by *POSITIVE_INTEGER* → *Cell.Worker*

Agents are able to perform *actions* that can have parameters like the action *InstallTool1(TOOL)* which is performed by the worker in order to mount a tool in the first slot. Both actions and state components can be *exported* by their owner agent to one or several other agent(s). Exportation is represented by an arrow preceding the name of the agents to which the action/state component can be shown.

InstallTool1(TOOL) → *Cell.Machine*

As already mentionned, 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 a methodological guideline for the specifier who wants to express something. The types of constraints are grouped into four families : (1) *basic* constraints, (2) *declarative* constraints, (3) *operational* constraints and (4) *cooperation* constraints. 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). A commented example of each of them is given below :

DERIVED COMPONENTS

> The output buffer being full is determined by its content being equal to its capacity.

$OutputBufferFull \triangleq (Card(OutputBuffer) = MaxOutputBufferCapacity)$

INITIAL VALUATION

> Initially, the machine is assumed to be off, with no tool mounted and no NC-program in memory.

$ToolLocation1 = UNDEF$

$ToolLocation2 = UNDEF$

$Status = Off$

$Prog = UNDEF$

Declarative constraints allow to express in a declarative manner constraints on the whole behaviour of an agent. *State behaviour* constraints are used to express conditions that have to be satisfied at each moment of the agent's life. *Action composition* constraint restrict the occurrences of actions to certain conditions of sequencing, parallelism, alternative, repetition etc... *Action duration* constraints can give the actions maximum, minimum or exact durations. These three types of constraints are illustrated below :

STATE BEHAVIOR

> The output buffer's content can never be greater than its capacity

$[\,]\ Card(OutputBuffer) \leq MaxOutputCapacity$

ACTION COMPOSITION

> A (perceived) request to produce a part is immediately processed

$comp2 \leftrightarrow w.RequestProduceOne <0\ sec> ProduceOne(\_)$

> Producing a part in batch mode can either succeed of fail

$TryProduceOneBatch \leftrightarrow ProduceOneBatch(\_) \oplus ProduceOneBatchFail(\_)$

ACTION DURATION

> Producing a part takes at least 70 seconds

$ProduceOne(\_) \geq 70\ sec$

> Producing a part takes at most 85 seconds

$ProduceOne(\_) \leq 85\ sec$

Operational constraints are *preconditions*, *effects of actions* and *triggerings*. The semantics of such patterns is broadly self-evident except for effects of actions for which we have to give some precisions. Actions can have a pre- and a post-effect. In the constraint's syntax, they are separated by an (optional) part between square brackets which is a condition determining if the post effect takes place or not. Another point is that the pre-effect only lasts until the end of the action occurrence while the post-effect lasts until the affected state components are changed by other action occurrences.

PRECONDITIONS

> Installing a tool in the first slot requires that the machine is either loading an nc-program, off or in standby. It also requires that the first tool location is empty.

InstallTool1(_) :(Cell.Machine.Status = LoadingNC ∨ Cell.Machine.Status = StandBy ∨ Cell.Machine.Status = Off) ∧ ToolLocation1 = UNDEF

EFFECT OF ACTIONS

> Producing a junk part has the effect of temporarily switching the machine's status (see state-transition diagram). It also removes a part from the input buffer.

ProduceOneFail(p1) :
Status := ProduceOne ;
InputBuffer := Remove(p1, InputBuffer)
[]
Status := StandBy

TRIGGERINGS

> Stopping batch production has to take place when the machine has been waiting for a part for exactly 60 seconds

$Lasted_{60\ secs}\ Status = WaitForPart\ /\ 0\ secs \rightarrow StopBatchProduction$

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 part of the state of other agents (*state perception*). Commented examples follow :

STATE INFORMATION

> The content of a tool location is always made visible to the worker

$\mathcal{K}(ToolLocation1.w\ /\ TRUE)$

ACTION INFORMATION

> Getting a part from the output stock is always made visible to the machine

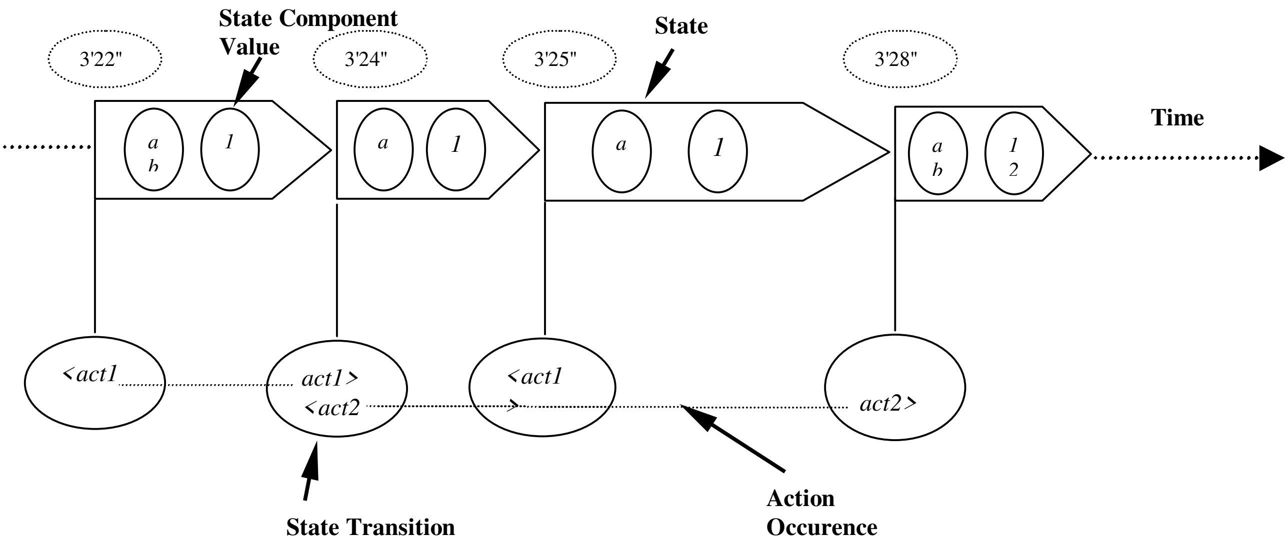
$\mathcal{K}(\ GetOutputPart(\_).m\ /\ TRUE\ )$

We are not going here to detail of each type of constraint (this is provided in [DuB97]). The specification in the Appendix can also be used to have a more precise idea of the expressiveness associated with the constraint patterns. One important remark about the constraints we have just shown is that they introduce a lot of undetermism, or, more precisely, their absence (or the presence of weak constraints) often yields undeterminism. For example, if no initial valuation constraints are specified for some state component (and if it is not constrained by a state behaviour constraint), its value at the starting point of an animation can be any value defined by its type and a choice has to be made by the person in charge of animating the concerned agent. From the point of view of actions, the absence of (or the presence of weak) preconditions allows an action to take place anytime, supposing that no other constraint restricts its occurrences, as it is possible for triggerings, state behaviours or compositions. But the same also applies to these constraints. E.g., if no composition constraint requires that two actions always have to follow each other in some defined order, their occurrences can take place in any way. Also from the point of view of action duration, looseness of specification is possible. If they are not declared as instantaneous or with a precise duration, actions can take any duration (possibly within some defined boundaries). The time at which the action begins and the time at which an action ends are therefore one of the most common decisions the users of the animator will have to make. Finally, we should also mention that information and perception of action occurrences or state values, if not ('sufficiently') constrained by cooperation constraints may or may not happen at some points in time. Again, when animating, the users will have to make the choice.
Places where undeterminism can take place in a specification are too many to mention here. What is important to remember is that it is present all over the specifications and that the main way by which it is solved in the animator is interactivity.

Semantics

The semantics of an **Albert II** specification is given by mapping it to a real-time temporal logic called **Albert-KERNEL**. The set of axioms which results from the translation of an **Albert II** specification into **Albert-KERNEL** defines a set of *models* of the specification. Each of these models is made of the 'sum' of the lives of the instances of agents it contains. It is important to note that at the level of a model we talk about *agent instances* while, at the level of the specification, agents (classes) are declared. In a particular model and unless otherwise stated in the specification (by declaring the agent as single), 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.

The life of an agent instance is an (possibly infinite) alternate sequence of *states* and *state transitions* (see *fig.3* below)[4]. The sequence is indexed by a real-time value which increases throughout the sequence.



[4] In order to resolve the *frame problem* [Bor92], state transitions (more precisely, the events they contain) are considered the only way by which state components can change their value.

Figure 3 : Partial life (or behaviour) of an agent instance

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-KERNEL** notion of event is made necessary to give a semantics to the **Albert II** notion of action. In fact, action occurrences can have some duration (see, e.g., *act2*) or can be instantaneous (as the second occurrence of *act1*). Actions have been therefore associated with events : each of them 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 / 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. Animating **Albert II** specifications consists in operationalising the construction of such behaviour but, as we will see later, with the distinction that animated behaviours can be admissible or not.

4. Approaches to the animation of formal specifications

Generally, when doing animation of specifications written in a formal language, there are roughly two possible approaches:

- one is to translate the specification into a program. Depending on the language and on the tool, different levels of automation of the translation process are possible, going from *fully automatic* to *manual*, the latter being of course not very practical. It is also possible that the specification is itself an executable program (whose execution mechanism is often embedded in an existing programming language). Typically, the programming languages that are used for this kind of tasks are declarative : functionnal or logic programming languages.

- a second possible approach is to establish a mapping from the specification language's constructs to automata modelling the allowed states and transitions as defined in the specification. This approach, which is generally used for model checking [Hen94, Ras97a] of formal specifications, is now also being used for animation. Currently, the tool sets described in [Her97, Alg95, Statemate] propose both exhaustive (model-checking) and interactive (animation) techniques. Regarding the category of systems that map specifications to automata, we are particularly interested in the ones that translate temporal logic specifications into automata. Currently, there is some work performed on mapping more and more expressive temporal logics to automata. In particular, [Ras97a] describes how to perform such a translation for a logic which is close to **Albert-KERNEL**.

The approach we have adopted cannot be classifyied in any of the two above but before caracterizing it more precisely, we are now going to motivate our choice by giving the pros and cons of the two approaches above.

The first approach requires that, to a very large extent, the specification language be operational. This implies that, although non-determinism might be present [Hay89], it is not as widespread as in languages like **Albert II.** Languages like Z [Spi92] or VDM [Jon91], which are the main languages for which animation systems of the first kind have been built [Sid97, Bre94, One92], are languages that are used to specify isolated functions or operations of the system without modelling explicitely whole behaviours of the system together with its environment. Languages like **Albert II,** on the other hand, aim at embodying the environment in the specification. Besides that, languages like **Albert II** do not limit their scope to isolated functions or operations but they allow to specify whole system and environment behaviours (i.e. all the possible sequences of actions that can be performed by all the agents). All this, of course, requires more expressiveness which, when it comes down to animation, necessitates the usual techniques to be rethough. For example, it is quite easy to understand that animating a Z specification consists for a user to enter the input of some function or operation and observe what are (all) the (possible) output produced [Bre94]. On the contrary, for a language like **Albert II**, we must be aware that beside setting the parameters of actions, animating involves such things as chosing the time at which actions take place, chosing one of the possible outputs of an operation call (since it will necessary in order to continue to elaborate the behaviour in progress), determining the perception by some agent of a state or an action coming from another agent,... A first thing we can conclude then is that, in our case, a much more

interactive animation technique is required. As we will see further, our approach does not completely abandon the technique we have just criticized but reuses it embedded in a higher-level mechanism, restricting its usages for tasks where little undeterminism is involved, that is, where little interactivity is needed, like evaluating a precondition, computing the effect of an action,...
Besides that, there are two other main problems with this approach. One of them is the fact that, in order to become executable, the specification often has to be transformed and, thereby, concepts that are not of interest to the specifier have to be introduced and others, which are of interest for him, become hidden in the resulting executable form. It is the case, for example, when one tries to transform a first order logic specification into a Prolog program : formulae are rewritten as Horn clauses, additional predicates have to be introduced (due to skolemisation transformations),... The other problem is of course expressiveness : not all specifications can be executed because the style they allow to express simply some properties would cause non terminating programs to be produced.

The second approach we have identified is the one which makes use of automata. Although, as we have already mentionned, work has been performed in order to translate more and more expressive logics into automata, we still do not have any method to translate **Albert II** to automata. In fact, in order to do this, the following transformation steps would have to be defined :

1. transform an **Albert II** specification into an **Albert-KERNEL** set of axioms (this task is currently being finalized within the Albert team in the context of another project) ;
2. transform the **Albert-KERNEL** axioms into axioms of a simpler logic which can be translated to automata like it has been done in [Ras97b] for mapping the MTL logic with dense time semantics to MTL with fictitious clock semantics using the framework of abstract interpretation ;
3. transform the set of axioms of the simpler logic into automata.

If we do that, we will not only be able to animate a specification but we will also have a decision procedure for **Albert II** specifications. We would therefore be able to do model checking of specification and also check various liveness and safety properties on it. Nevertheless, the automata-based approach suffers some drawbacks from the animation point of view. First, and similarly to the previous approach, the transformation the specification will have to endure will make it almost unrecognizable to the people who have written the **Albert II** specification. One big advantage of the approach we have chosen, as we will see, is that we keep the dialog with the users at the **Albert II** level. With the first semantics of the language (as defined in [DuB95]), which was based on the **Albert-CORE** logic, the problems were that additional arguments of predicates had to be introduced for keeping track of action occurrence numbers, that new predicates were introduced for giving a semantics to the information and perception mechanisms, that the agent hierarchy was flattened,... The new **Albert-KERNEL** logic tends to solve some of these problems but these will reappear anyway at the second step of the transformation.
A second drawback is the loss of expressiveness due to the second step of the transformation which has to restrict the logic so that it can be tranformed to automata. Nevertheless, loss of expressiveness in inevitable when one wants to animate an **Albert II** specification since it is written in a dense-time temporal predicative logic which is far too expressive to be animated as it is.
But, besides the two drawbacks mentioned above, one big advantage of the automata-based approach is that an automaton corresponding to a specification only admits sequences of transitions which belong to models (admissible behaviours) of the specification. Since the behaviours one can build by interacting with the animator can only be finite (because, of course, they can only be constructed in a finite time), they consist in prefixes of possible lives of the agents. That these prefixes are parts of admissible lives is guaranteed by construction of the automaton while, with our approach, no guarantee can be given : the constructed part of behaviour may be such that it will generate future obligations that no life of the system would never be able to verify and we have no way to check that since it would require building the behaviour until the end (which is infinitely far from the starting point).

Finally, in order to be complete wrt to the examination of the pros and cons of all approaches, we have to mention that the automata-based approach suffers from a serious drawback know as the *state-explosion problem* [Hen94] which causes the size of the automata and (therefore) the computation time to increase exponentially as the specification grows in size and which, in order to be reduced, requires optimization techniques such as symbolic encoding of states or on-the-fly generation of the automata.

How can our approach be characterized ?

The approach we have chosen can be described as the direct encoding of the *upper level* of the **Albert II** language's semantics in an algorithm that will take the specification as input. Since this algorithm, in order to animate, requires the specification (which, to a certain extent, is some kind of algorithm) as input, it is often refered to as *meta-algorithm.*

As we have already mentioned in section 3, Albert II is made of a series of patterns on top a real-time temporal predicative logic. The way these patterns are translated into temporal logic expressions is defined by axiom schemas. This means that the temporal logic statements corresponding to filled-in patterns are always translated to logic formulae with the same structure and, therefore, similar meanings. We thus estimate that the semantics of these statements could be safely 'hard-coded' into a meta-algorithm which operationalises the properties that the specification has to satisfy by performing particular actions (e.g. checking conditions, requiring events to take place, refusing events to take place, ...) at definite moments of the animation. We will not go here into the detail of such an algorithm (it will be detailled in a subsequent report), we will just give an example.

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 the closest and the latest time at which the end of the action has to take place. This information is put into the 'obligations' which 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 that 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.

We have said here above that we were not completely giving up the approach that consists in transforming specifications into declarative programs. In fact, we are reusing similar techniques for computations on parts of the specification which do not conform to a strict predefined structure and which, therefore, cannot be 'hard-coded' in the meta-algorithm. This is typically the case for arbitrarily complex logical or free expressions that can appear in particular slots of some constraint patterns. Of course, only a restricted subset of these constraints can be treated but, fortunately, they are much less used than more constrained patterns. The strategy we adopt for developing the animator is first to start from more constrained parts and then progressively extend the coverage to more unconstrained ones.

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 agents sublives we have constructed up to the current timepoint are not admissible sublives wrt to the constraints expressed in the specification. For example, if we have an action composition constraint imposing that all occurrences of the action *TurnOn* of some agent are followed, within 2 minutes, by an occurrence of the *SetUp* action and if, at the current time in the animation, an occurrence of *TurnOn* takes places, we will only be able to conclude, 2 minutes after, that the *SetUp* action can or cannot actually take place by looking at the values of its *context* (preconditions, etc...) at that moment. The reason for this is that, since undetermism is largely used in **Albert II** specifications, future contexts cannot be determined as they require choices being made by the users of the animation.
As a conclusion for this remark, we can say that while automata-based techniques provide proof that there will always be an admissible behaviour beginning with the animated prefix, our technique does not allow to do that. But, on the other hand, we have to consider two things :

- One is that this claim should be attenuated by the fact that with our technique we are still able to give the list of obligations that the animation of a life's prefix has generated for the future (even though we are not able to prove that they can be satisfied). This list is very important : the users may be able to estimate with a reasonable certainty that some obligations will be satisfied and/or that some will not. And, even if they are not able to do that, we can imagine that, for some critical properties, the list of obligations may be used as input for a theorem proving system [Cha97] which, in this perspective, is seen as a complemetary technique to animation.

- Not being able to make proof is not has serious as not being able to find errors in specifications. And, not only we are able to find errors, but we are also able to give feedback on them in terms which are the terms of the people who have written the specification.

5. Architecture of the tool

At a high level of granularity, the architecture of the tool can be represented as shown in *fig.4*.

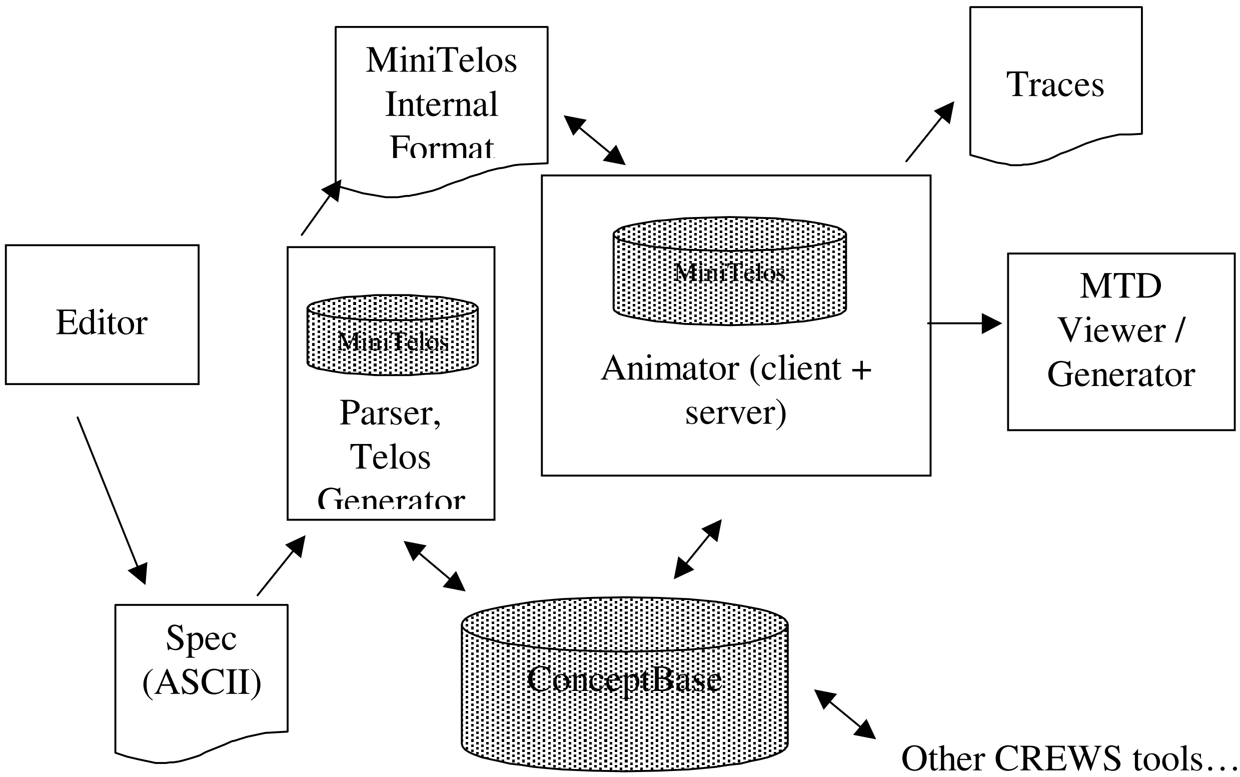


Figure 4 : High-level architecture of the Albert II specification animator

The specifications used in the animator are produced with the support of the Albert II CAT-Edit tool and are in ASCII format. A translation utility will parse the specification file and generate MiniTelos internal representation which is then merged with animation-specific information and used throughout the tool. MiniTelos is thus what we use for managing data within the animator. It consists of a library of reusable Java classes. Its internal representation is made of `serializable' Java objects that can be either written to a file or can be converted by MiniTelos into an ASCII file containing frames that can then be used by ConceptBase [Jar95]. These are the two ways persistent information can be stored. Animation client and server applications embed MiniTelos within their set of classes and use it to manage information. Animation clients only need information about agents managed locally and therefore only deal with *views* of the global MiniTelos base which is managed by the server.

On the other end of the animator, there is the production of traces. First, we intend to have very basic generation of textual traces but we plan to extend it afterwards by generating traces in extended MTDs (Message Trace Diagrams) notation and with the possibility to extract views on certain alternative paths, agents and/or time intervals.

As we will see in the next section, the animation server 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 Agent Managers 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 : management of local MiniTelos bases is performed by Agent Managers.

6. Using the tool

As we will now see, the animation is distributed in the sense that different stakeholders animate different agent instances of the specification. They can create initial states, consult states, give values to undetermined expressions and perform actions with a certain degree of autonomy. On the other hand, decisions regarding the flow of the animation (what is the time at the beginning, from which state are we building a transition) have to be made by a single person we call the *coordinator*. Besides these tasks performed autonomously or centrally, there are also tasks which require cooperation. In the rest of this section, we will go into the details of tasks pertaining to each this three types. The logic we will to follow is the one of the typical steps of an animation.

Creating an animation

When the *coordinator* has started the animation server, he is given the choice either to open an existing animation from the repository or to create a new one. In order to go through most of the steps, we make the assumption that creation of a new animation is selected.

As this selection is made, three tabbed folders for animation management are displayed. The first one (see *fig.5*) is dedicated to general properties of the animation. It is here that the *coordinator* associates the animation with the specification it will animate by clicking on the "Import..." button and then chosing a file among those produced by the CAT-Edit tool. Also in this folder, the animation is given a name and, possibly, a textual description.

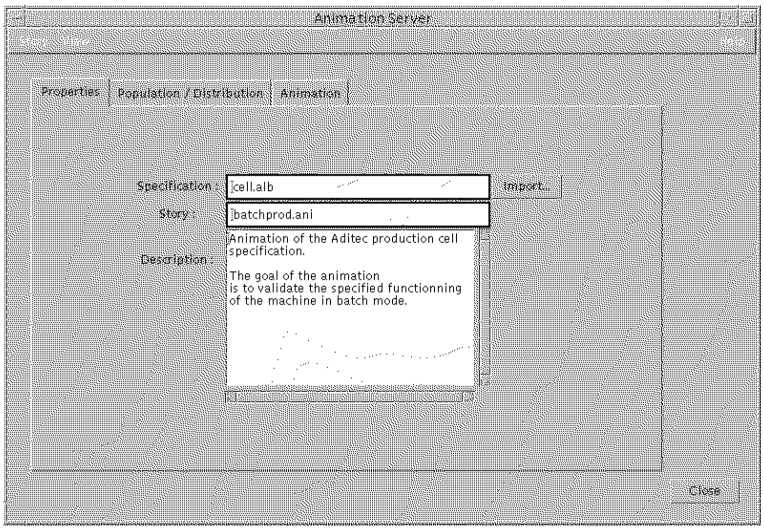


Figure 5 : Setting the general properties of an animation

The second folder (see *fig.6*) is used to fix the agents' population and distribute it to the other users. When a new animation is created, the default instantiation is of one instance per agent or society class. If an agent or society instance is an instance of a class whose number of instances is not limited to one, the user can click on the button "Duplicate" to create an additional instance. The "Delete" button has the effect of deleting an instance but it will not always work since every class has to have at least one instance. Note that duplicating/deleting a society instance will perform the operation recursively for everyone of its imbedded society or agent instances. When created, agent or society instances are given a default name (made of their class name and an identifying number) which can be changed by clicking on the "Rename..." button.

After the *coordinator* has started the animation and the other users have started animation clients, the users are able to connect to the animation server and identify themselves (not shown). The server therefore knows who are the users which are connected and, by clicking on the "Assign to user..." button when an agent or society instance is selected, the *coordinator* is able to associate the instance with the user who will animate it. (The *coordinator* is also able to reuse a configuration of the distribution from a previous animation session if the animation is not a new one. This is done through the use of the "Recover Distribution" command). Note that assigning a society instance to a user means to associate all its embedded society and agent instances to that user.

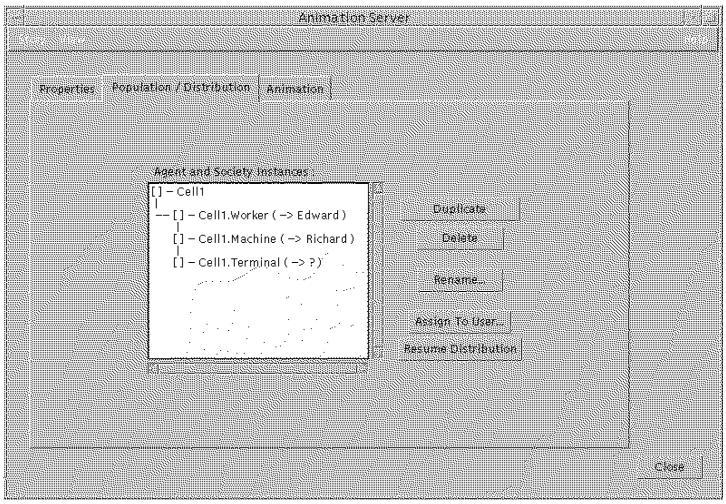


Figure 6 : Populating and distributing agent instances

Starting the animation

Once every instance has been associated with a user, it is possible to start the animation. For this, the *coordinator* goes to the third folder where the structure of the global behaviour will be represented as it will be constructed (see *fig.11* further in this section). At this moment, since the animation has just been created, it is empty. The only thing the *coordinator* can do is thus to create an initial state. He clicks on the appropriate button and is asked to enter a begin time for the state. Note that as soon as a state is created, it is not possible anymore to change the agents and societies population since, in **Albert II**, the population is considered constant in every possible behaviour.

Setting the initial state

The other users are now automatically prompted to enter the values of the state components for the initial state. Every user then sees a window per agent instance he is responsible for appearing on his screen (called Agent Manager window and entitled with the name of the agent instance). A bit of explanation is necessary at this point regarding the Agent Manager which is the main interface given to users. Every such window allows a user to browse through a series of "frames". Each frame represents a couple state transition/state that is part of the animation built so far and where a state is always associated with the state transition that created it. Therefore, every frame contains two folders : the state transition folder and the state folder. The only exception is for the initial states which are not associated with any state transition since state components are supposed to have a value at the beginning of the agents' lives. State folders are of two kinds : state editing folders (for the initial states under construction) and state consulting folders (for states, either intial or not, that have already been built). State transition folders are also of two kinds : state transition editing folders (for the transitions under construction) and state transition consulting folders (for transitions that have already been built).

If we come back to the current task of setting the initial state, we notice that every Agent Manager displays a state editing folder containing the list of the agent instance's state components and slots for entering their value.

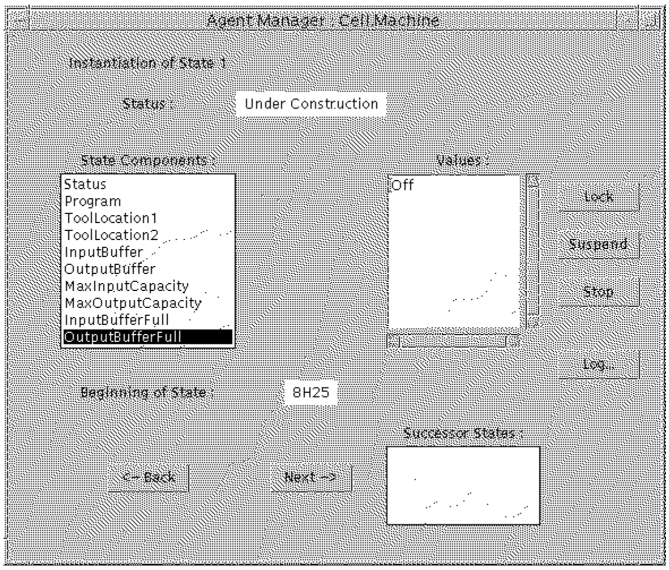


Figure 7 : Setting the initial state of an agent instance

State components which are imposed unique value by some constraints (derived components, initial valuation, state behaviour) do not have to be instantiated. Once users have instantiated all the other state components, the users "Lock" their changes. Incompatibility of the entered value for some state component with some constraint of the specification or typing errors are notified to the user (see *fig.8*) and he is prevented to lock until every state component is given an acceptable value. The "Log" button on the error box below allows to record that an unexpected behaviour happened at some point in time regarding some particular user action and violating some constraint(s) of the specification. The "Log" functionality will also be extended in order to offer the possibility of storing remarks the users want to express regarding unexpected reactions of the animator but that were not considered by the animator as not fitting with the specification. All this information will be usefull in order to correct the specification after the animation.
Besides the "Lock" functionality, the users are able to interrupt the state instantiation process or to stop it definitively it is not necessary nor possible to go further on testing the scenario.

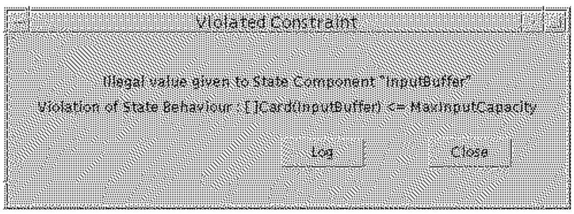


Figure 8 : Getting feedback when constraints are violated

After locking its instantiation of the intial state, the user is asked to enter the truth values of the information / perception conditions that are not defined by the specification. Values of state components that are shown to other agent instances will be passed to the animation server in order to be redispatched to the right agent instances. After this is done, Agent Managers are able to display the initial states of the agent instances they manage. Both the list of the agent instance's own state components and the list of those perceived from other agents are displayed. As they are selected, their values appear on their right.

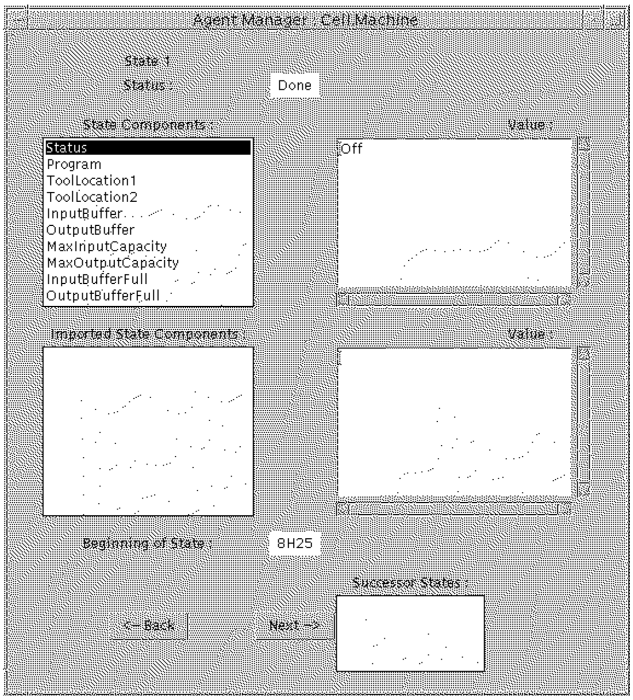


Figure 9 : Consulting the initial state of an agent instance

Together with the notification that a new intial state has been created, the animation server has received from the various animation clients the lowest time bound for the agents' obligations[5]. We will explain below how this is used.

Building the transition to the next state (server's side)

In order to build a transition from a state to another (to be created), the *coordinator* has to select the starting state and click on the "Create Change" button. Then each user is prompted to enter the closest time (wrt the time the chosen starting state begins) at which he wants (at least) one of its agent instances to perform an action. Then, for the lowest time bound of obligations and the minimum of the values just entered by the users, the animation server will compute again the minimum which will be the time for the next change and, of course, also the time for the end of the chosen starting state. But this does not mean that the open time interval associated with the starting state becomes closed. In fact, as we will see later, if the state is chosen again to be the starting point of an alternative change, the end time of the state will be different. Therefore, what informs us on the end time of a state wrt to a particular path is the begin time of the next state (or of the state transition) in the same path.

At this point, it is up to the animation clients' users to construct the state transitions of all the agent instances.

[5] The obligations of an agent instance in a certain state is a set of beginnings and ends of action occurrences and state changes that will have to take place in the future. This set is deduces by the animator wrt to the constraints of the specification, the values of the past states and the events that have taken place in the part of the behaviour that has already been built.

Building the transition to the next state (client's side)

In order to build the transition that will lead to a new state, the user has to display in the Agent Manager the frame corresponding to the transistion/state couple whose transition has been asked for creation by the *coordinator*. The frame contains a state transition editing folder (see *fig.10*) and an empty state folder (since the state cannot be computed until the change is fixed).

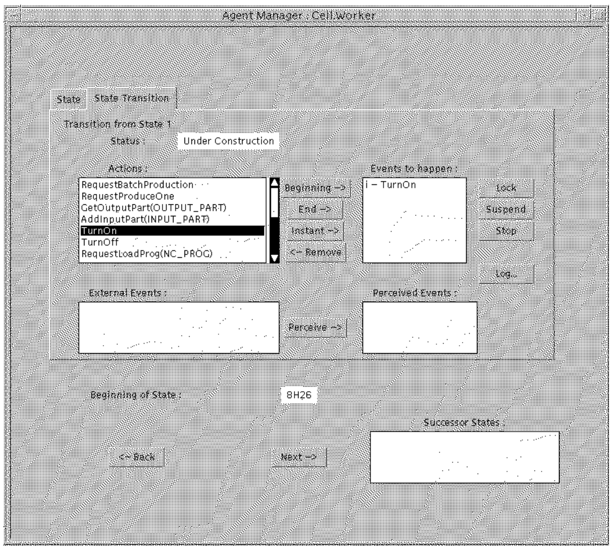


Figure 10 : Building a state transition

The user can move from a transition/state to another by using the browser-like "Back" and "Next" buttons with the exception that, since there may be alternative paths followed, there may also be several next transitions/states from which one must be selected before clicking on the "Next" button. Since, so far, we have only created an initial state, the only browsing the user is able to do is to go back to the intial state and consult it.

Now, let us come back to the state transition editing folder. The process of constructing a transition is as follows. The list of actions that have been declared for the agent are displayed as a list on the Agent Manager windows of any of their instances. Actions can be selected and, by clicking on the "Beginning", "End" or "Instant" button, one can decide to add to the current transitions 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 dialog box appears for entering them. There are also two other tasks the users have to fulfill at this point : (i) attachment of the actions to higher-level composed actions (TBD), (ii) chosing to show or not an action to other agent instances when no condition is present in the specification and (iii) chosing to perceive or not the events that are shown by the other agent instances when no condition is present in the specification.

In the same way as for the initial state (see above) messages are displayed whenever the events proposed by the user are not compatible with the specification. Similarly, the source of the contradiction (i.e. some constraints of the specification + context of the animation) is shown and logging is possible. Messages originating from the user (as opposed to the animator) can also be recorded.

When a user has finished, he has to lock his changes. No "Lock" can take place if some user has chosen to stop the animation or before all the events that had to take place were actually selected and full information concerning their occurrence was given by the user. When all the users have locked the state transitions of all the agent instances they are responsible for, the next state of every agent instance is computed. For states components whose information/perception conditions are not defined, input is requested from users. Note that the state resulting from the effects of the selected events may be an unadmissible state wrt the specification. This results in a dead-end branch which cannot be further investigated. Such states are especially marked both in the Agent Manager Windows and on the *coordinator*'s display. The other possible origin of a dead-end branch is the fact that some user has decided to stop building the current change because he was prevented to perform the events he inteded to. When one user pushes on the "Stop" button, all the other users can do is also to stop. As for initial state instantiation, interruption is also possible.

Building more state transitions and initial states

For building other transitions, all the *coordinator* has to do is to select an initial state or another state from the animation's tree structure built so far. If it is a initial state, it must be *done* (i.e. it cannot have been stopped or suspended or found unadmissible by the animator wrt to the specification´s constraints). If it is not an initial state it must be *done* also (i.e. the change that lead to it must be *done*, in the same sense as for the initial state, and the state must have been checked as admissible wrt the specification´s constraints).

If the state that is selected as starting point of a transition already has a transition following it, an alternative branch is constructed in the animation's tree. If not, it is just appended at the end of the current branch.

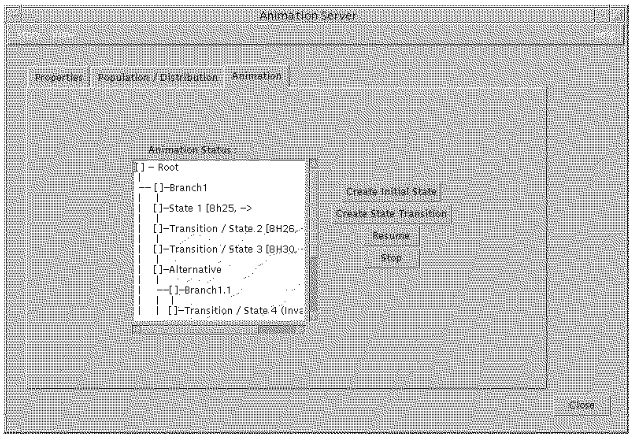


Figure 11 : Creating an alternative branch

Another point is that the number of initial states is not restricted to one. Just in the same way as we have explained it at the beginning of this section, other initial states can be created. This will cause alternative branches to be built from the top of the tree.

7. Conclusion and outlook

We have reported here on the development of the animator for specifications written in the **Albert II** language insisting on the fact that the goal pursued by the language (to be well-suited for Requirements Engineering of complex systems) made it necessary to work on a new approach to animate specifications. Work has been accomplished and is still in progress in designing a tool that deals with the most used and constrained parts of the language and whose main qualities are that (1) many undeterministic aspects are treated by 'intelligent' interactions with users, (2) it is distributed and allows cooperative work between stakeholders, (3) it allows to explore alternative behaviours and (4) it tries to keep the dialog at the level of the vocabulary used in the specification. Work has also been accomplished (but not reported in this paper) in cooperation with the CREWS team at the RWTH-Aachen in order to examine how our validation technique can be coupled with their elicitation techniques so that each (validation or elicitation) phase can take advantage of the output produced by the other in order to improve the Requirement Engineering process.
Future work is expected to take on the followings issues : (1) progressively extending the set of constructs the animator is able to deal with, (2) producing traces from animations, (3) using information about exceptional and normative scenarios in order to guide the animation and, finally, (4) see what is feasible wrt building a

domain-specific graphical layer on top of some agent manager windows in order to keep interaction even closer to users' terms.

8. Bibliography

[Alg95] Algayres, B., Lejeune, Y., Hugonnet, F., GOAL : Observing SDL Behaviors with GEODE. *Paper presented at the 7th SDL Forum*, Oslo, Norway, 26-29 September, 1995.

[Bor92] 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.

[Bre94] Breuer, P.T. and Bowen, J.P., Towards Correct Executable Semantics for Z, *in B owen J.P. and Hall J.A. (Eds.), Z User Workshop,* Cambridge, Workshops in Computing, Springer-Verlag, 185-209, 1994.

[Cha97] Chabot, F., Semantic Embedding of Albert-CORE within PVS. *Presented at the Doctoral Consortium of the Third IEEE International Symposium on Requirements Engineering (RE'97), Annapolis MD*, January 1997.

[Dub93] Dubru, F. Prototypage de Spécifications Formelles des Besoins: d'ALBERT vers OBLOG. Master thesis, Computer Science Department, University of Namur, Namur (Belgique), June 1993.

[Dub95] Dubois, E., Du Bois Ph. and Dubru, F., 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.

[DuB95] Du Bois, Ph., 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.

[DuB97] Du Bois, Ph., The Albert II Reference Manual, Technical Report, University of Namur (Belgium). Available at `http://www.info.fundp.ac.be/~phe/albert.html`

[Hay89] Hayes, I.J. and Jones, C.B., Specifications are not (necessarily) executable. *Software Engineering Journal*, 4(6):330-338, November. 1989.

[Her97] Hernalsteen, C., de Jacquier, A., Massart, Th., A Toolset for the Analysis of ET-LOTOS Specifications, *paper prensented at the Meeting on Validation and Verification of Formal Descriptions of the Fundamental Computer Science F.N.R.S. Contact Group*, Namur (Belgium), May 6, 1997.

[Hen94] Thomas A. Henzinger, Xavier Nicollin, Joseph Sifakis, and Sergio Yovine. Symbolic model checking for real-time systems. *Information and Computation* 111:193-244, 1994. A preliminary version appeared in the *Proceedings of the Seventh Annual IEEE Symposium on Logic in Computer Science* (LICS 1992), pp. 394-406.

[Hey97] Heymans, P., Some Thoughts about the Animation of Formal Specifications written in the Albert II Language. *Presented at the Doctoral Consortium of the Third IEEE International Symposium on Requirements Engineering (RE'97), Annapolis MD*, January 1997.

[Jac95] 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

[Jar95] Jarke, M. and Gallersdorfer, 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, 1995.

[Jon91] Jones, C.B., Systematic Software Development using VDM. *Prentice Hall International Series in Computer Science*, 1991.

[One92] O'Neill, G., Automatic Translation of VDM specifications into Standard ML programs. *The Computer Journal*, 35(6):623-624, 1992.

[Ras97a] Raskin, J.-F. and Schobbens P.-Y., State Clock Logic: a Decidable Real-Time Logic. *In the proceedings of Hart'97: Hybrid and Real-Time Systems, published as volume 1201 of the serie Lecture Notes in Computer Science (Sringer Verlag), pp 31-47,* Grenoble (France), March 26-28, 1997.

[Ras97b] Raskin, J.-F. and Schobbens P.-Y., Real-Time Logics : Fictitious Clock as an Abstraction of Dense Time. *In the proceedings of Tacas'97: Tools and Algorithms for the Construction and Analysis of Systems, published as volume 1217 of the serie Lecture Notes in Computer Science, Twente (Springer Verlag), pp165-182*, Twente (The Netherlands), April 2-4, 1997.

[Rol96] Rolland, C., Ben Achour, C., Cauvet, C., Ralyté, J., Sutcliffe, A., Maiden, N.A.M., Jarke, M., Haumer, P., Pohl K., Dubois, E., Heymans, P., *A Proposal for a Scenario Classification Framework (CREWS Report 96-01).* Available at `http://SunSite.Informatik.RWTH-Aachen.DE/CREWS/reports.html`

[Sid97] Siddiqi, J.I., Morrey, I.C., Roast, C.R. and Ozcan, M.B., Towards Quality Requirements via Animated Formal Specifications, *to appear in Annals of Software Engineering* , Vol 3, (1997).

[Spi92] The Z Notation : A Reference Manual. Prentice Hall International Series in Computer Science, 2nd Edition, 1992.

[Statemate] StateMate Magnum by i-Logix. Information available at `http://www.ilogix.com/`

Appendix : Specification of the production cell

SPEC CELL

BASIC TYPES

```
An NC-program is an 'atomic' concept of the specification
```

NC_PROG

```
A tool is an 'atomic' concept of the specification
```

TOOL

CONSTRUCTED TYPES

```
States originating from state-transition diagram of the machine
```

MACHINE_STATUS=ENUM[Off, StandBy, LoadingNC, ProduceOne, WaitForPart, BatchProduction]

```
There are three types of input part that fit the input buffer slots
```

INPUT_PART_TYPE=ENUM[in_type1, in_type2, in_type3]

```
There are three types of output parts that fit the output buffer slots
```

OUTPUT_PART_TYPE=ENUM[out_type1, out_type2, out_type3]

POSITIVE_INTEGER=INTEGER

```
An input part is of a certain type and has a number which makes it different from
all the other parts in the input buffer
```

INPUT_PART=CP [type : INPUT_PART_TYPE , number : POSITIVE_INTEGER]

```
An output part is of a certain type and has a number which makes it different from
all the other parts in the output buffer
```

OUTPUT_PART=CP [type : OUTPUT_PART_TYPE , number : POSITIVE_INTEGER]

```
An order can be either (1) waiting to be processed, (2) being processed by machine
set-up, (3) being processed by machine production or (4) finished.
```

ORDER_STATUS=ENUM[Waiting, SetUp, Started, Finished]

```
An order concerns a certain quantity of input parts to be transformed.

It requires a particular NC-program to be used and has a current status.
```

ORDER=CP [inputType : INPUT_PART_TYPE , quantity : POSITIVE_INTEGER , ncProg : NC_PROG , status : ORDER_STATUS]

```
States originating from state-transition diagram of the terminal
```

TERMINAL_STATUS=ENUM[StandBy, Processing]

OPERATIONS

```
The transform operation gives the type of the part the machine produces given the
type of the input, the NC-program loaded and the tools installed.
```

Transform: INPUT_PART_TYPE x NC_PROG x TOOL x TOOL
*→ OUTPUT_PART_TYPE**

SOCIETY CELL

(Machine)

(Terminal)

(Worker)

AGENT CELL.MACHINE

DECLARATION

STATE COMPONENTS

```
A machine has a current status
```

Status instance-of *MACHINE_STATUS* → *Cell.Worker*

```
A cell may have a program loaded
```

Program instance-of *NC_PROG**

```
There are two slots in the machine on which tools can be mounted. This is the
first one
```

ToolLocation1 instance-of *TOOL** → *Cell.Worker*

```
This is the second one
```

ToolLocation2 instance-of *TOOL** → *Cell.Worker*

```
The machine has an input buffer
```

InputBuffer set-of *INPUT_PART* → *Cell.Worker*

```
The machine has an output buffer
```

OutputBuffer set-of *OUTPUT_PART* → *Cell.Worker*

```
The size of the input buffer is a constant
```

**MaxInputCapacity* instance-of *POSITIVE_INTEGER*

```
The size of the output buffer is a constant
```

**MaxOutputCapacity* instance-of *POSITIVE_INTEGER*

```
The input buffer being full is determined by its current content and its maximum
capacity
```

InputBufferFull instance-of *BOOLEAN* derived-from *InputBuffer, MaxInputCapacity* → *Cell.Worker*

```
The output buffer being full is determined by its current content and its maximum
capacity
```

OutputBufferFull instance-of *BOOLEAN* derived-from *OutputBuffer, MaxOutputCapacity* → *Cell.Worker*

ACTIONS

```
In manual mode, a machine can produce an output part from
an input part if the right NC-program and the right tools are loaded.
```

ProduceOne(INPUT_PART)

```
In manual mode, a machine can fail to produce an output part from
an input part if the wrong NC-program or the wrong tools are loaded.
```

ProduceOneFail(INPUT_PART)

> In batch mode, a machine can produce an output part from
> an input part if the right NC-program and the right tools are loaded.

ProduceOneBatch(INPUT_PART)

> In batch mode, a machine can fail to produce an output part from
> an input part if the wrong NC-program or the wrong tools are loaded.

ProduceOneBatchFail(INPUT_PART)

> In batch mode, if at least an input part is present in the
> input buffer and if it is not already processing, the machine
> has to process an input part (see triggerings). But the result
> depends on the NC-program and on the tools. Therefore,
> the production operation will either succeed or fail
> (see compositions –→ alternative)

TryProduceOneBatch

> Batch mode stops if production has been impossible for
> 60 seconds (because no input part was available or
> because the outputbuffer was full)

**StopBatchProduction*

> When asked to by the worker, the machines loads the requested NC-program

LoadProg(NC_PROG)

> Composed action #1 (see compositions)

comp1

> Composed action #2 (see compositions)

comp2

BASIC CONSTRAINTS

DERIVED COMPONENTS

> The input buffer being full is determined by its content being
> equal to its capacity.

$InputBufferFull \triangleq (Card(InputBuffer) = MaxInputBufferCapacity)$

> The output buffer being full is determined by its content being
> equal to its capacity.

$OutputBufferFull \triangleq (Card(OutputBuffer) = MaxOutputBufferCapacity)$

INITIAL VALUATION

> Initially, the machine is assumed to be off, with no tool
> mounted and no NC-program in memory.

ToolLocation1 = UNDEF

ToolLocation2 = UNDEF

Status = Off

Prog = UNDEF

DECLARATIVE CONSTRAINTS

STATE BEHAVIOR

`The input buffer's content can never be greater than its capacity`

[ ] Card(InputBuffer) ≤ MaxInputCapacity

`The output buffer's content can never be greater than its capacity`

[ ] Card(OutputBuffer) ≤ MaxOutputCapacity

ACTION COMPOSITION

`A (perceived) request to load an NC-program is immediately processed`

comp1 ↔ w.RequestLoadProg(ncp) <0 sec> LoadProg(ncp)

`A (perceived) request to produce a part is immediately processed`

comp2 ↔ w.RequestProduceOne <0 sec> ProduceOne(_)

`Producing a part in batch mode can either succeed of fail`

TryProduceOneBatch ↔ ProduceOneBatch(_) ⊕ ProduceOneBatchFail(_)

`Occurrences of these actions are restricted to compositions.`

*{comp1, comp2, w.RequestLoadProg, LoadProg,
w.RequestProduceOne, ProduceOne,
TryProduceOneBatch, ProduceOneBatch,
ProduceOneBatchFail}*

ACTION DURATION

`Producing a part takes at least 70 seconds`

ProduceOne(_) ≥ 70 sec

`Producing a part takes at most 85 seconds`

ProduceOne(_) ≤ 85 sec

`Producing a junk part takes at least 70 seconds`

ProduceOneFail(_) ≥ 70 sec

`Producing a junk part takes at most 85 seconds`

ProduceOneFail(_) ≤ 85 sec

`Producing a part in batch mode takes at least 70 seconds`

ProduceOneBatch(_) ≥ 70 sec

`Producing a part in batch mode takes at most 85 seconds`

ProduceOneBatch(_) ≤ 85 sec

`Producing a junk part in batch mode takes at least 70 seconds`

ProduceOneBatchFail(_) ≥ 70 sec

`Producing a junk part in batch mode takes at most 85 seconds`

ProduceOneBatchFail(_) ≤ 85 sec

`Loading an NC-program takes 15 seconds`

LoadProg(_) = 15 sec

OPERATIONAL CONSTRAINTS

PRECONDITIONS

> Producing a part can only take place if there is an admissible part in the input buffer

ProduceOne(p) : $p \in InputBuffer \wedge transform(type(p), Program, ToolLocation1, ToolLocation2) \neq UNDEF$

> Producing a junk part can only take place if there is an unadmissible part in the input buffer

ProduceOneFail(p) : $p \in InputBuffer \wedge transform(type(p), Program, ToolLocation1, ToolLocation2) = UNDEF$

> Producing a part in batch mode can only take place if there is an admissible part in the input buffer, if the output buffer is not full and if the machine is waiting for a part

ProduceOneBatch(p) : $Status = WaitForPart \wedge p \in InputBuffer \wedge \neg OutputBufferFull \wedge transform(type(p), Program, ToolLocation1, ToolLocation2) \neq UNDEF$

> Producing a junk part in batch mode can only take place if there is an unadmissible part in the input buffer, if the output buffer is not full and if the machine is waiting for a part

ProduceOneBatchFail(p) : $Status = WaitForPart \wedge p \in InputBuffer \wedge \neg Empty(InputBuffer) \wedge \neg OutputBufferFull \wedge transform(type(p), Program, ToolLocation1, ToolLocation2) = UNDEF$

> Batch production stops only when the machine has been waiting for a part during 60 seconds

StopBatchProduction : $Lasted_{60\ secs}\ Status = WaitForPart$

EFFECT OF ACTIONS

> Producing a part has the effect of temporarily switching the machine's status (see state-transition diagram). It also removes a part from the input buffer and add a part to the output buffer.

ProduceOne(p1)
<u>with</u> $type(p2) = transform(type(p1), Program, ToolLocation1, ToolLocation2)$
$\wedge \neg \exists p3\ (p3 \in OutputBuffer \wedge number(p3) = number(p2))$:
InputBuffer := Remove(p1, InputBuffer) ;
Status := ProduceOne ;
[]
Status := StandBy ;
OutPutBuffer := Add(p2,OutputBuffer)

> Producing a junk part has the effect of temporarily switching the machine's status (see state-transition diagram). It also removes a part from the input buffer.

ProduceOneFail(p1) :
Status := ProduceOne ;
InputBuffer := Remove(p1, InputBuffer)
[]
Status := StandBy

> Producing a part in batch mode has the effect of temporarily switching the machine's status (see state-transition diagram). It also removes a part from the input buffer and add a part to the output buffer.

ProduceOneBatch(p1)
with *type(p2) = transform(type(p1),Program,ToolLocation1,ToolLocation2)*
∧ ¬ ∃ p3 (p3 ∈ OutputBuffer ∧ number(p3) = number(p2)):
Status := BatchProduction ;
InputBuffer := Remove(p1, InputBuffer) ;
[ ]
Status := WaitForPart ;
OutPutBuffer := Add(p2,OutputBuffer)

> Producing a junk part in batch mode has the effect of temporarily switching the machine's status (see state-transition diagram).

ProduceOneBatchFail(p1) : Status := BatchProduction ;
[ ]
Status := WaitForPart ;
InputBuffer := Remove(p1, InputBuffer)

StopBatchProduction : [ ] Status := StandBy

> Loading an NC-program has the effect of temporarily switching the machine's status (see state-transition diagram) and putting the requested program in the machine's memory.

LoadProg(ncp) : Status := LoadingNC ;
[ ]
Status := StandBy ;
Program := npc

> Perceiving a request for batch production has the effect of putting the machine in the status of waiting for a part(see state-transition diagram).

w.RequestBatchProduction :
[ ]
Status := WaitForPart

> When the worker gets an output part from the machine, this has the effect of removing a part from the machine's output buffer

w.GetOutputPart(p) :
[ ]
OutputBuffer := Remove(p,OutputBuffer)

> When the worker adds an input part from the machine, this has the effect of adding a part to the machine's input buffer

w.AddInputPart(p) :
[ ]
InputBuffer := Add(p,InputBuffer)

> When the worker turns on the machine, this has the effect of putting it in standby status

w.TurnOn :
[ ]
Status := StandBy

> When the worker turns off the machine, this has the effect of putting it in off status

w.TurnOff :
[ ]
Status := Off ;
Prog := UNDEF

```
When the worker installs a tool in the first location, this has the effect of
putting the tool in the location !
```

w.InstallTool1(t) :
[ ]
ToolLocation1 := t

```
When the worker installs a tool in the second location, this has the effect of
putting the tool in the location !
```

w.InstallTool2(t) :
[ ]
ToolLocation2 := t

```
When the worker uninstalls a tool from the first location, this has the effect of
giving an undefined value to the content of the location
```

w.UninstallTool1 :
[ ]
ToolLocation1 := UNDEF

```
When the worker uninstalls a tool from the second location, this has the effect of
giving an undefined value to the content of the location
```

w.UninstallTool2 :
[ ]
ToolLocation2 := UNDEF

TRIGGERINGS

```
Trying to produce a part in batch has to happen when the machine waits for a part,
when its input buffer is not empty and when its output buffer is not full.
Triggering of the action is then instantaneous.
```

$Status = WaitForPart \wedge \neg Empty(InputBuffer) \wedge \neg OutputBufferFull \,/\, 0\ secs \rightarrow TryProduceOneBatch$

```
Stopping batch production has to take place when the machine has been waiting for
a part for exactly 60 seconds
```

$Lasted_{60\ secs}\ Status = WaitForPart \,/\, 0\ secs \rightarrow StopBatchProduction$

COOPERATION CONSTRAINTS

ACTION PERCEPTION

```
For a request to produce in manual mode to be processed by
the machine, some conditions must be met : (1) the machine
must be in stand-by status, (2) tools have to be mounted in
the 2 slots, (3) there must be places left in the output buffer
and (4) an NC-program must have been loaded since at least
3 minutes.
```

$\mathcal{XK}(w.RequestProduceOne \,/$
$Status = StandBy$
$\wedge\ ToolLocation1 \neq UNDEF$
$\wedge\ ToolLocation2 \neq UNDEF$
$\wedge\ \neg\ OutputBufferFull$
$\wedge\ \neg\ Empty(InputBuffer)$
$\wedge\ (Lasted_{3\ min}\ (\ Program = ncp))$
$\wedge\ ncp \neq UNDEF)$

```
For a request to produce in batch mode to be processed by
the machine, some conditions must be met : (1) the machine
must be in stand-by status, (2) tools have to be mounted in
the 2 slots, (3) there must be places left in the output buffer
and (4) an NC-program must have been loaded since at least
3 minutes.
```

$\mathcal{XK}$(*w.RequestBatchProduction /*
Status = StandBy
$\wedge$ *ToolLocation1* $\neq$ *UNDEF*
$\wedge$ *ToolLocation2* $\neq$ *UNDEF*
$\wedge \neg$ *OutputBufferFull*
$\wedge \neg$ *Empty(InputBuffer)*
$\wedge$ *(Lasted*$_{3\ min}$ *(Program = ncp))*
$\wedge$ *ncp* $\neq$ *UNDEF)*

```
When the worker gets an output part, the machine always undergoes its effect
```

$\mathcal{K}$(*w.GetOutputPart(_) / TRUE)*

```
When the worker adds an input part, the machine always undergoes its effect
```

$\mathcal{K}$(*w.AddInputPart(_) / TRUE)*

```
The worker can only turn on the machine when it is off.
```

$\mathcal{XK}$(*w.TurnOn / Status = Off)*

```
The worker can only turn off the machine when it is on.
```

$\mathcal{XK}$(*w.TurnOff / Status = StandBy)*

```
The worker can only request the machine to load an NC-program when it is in
standby status.
```

$\mathcal{XK}$(*w.RequestLoadProg(_) / Status = StandBy)*

```
Installing a tool is always made visible to the machine
```

$\mathcal{K}$(*w.InstallTool1 / TRUE)*

```
Installing a tool is always made visible to the machine
```

$\mathcal{K}$(*w.InstallTool2 / TRUE)*

```
Uninstalling a tool is always made visible to the machine
```

$\mathcal{K}$(*w.UninstallTool1 / TRUE)*

```
Uninstalling a tool is always made visible to the machine
```

$\mathcal{K}$(*w.UninstallTool2 / TRUE)*

STATE INFORMATION

```
The content of a tool location is always made visible to the worker
```

$\mathcal{K}$(*ToolLocation1.w / TRUE)*

```
The content of a tool location is always made visible to the worker
```

$\mathcal{K}$(*ToolLocation2.w/ TRUE)*

```
The status of the machine is always made visible to the worker
```

$\mathcal{K}$(*Status.w / TRUE)*

```
The content of the input buffer is always made visible to the worker
```

𝒦(InputBuffer.w / TRUE)

The content of the output buffer is always made visible to the worker

𝒦(OutputBuffer.w / TRUE)

The content of the input buffer being full is always made visible to the worker

𝒦(InputBufferFull.w / TRUE)

The content of the output buffer being full is always made visible to the worker

𝒦(OutputBufferFull.w / TRUE)

AGENT CELL.TERMINAL

DECLARATION

STATE COMPONENTS

The orders that have to be processed are initially
put into a table. They are indexed with positive integers.
The first order to be processed is the one indexed with
number 1 (initial value of CurrentOrder). The following
orders are processed (increasing the value of CurrentOrder
by 1) until an order with value UNDEF in encountered.

Orders table-of *ORDER* indexed-by *POSITIVE_INTEGER* → *Cell.Worker*

Denotes the index value of the order the worker claims
to be processing at a certain time.

CurrentOrder instance-of *POSITIVE_INTEGER* → *Cell.Worker*

This variable tells if at a certain time the terminal displays
the current order.

ShowCurrentOrder instance-of *BOOLEAN*

see state-transition diagram

Status instance-of *TERMINAL_STATUS* → *Cell.Worker*

ACTIONS

The terminal can obey to a request from the worker to
show him the current order

ShowOrder

The terminal can obey to a request from the worker to
record the fact that he has started the set-up for the current
order to be processed.

RecordSetUpBegin

The terminal can obey to a request from the worker to
record the fact that he has started production of the current
order.

RecordWorkingBegin

The terminal can obey to a request from the worker to
record the fact that he has finished production of the current

```
order.
```

RecordWorkingEnd

```
Composed action #1 (see compositions)
```

comp1

```
Composed action #2 (see compositions)
```

comp2

```
Composed action #3 (see compositions)
```

comp3

```
Composed action #4 (see compositions)
```

comp4

BASIC CONSTRAINTS

INITIAL VALUATION

```
Initially, the worker is asked to process the first order.
```

CurrentOrder = 1

```
The worker does not have automatically acces to the
current order, he has to request it.
```

ShowCurrentOrder = FALSE

```
Initially, the terminal does not perform any operation.
```

Status = StandBy

DECLARATIVE CONSTRAINTS

ACTION COMPOSITION

```
A (perceived) request to display the current order is immediately processed
```

comp1 ↔ w.RequestOrder <0 sec> ShowOrder

```
A (perceived) request to record that processing of the current order is in its
set-up phase is immediately processed
```

comp2 ↔ w.RequestSetUpBegin <0 sec> RecordSetUpBegin

```
A (perceived) request to record that processing of the current order is in its
working phase is immediately processed
```

comp3 ↔ w.RequestWorkingBegin <0 sec> RecordWorkingBegin

```
A (perceived) request to record that processing of the current order is finished
is immediately processed
```

comp4 ↔ w.RequestWorkingEnd <0 sec> RecordWorkingEnd

```
Occurrences of these actions are restricted to compositions.
```

{comp1, comp2, comp3, comp4,
w.RequestOrder, ShowOrder,
w.RequestSetUpBegin, RecordSetUpBegin,
w.RequestWorkingBegin, RecordWorkingBegin,
w.RequestWorkingEnd, RecordWorkingEnd}

OPERATIONAL CONSTRAINTS

EFFECT OF ACTIONS

```
Showing the current order temporarily puts the terminal in 'processing' status and
makes the current order visible.
```

ShowOrder : Status := Processing ;
[ ]
ShowCurrentOrder := TRUE ;
Status := StandBy

```
Recording that the processing of the current order is being set-up temporarily
puts the terminal in 'processing' status, makes the current order invisible and
changes its status to 'sset-up' the shopfloor data collection.
```

RecordSetUpBegin :
Status:= Processing ;
ShowCurrentOrder := FALSE ;
[ ]
Orders[CurrentOrder] :=
<inputType(Orders[CurrentOrder]),quantity(Orders[CurrentOrder]),ncProg(Orders [CurrentOrder]), SetUp>;
Status := StandBy;

ShowCurrentOrder := FALSE ;

```
Recording that the processing of the current order has begun temporarily puts the
terminal in 'processing' status, makes the current order invisible and changes its
status to 'started' the shopfloor data collection
```

RecordWorkingBegin :
Status:= Processing ;
ShowCurrentOrder := FALSE ;
[ ]
Orders[CurrentOrder] :=
<inputType(Orders[CurrentOrder]),quantity(Orders[CurrentOrder]),ncProg(Orders [CurrentOrder]), Started>;
Status := StandBy;

ShowCurrentOrder := FALSE ;

```
Recording that the processing of the current order has finished temporarily puts
the termianl in 'processing' status, makes the current order invisible and changes
its status to 'finished' the shopfloor data collection
```

RecordWorkingEnd :
Status:= Processing ;
ShowCurrentOrder := FALSE ;
[ ]
Orders[CurrentOrder] :=
<inputType(Orders[CurrentOrder]),quantity(Orders[CurrentOrder]),ncProg(Orders [CurrentOrder]), Finished>;
CurrentOrder := CurrentOrder+1 ;
Status := StandBy

COOPERATION CONSTRAINTS

ACTION PERCEPTION

```
A request to show the current order is always perceived from the worker
```

$\mathcal{K}$(w.RequestOrder / TRUE)

A request to record that processing of the current order is being set up is always perceived

$\mathcal{K}$(w.ReportSetUpBegin / TRUE)

A request to record that processing of the current order has started is always perceived

$\mathcal{K}$(w.ReportWorkingBegin / TRUE)

A request to record that processing of the current order has finished is always perceived

$\mathcal{K}$(w.ReportWorkingEnd / TRUE)

AGENT CELL.WORKER

DECLARATION

ACTIONS

The worker can request the machine to perform batch production

**RequestBatchProduction → Cell.Machine*

The worker can request the machine to produce a part

**RequestProduceOne → Cell.Machine*

The worker can get a part from the machine's output stock

**GetOutputPart(OUTPUT_PART) → Cell.Machine*

The worker can add a part to the machine's input stock

**AddInputPart(INPUT_PART) → Cell.Machine*

The worker can turn on the machine

**TurnOn → Cell.Machine*

The worker can turn off the machine

**TurnOff → Cell.Machine*

The worker can request the machine to load an NC-program

**RequestLoadProg(NC_PROG) → Cell.Machine*

The worker install a tool in the first appropriate slot of the machine

InstallTool1(TOOL) → Cell.Machine

The worker install a tool in the second appropriate slot of the machine

InstallTool2(TOOL) → Cell.Machine

The worker uninstall a tool from the first appropriate slot of the machine

UninstallTool1 → Cell.Machine

The worker uninstall a tool from the second appropriate slot of the machine

UninstallTool2 → Cell.Machine

The worker can request the terminal to display the current order

**RequestOrder → Cell.Terminal*

The worker can request the terminal to record that processing the current order is being set up

$*ReportSetUpBegin \rightarrow Cell.Terminal$

> The worker can request the terminal to record that processing the current order has started

$*ReportWorkingBegin \rightarrow Cell.Terminal$

> The worker can request the terminal to record that processing the current order has finished

$*ReportWorkingEnd \rightarrow Cell.Terminal$

OPERATIONAL CONSTRAINTS

PRECONDITIONS

> Getting an output part requires that there is (at least) a part in the output buffer

$GetOutputPart(p) : p \in m.OutputBuffer$

> Adding an input part requires that the part being added does not already belong to the input stock and that the input buffer is not full

$AddInputPart(p) : \neg\, p \in m.InputBuffer \wedge \neg\, m.InputBufferFull$

> Installing a tool in the first slot requires that the machine is either loading an nc-program, off or in standby. It alos requires that the first tool location is empty.

$InstallTool1(\_) :(Cell.Machine.Status = LoadingNC \vee Cell.Machine.Status = StandBy \vee Cell.Machine.Status = Off) \wedge ToolLocation1 = UNDEF$

> Installing a tool in the second slot requires that the machine is either loading an nc-program, off or in standby. It alos requires that the second tool location is empty.

$InstallTool2(\_) :(Cell.Machine.Status = LoadingNC \vee Cell.Machine.Status = StandBy \vee Cell.Machine.Status = Off) \wedge ToolLocation2 = UNDEF$

> Uninstalling a tool from the first slot requires that the machine is either loading an nc-program, off or in standby. It alos requires that the first tool location is not empty.

$UninstallTool1(\_) :(Cell.Machine.Status = LoadingNC \vee Cell.Machine.Status = StandBy \vee Cell.Machine.Status = Off) \wedge \neg\, ToolLocation1 = UNDEF$

> Uninstalling a tool from the second slot requires that the machine is either loading an nc-program, off or in standby. It alos requires that the second tool location is not empty.

$UninstallTool2(\_) :(Cell.Machine.Status = LoadingNC \vee Cell.Machine.Status = StandBy \vee Cell.Machine.Status = Off) \wedge \neg\, ToolLocation2 = UNDEF$

> Requesting the current order to be displayed by the terminal requires that it is not processing.

$RequestOrder : Cell.Terminal.Status \neq Processing$

> Requesting to record that processing of the current order is being set up requires that the terminal is not processing, that there is a current order and that it is in 'waiting' status.

$ReportSetUpBegin : Cell.Terminal.Status \neq Processing \wedge Cell.Terminal.Order[Cell.Terminal.CurrentOrder] \neq UNDEF \wedge status(Cell.Terminal.Order[Cell.Terminal.CurrentOrder]) = Waiting$

> Requesting to record that processing of the current order has begun requires that the terminal is not processing, that there is a current order and that it is in 'setup' status.

ReportWorkingBegin : Cell.Terminal.Status ≠ Processing ∧ Cell.Terminal.Order[Cell.Terminal.CurrentOrder]≠ UNDEF ∧ status(Cell.Terminal.Order[Cell.Terminal.CurrentOrder]) = SetUp

> Requesting to record that processing of the current order has finished requires that the terminal is not processing, that there is a current order and that it is in 'started' status.

ReportWorkingEnd : Cell.Terminal.Status ≠ Processing ∧ Cell.Terminal.Order[Cell.Terminal.CurrentOrder]≠ UNDEF ∧ status(Cell.Terminal.Order[Cell.Terminal.CurrentOrder]) = Started

COOPERATION CONSTRAINTS

STATE PERCEPTION

> The content of the machine's first tool location is always perceived

𝒦(m.ToolLocation1 / TRUE)

> The content of the machine's second tool location is always perceived

𝒦(m.ToolLocation2 / TRUE)

> The machine's status is always perceived

𝒦(m.Status / TRUE)

> The content of the machine's input buffer is always perceived

𝒦(m.InputBuffer / TRUE)

> The content of the machine's output buffer is always perceived

𝒦(m.OutputBuffer / TRUE)

> The machine's input buffer being full is always perceived

𝒦(m.InputBufferFull / TRUE)

> The machine's output buffer being full is always perceived

𝒦(m.OutputBufferFull / TRUE)

ACTION INFORMATION

> Requests to do batch processing are always made visible to the machine

𝒦(RequestBatchProduction.m /TRUE)

> Requests to produce a part are always made visible to the machine

𝒦(RequestProduceOne.m / TRUE)

> Getting a part from the output stock is always made visible to the machine

𝒦(GetOutputPart(_).m / TRUE)

> Adding a part to the input stock is always made visible to the machine

𝒦(AddInputPart(_).m / TRUE)

> Turning it off is always made visible to the machine

𝒦(TurnOff.m / TRUE)

> Turning it on is always made visible to the machine

𝒦(TurnOn.m / TRUE)

> Requests to load an NC-program are always made visible to the machine

$\mathcal{K}$(*RequestLoadProg(_).m / TRUE*)

```
Installing a tool in the first slot is always made visible to the machine
```

$\mathcal{K}$(*InstallTool1(_).m / TRUE*)

```
Installing a tool in the second slot is always made visible to the machine
```

$\mathcal{K}$(*InstallTool2(_).m / TRUE*)

```
Uninstalling a tool from the first slot is always made visible to the machine
```

$\mathcal{K}$(*UninstallTool1(_).m / TRUE*)

```
Uninstalling a tool from the second slot is always made visible to the machine
```

$\mathcal{K}$(*UninstallTool2(_).m / TRUE*)

```
Requesting the current order to be displayed by the terminal is always made
visible to the terminal
```

$\mathcal{K}$(*RequestOrder.t/ TRUE*)

```
Requesting to record that processing of the current order is being set up is
always made visible to the terminal
```

$\mathcal{K}$(*ReportSetUpBegin.t/ TRUE*)

```
Requesting to record that processing of the current order has begun is always made
visible to the terminal
```

$\mathcal{K}$(*ReportWorkingBegin.t/ TRUE*)

```
Requesting to record that processing of the current order has finished is always
made visible to the terminal
```

$\mathcal{K}$(*ReportWorkingEnd.t/ TRUE*)