Institut d'Informatique

Rue Grandgagnage, 21
B-5000 Namur
BELGIUM

Agent-oriented Requirements Engineering: A Case Study using the ALBERT Language

Eric Dubois, Philippe Du Bois,
Frédéric Dubru and Michaël Petit

RP-94-001 September 1994

AGENT-ORIENTED REQUIREMENTS ENGINEERING A CASE STUDY USING THE ALBERT LANGUAGE*

Eric Dubois, Philippe Du Bois, Frédéric Dubru and Michaël Petit
Institut d'Informatique, University of Namur
Rue Grandgagnage, 21 – B-5000 Namur (Belgium)
Fax: +32 81 72.49.67 – Phone: +32 81 72.49.80
Email: {edu, pdu, fdb, mpe} @ info.fundp.ac.be

Abstract

Requirements Engineering is now widely recognised as a key issue in the design and implementation of information systems. Recent research trends plead for the use of formal (i.e. mathematical/logical) languages at this stage of analysis in order to cope with the classical difficulties arisen from the interactions with customers. In this paper, we introduce the Albert language, a formal language based on the concept of 'agent' (seen as a specialization of the 'object' concept) in terms of which one may express real-time requirements as well as 'non-functional' requirements related to the reliability and security aspects of agents. The language is fully presented and illustrated through the handling of a non trivial computer-integrated manufacturing case study. Finally, some methodological guidelines are proposed for providing some help to the analyst in the incremental elaboration of a complex requirements document and a set of tools developped around Albert is presented.

1 Introduction

In this paper, our aim is at reporting some preliminary results on the introduction of *formal* methods and *object-orientation* paradigm in the development of distributed real-time systems. More specifically, in this paper, we address the *requirements phase* of the system lifecycle where the customers wishes have to be captured and analysed.

Requirements Engineering (or Requirements Analysis) is now widely recognised as a critical activity in the context of information system and software development. Besides classical semi-formal languages that are in use for the modelling and the analysis of requirements (like, e.g., SADT, MERISE, SSADM, etc), two recent trends can be identified in some more recent languages designed for that purpose : on the one hand, languages based on mathematical/logical theoretical grounds are now emerging (see e.g. CIM [Bubenko80], RML [GBM86], GIST [Feather87], MAL [FP87], ERAE [DHR91] and TELOS [MBJK90]); on the other hand, the object-orientation paradigm, originally used at the design and implementation levels, is now

*To appear in the proceedings of the Fourth International Working Conference on Dynamic Modelling and Information Systems – DYNMOD'94, Noordwijkerhout (the Netherlands), September 28-30, 1994.

being investigated and promoted at the requirements level (see e.g. [CY91, SM88]). Along these directions, within the framework of the Esprit ICARUS project, we have developed the Albert (ALBERT, Agent-oriented Language for Building and Eliciting Real-Time requirements) language which aims at the expression of (i) statements about real-world entities, (ii) performances requirements and (iii) visibility and reliability requirements. More precisely, the language is based on a specialization of the *object* concept with the following features [DDP93b]:

- An agent must not be considered in isolation but through its relationships with other agents. The society of agents is involved in a joint problem solving characterized by some **goals** to be achieved within the system.
- An agent is characterized by an internal **state** (by analogy with human agents, the internal state is sometimes referred as 'mental state'). The state contains the knowledge of the agent about the external world. This knowledge is represented in terms of data which correspond to the *information* level.
- An agent is responsible for **actions**. They are characterized through the changes that they bring to its internal state. They support the description of the *activities* level. Moreover, there are restrictions on the agent capabilities for performing actions in given situations. These supports the description of the *behavioural* level.
- An agent will not act autonomously but will require **cooperation** (and thereby communication) with other agents. These aspects cover part of the *resources* level by making possible to describe commitments given by an agent to another one in terms of actions to be performed or required accesses to some piece of information.

This paper aims at an in-depth presentation of the Albert language and of its use in an industrial context. In Sect.2, we briefly recall some specificities of the Requirements Engineering activity and we precise the specific charcteristics of the designed language. Then, in Sect.3, the language itself is fully presented through the handling of two case studies related to Computer Integrated Manufacturing applications. On top of a language, some methodological guidance is definitively required for an incremental elaboration of a complex requirements document. In Sect.4, we briefly describe three general strategies that we have experimented. Finally, Sect.5 concludes with a brief report on the development of supporting tools for the Albert language and on future researches perspectives.

2 Requirements Engineering

In this section, we briefly outline the specificity of the requirements engineering activity (RE). We conclude with the motivations underlying the Albert language.

2.1 The Specificity of the Requirements Engineering activity

Since a few years, there is a large consensus in the Software and Information System (IS) communities, on the necessity to make distinct three different documents in the IS development

lifecycle at the Design Engineering (DE) level (as depicted on bottom of Fig.1): (i) the **software specification** product where are precisely defined the behavioural properties (the *What*) expected from the final code, (ii) the **design** product where is presented the logical design of a modules *architecture* describing an abstract solution for the problem described by the specification and (iii) the final **code** where is presented the physical design of programs.

The IS lifecycle described above covers well the activities leading from specifications to programs, but does not address the activity of obtaining the initial specifications, i.e. Requirements Engineering (RE). This activity starts from informal wishes expressed by one or several customers and elaborate a so-called *requirements document* where the system to be developed is defined in a precise way. In the context of the development of information systems, the requirements document should include not only specifications on the software piece to be implemented but also on the *environment* (made of devices, hardware, humans, etc) around this software as well as the interactions taking place between both. Such systems are sometimes referred as *composite* systems [Feather87] or Open Information Systems [Hewitt91].

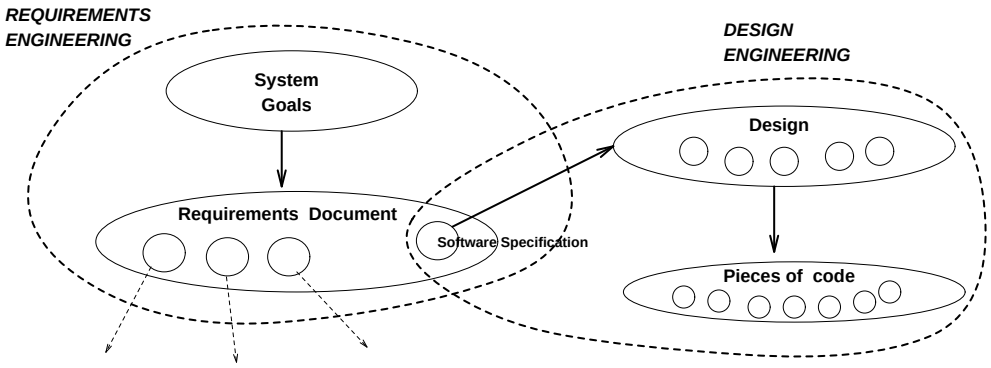


Figure 1: I.S. Development Activities

On the top of the Fig.1, two important products of the RE phase are made distinct.

1. the **goals** of the system to be developed. For example, in the context of a CIM application, one may think to a goal as "when a production order is issued, a bolt has to be produced within 10 minutes",

2. the **requirements document** where are described the different components of the system as well as the set of requirements attached to each individual component. For example, in the CIM example, the requirements document will identify the different components (like, e.g., a Controller, a Lathe machine, a Robot) and the way they will interact together for producing the requested bolt within the appropriate delay.

The RE products presented above are quite similar to the DE products presented before. One may consider that, on the one hand, the *Goals* product (at the RE level) is analogous to the *Specification* product at the DE level and, on the other hand, the *Requirements document* product (at the RE level) is analogous to the *Design* product (at the DE level).

To conclude and summarise Fig.1,

- the requirements document is specifying a *solution* for the *problem* expressed in terms of goals associated with the *whole system*;

- the code is describing a *solution* for the *problem* expressed in terms of a specification associated with the *software system*.

One may imagine a similar implementation lifecycle for the other components of the requirements document (e.g. the implementation of a robot). This aspect is not depicted on the figure.

2.2 About the Design of the Language

Basically, the design of the language has been done according to three directions : agent-orientation, formality and expressiveness.

2.2.1 An "Agent-Oriented" Paradigm

As a first approximation, "agent-oriented" can be understood as "object-oriented". This means that the language meets the main OO principles, namely encapsulation of data structures and actions on them in one unit called here an *agent*. The word "agent" has been preferred to the one of object for the following reasons: as the requirements document is intended to describe the *contractual* behaviour attached with the different components of a system, we feel that the word "agent" is well appropriate for denoting components having responsibilities and perceptions within the system.

Since the requirements document is a central document resulting from an agreement between *customers* expressing their wishes about admissible agent behaviours and *designers* in charge of implementing them, we feel important to use a rigorous specification language supporting a "natural" mapping between all kinds of things of interest and the various language concepts being available. With respect to that aspect, usual OO specification languages tend to propose a too *operational* style (based, in most cases, on Petri nets). In  , we have tried to support a more declarative style for expressing requirements.

2.2.2 Formality

 is formal. It has a formal semantics giving a precise meaning to all specifications written in this language. The existence of these rigorous rules of interpretation provides support to the analyst in its modelling task by giving him/her some hints in the application of the language constructs. Besides, rules of deductive inference are also available for supporting automatic reasoning on the written specification in order to help the analyst to analyse it (e.g., for discovering inconsistencies and incompletenesses) and to validate it (e.g., through its animation).

2.3 Expressiveness

Several formal specification languages have been proposed for the purpose of describing software components (e.g., Z, VDM, Larch). The  language goes along these lines but

with a greater expressiveness, i.e. a large variety of requirements can be easily encoded without the introduction of any over-specification. This variety is illustrated in the rest of this section.

Information Structures Modelling. The ERA (Entity-Relation-Attribute) model is now considered as standard for the data modelling aspect. However, in the case where the number of data is large and/or where the complexity of data is important (e.g., the plan of an aircraft), it is recognised that the ERA model suffers from a lack of basic structuring mechanisms (like, specialization, aggregation and classification mechanisms). Albert offers the possibility of mapping data structures in an ERA-like way but makes also possible to support more structured descriptions, expressed in terms of predefined mechanisms used for the elaboration of complex *data types*.

Historical Data Management. Traditional RE approaches rely on a "snapshot" view of the information state. This information state, at a time *t*, mirrors the real-world state of information at this time but also records information related to the past, which entails the risk of introducing over-specifications according to the way these historical information are represented. For example, in an ERA context, the mapping of a requirement like "a book can only be borrowed once by a reader" will lead to the introduction of a somewhat artificial data structure for keeping trace of the "borrowed books".

In the Albert language, we have chosen to represent, in an implicit way, the historical sequence of information states in order to make possible the expression of a variety of requirements referring information at different moments of time.

Effects of Actions. In some traditional approaches, actions, which alter information states, are described in an algorithmic way (pseudo-code, decision tables, etc). In contrast, Albert adopts a "functional" characterization style where the effect of an action is expressed in terms of a mathematical relationship among two successive information states.

Causalities among Actions. Actions triggering is usually ensured through ECA (Event-Condition-Action) rules, i.e., at any moment, when an event occurs and if a condition on the current information state is met, then the action happens. Thereby, the supported specification style is an *operational* one since, at any moment, we need to evaluate the set of occurred events and the set of fulfilled conditions in order to determine the set of candidate actions.

In many cases, this style may lead to the introduction of over-specifications at the information state level in order to keep track of actions occurrences and of specific causalities among them. Consider a statement in a *lift management system* like "the push on the button at a floor is followed by the visit of the lift at this floor". The use of an operational specification style results in a mapping of this statement where an extra piece of state information (like "pending requests") will be introduced for recording the happening of events.

In Albert , like in some other recent specification languages, the reactive nature of a system is expressed without any over-specification, in terms of *processes* (or *transactions*), viz. sequence of events.

Real-World Non-Determinism. Most of the existing formal specification languages have been designed with the purpose of modelling the behaviour of a software component. In particular, in such languages, the assumption is that the description of state changes implicitly defines the system behaviour. Implicitly, it is assumed that actions occur when their preconditions are satisfied.

When dealing with the modelling of real-word things (e.g., a human behaviour), we need to introduce elements of uncertainty associated with occurrences of actions. Primitives are offered in Albert to support the modelling of deterministic happenings (things that *must* happen) as well as non-deterministic ones (things that *may* happen).

Application Scope. Unlike most of RE approaches focussing on the modelling of centralised business systems (i.e. non distributed systems), the scope of the Albert language is *real-time cooperative* systems.

The real-time aspect stems from our system modelling in terms of histories (i.e. sequences of time-stamped states and actions) allowing the expression of statements like "a book cannot be borrowed by a reader for more than 20 days" or "the push on the button at a floor is followed by the visit of the lift at this floor within the next two minutes".

The cooperative aspect is covered through the modelling of distributed systems in terms of agents, each of them being characterized with time-varying communication/information possibilities.

3 Albert : the Language

In this section, the Albert language is presented. After an intuitive description of the models associated with a specification, each constructs of the language is introduced. The last part of this section is devoted to comments on a real-size case study: the Chessmen Making Shop.

Running example

The Albert language will be introduced using parts of a small toy example: the Bolts Manufacture.

The Bolts Manufacture is composed of production cells and a manager. Each cell has two stocks: one for rivets, one for bolts. The manager is in charge of supplying the cells with rivets. When the manager gives a production order to a cell, the cell takes a rivet from its stock and produces a bolt from it within 10 minutes. The manager is also in charge of taking bolts out of the cells stocks.

3.1 Models of a Specification

The purpose of our requirements language is to define admissible behaviours of a composite system. This description, which must abstract of irrelevant details, is usually called a *model*

of the system. A specification language is best characterized by the structure of models it is meant to describe.

The rules for deriving the set of admissible models from a given specification expressed in the formal language is beyond the scope of this paper, which will remain informal.

In order to master their complexity, models of a specification are derived at two levels:

- at the agent level: a set of possible behaviours is associated with each agent without any regard to the behaviour of the other agents;
- at the society level: interactions between agents are taken into account and lead to additional restrictions on each individual agent behaviour.

The specification describes an agent by defining a set of possible *lives* modelling all its possible behaviours. A life is an (in)finite alternate sequence of *changes* and *states*; each state is labelled by a time value which increases all along the life (see Fig.2).

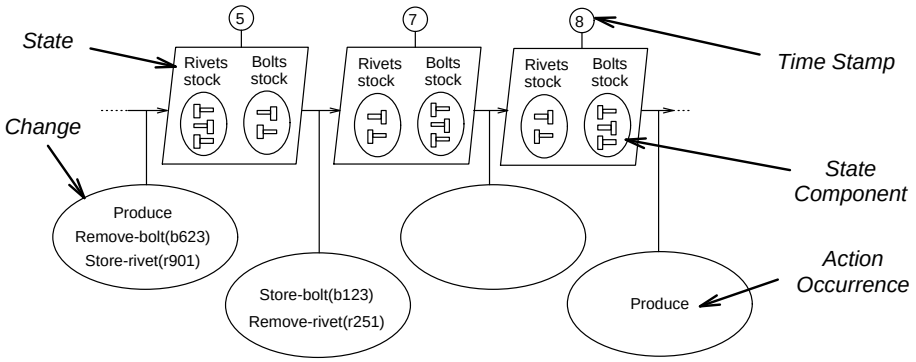


Figure 2: A possible life of the *Cell* agent

The term "*history*" refers to the sequence of changes which occur in a possible life of the agent.

A change is composed of several occurrences of simultaneous *actions* (the absence of action is also considered as a change). In our terminology we use the word 'action' both for denoting:

- happenings having an effect on the state where it occurs (called *actions* in some existing specification languages: e.g. [RFM91], [JSS91]);
- happenings with no direct influence on the state (called *events* in some other specification languages: e.g. [DHR91]).

The term "*trace*" refers to a sequence of states being part of a possible life of the agent. A state is structured according to the information handled in the considered application in terms of *state components*. In the case study, a specific state structure is associated with the *Cell* agent (see Sect.3.3).

The value of a state at a given time in a certain life can always be derived from the sub-history containing the changes occurred so far.

3.2 Language Constructs

Basically, the formal language that we propose is based on a variant of *temporal logic* [GB91], a mathematical language particularly suited for describing histories. This logic is itself an extension of multi-sorted first order logic, still based on the concepts of variables, predicates and functions. In this paper, three extensions are taken into account:

1. the introduction of **actions** to overcome the well-known *frame* problem [BMR92], a typical problem resulting from the use of a declarative specification language;
2. the introduction of **agents** together with their properties (responsibilities for actions, for providing perceptions, ...). This object-oriented concept can also be seen as a possible way of structuring large specifications in terms of more finer pieces, each of them corresponding to the specification of an agent guaranteeing a part of the global behaviour of the whole system;
3. the identification of **typical patterns of constraints** which support the analyst in writing complex and consistent formulas. In particular, typical patterns of formulas are associated with actions.

Using the language involves two activities: (i) writing *declarations* introducing the vocabulary of the considered application, and (ii) expressing *constraints*, i.e. logical statements which identify possible behaviours of the composite system and exclude unwanted ones.

A graphical syntax (with a textual counterpart) is used to introduce *declarations* and to express some typical *constraints* frequently encountered. The expression of the other constraints is purely textual.

3.3 Declarations

3.3.1 Declaration of Agents

The declaration part of an agent consists in the description of its states structure and the list of the actions its history can be made of. Importation and exportation links between agents are also graphically described.

Agents are considered as *specialized* objects; therefore, our modelling of a state structure is largely inspired by recent results in O-O conceptual modelling (see, e.g., OBLOG [SSE89] and O* [Brunet91]).

The state is defined by its components which can be *individuals* or *populations*. Usually populations are *sets* of individuals but they can also be structured in *sequences* or *tables*. Components can be time-varying or constant. Elements of components are typed using:

- predefined elementary data types (like, *STRING, BOOLEAN, INTEGER,...*) equipped with their usual operations [1];

[1] Operations on data types should not be confused with actions of agents: operations denote only mathematical functions, they may be used to simplify expressions in constraints but cannot be used to model the dynamic behaviour of systems (i.e. agents).

- elementary types defined by the analyst (like, *BOLT* and *RIVET* in our example), those are types for which no structure is given, they are only equipped with an equality predicate;
- more complex types built by the analyst using a set of predefined type constructors like extension[2], set, list, Cartesian product,... and elementary types; on top of operations inherited from their structure, new operations can be defined on these new types;
- types corresponding to agent identifiers. Agents includes a key mechanism that allows the identification of the different instances. A type is automatically associated to each class of agent. This corresponds to the type of agents identifiers within that class. E.g., each *Cell* agent has an identifier of type *CELL* [3] [4].

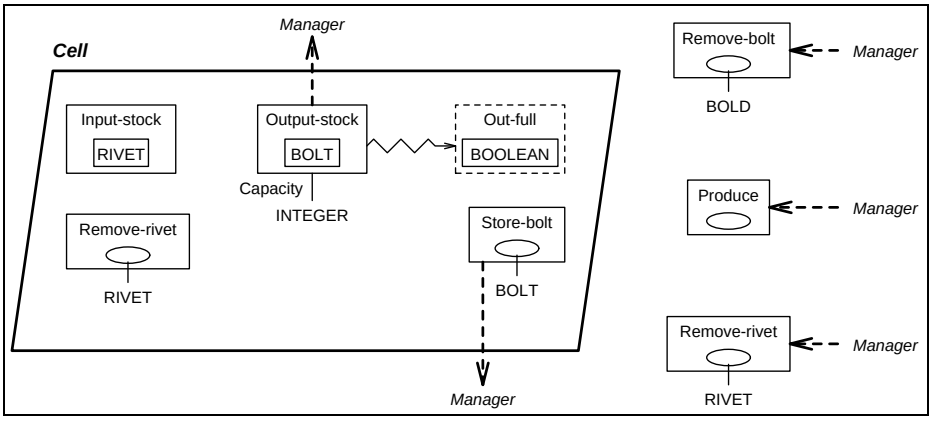


Figure 3: Declaration associated with the *Cell* agent

Figure 3 proposes the graphical diagram associated with the declaration of the state structure of the *Cell* where:

- *Input-stock* and *Output-stock* are considered as two set populations, respectively of type *RIVET* and *BOLT*;
- The *Output-stock* is characterized by a *Capacity* attribute of type *INTEGER*;
- *Out-full* is an instance of type *BOOLEAN*;
- *Produce*, *Store-rivet*, *Remove-bolt*, *Remove-rivet* and *Store-bolt* are five actions which may happen in a *Cell* history. Actions can have arguments[5]; for example, each occurrence of the *Store-rivet* action has an instance of type *RIVET* as argument.

The wavy line between the *Output-stock* and the *Out-full* components expresses that the value of the latter is derived from the former. The value of a component may be derived from one or several others.

[2]The extension type constructor (noted "*") adds a special value "UNDEF " to a data type. E.g., a variable of type *BOLT** may take any value of type *BOLT* or the special value "UNDEF ".

[3]When an agent is unique (like, e.g., the *Manager* agent), then a constant is also automatically defined to refer to the identifier of that agent.

[4]Inside the description of an agent, the *self* constant refers to the proper identifier of the described agent.

[5]Arguments may be regarded as input or output arguments but there is no difference on a semantics point of view.

The diagram also includes graphical notations making possible to distinguish between internal and external actions and to express the visibility relationships linking the agent to the outside (*Importation* and *Exportation* mechanisms):

(i) Information within the parallelogram is under the control of the described agent (the *Cell*) while information outside from the parallelogram denotes elements (state components or actions) which are imported from other agents of the society the agent belongs to. From the graphical declaration, it can be read that *Cell* has the initiative for *Remove-rivet* and *Store-bolt* actions while it lets the *Manager* having the initiative of *Store-rivet* and *Remove-bolt* actions [6];

(ii) For information in the parallelogram, boxes without arrow indicate that this information is not visible from the outside. Conversely, boxes with arrow denote information which is exported to the outside. From the graphical declaration, it can be read that the *Manager* may have knowledge of the *Output-stock* state and of the *Store-bolt* actions.

Importation and *Exportation* are static properties; *Perception* and *Information* are their dynamic counterparts and provides the analyst with a finer way of controlling how agents can see information inside each other (perception and information constraints will be discussed in Sect.3.4.3).

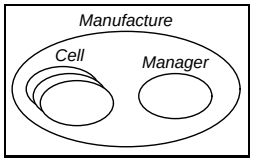


Figure 4: Declaration associated with the *Manufacture* agent

3.3.2 Declaration of a Society

Agents are grouped into societies. Societies themselves can be grouped together to form larger societies. In fact, a specification consists in a hierarchy of agents (a tree-like structure).

Figure 4 shows the declaration associated to the *Manufacture*.

The existing hierarchy among agents is expressed in term of two combinators: Cartesian Product and Set. In our specific case, the *Manufacture* agent is an aggregate of one *Manager* agent and several *Cell* agents (which form a "class").

3.4 Constraints

Constraints are used for pruning the (usually) infinite set of possible lives of an agent.

Unlike usual O-O design languages, the Albert semantics is not operational. A life must be extensively considered before it can be classified as possible or not, i.e. adding new states and changes at the end of a possible life does not necessarily result in a possible life.

[6] In the textual part of the specification, external actions will be referred prefixed with the identifier of the agent responsible for it

Figure 5 introduces the specification associated with the behaviour of the *Cell* agent and refers to the graphical declaration introduced in Fig.3.

In order to provide some methodological guidance to the analyst, properties are classified under ten headings and grouped into three families: Basic Constraints, Local Constraints, and Cooperation Constraints.

The identification of these different headings and families has been an important of our work because, from large case studies performed by analysts, we have experimented that a logical language (although sufficiently expressive) cannot be easily used because such a language is too flat and does not include any methodological guidance (a few years ago, the same was true for assembly programming languages and resulted in the development of new programming language offering higher level constructs).

3.4.1 Basic Constraints

Basic constraints are used to describe the initial state of an agent and to give the derivation rules for the derived components. Those constraints are respectively put under the headers **Initial Conditions** and **Derived Components**.

On Fig.5, the derived components constraint expresses that the boolean value Out-full is true when the number of items in the output stock is equal to its capacity and is false otherwise. The initial conditions constraint asserts that, in the initial state of the cell, the output stock is empty.

3.4.2 Local Constraints

Local constraints are related to the internal behaviour of the agent. They are classified under four headings: **State Behaviour**, **Effects of Actions**, **Causality** and **Capability**.

State Behaviour. Constraints under this heading express properties of the states or properties linking states in an admissible life of an agent.

First of all, there are constraints which are true in all states of the possible traces of an agent (see the first state behaviour constraint on Fig.5 expressing that the input stock may never be empty). These constraints are written according to the usual rules of strongly typed first order logic.

On top of constraints which are true in all states (usually referred as *invariants*), there are constraints on the evolution of the system (like, e.g. if this property holds in this state, then it holds in all future ones) or referring states at the different times (see the second state behaviour constraint on Fig.5 expressing that a rivet may not stay indefinitely in the input stock). Writing these constraints requires to be able to refer to more than one state at a time. This is done in our language by using additional temporal connectives which are prefixing statements to be interpreted in different states. These connectives ($\Diamond$, $\blacklozenge$, $\Box$, $\blacksquare$, $\mathcal{U}$, $\mathcal{S}$) are inspired from temporal logic (see e.g. [Sernadas80, MP91]) and express respectively "sometimes in the future", "sometimes in the past", "always in the future", "always in the past", "until", "since". There are constraints related to the expression of real-time properties. They are needed to describe delays or time-outs (like, e.g., "an element has to be removed from its

population within 15 minutes") and are expressed by subscripting temporal connectives with a time period. This time period is made precise by using usual time units: *Sec, Min, Hours, Days, ...* [Koymans92].

Effects of Actions. Beyond this heading, we describe the effects of actions[7] which may alter states in lives (see on Fig.5 how, e.g., an occurrence of the *Store-bolt(b)* action alters the output stock). Only actions which bring a traceable change are described here (for example, we do not describe the role of the *Produce* action).

In the description of the effect of an action, we use an implicit *frame rule* saying that states components for which no effect of actions are specified do not change their value in the state following the happening of a change.

The effect of an action is expressed in terms of a property characterising the state which follows the occurrence of the action. The value of a state component[8] in the resulting state is characterised in terms of a relationship referring to (i) the action arguments, (ii) the agent responsible for this action (if this action is an external one, the name of the agent is prefixing the action) and (iii) the previous state in the history.

In the last statement of the effects clause on Fig.5, we express that the effect associated with the action *Store-rivet* issued by the external agent *Manager* is to add a rivet in the *Input-stock* of the *Cell*. In the pattern associated with the definition of an action, the left hand side of the equation characterises the state as it results from the occurrence of the action while the right hand side refers to the state as it is before the occurrence of the action.

Causality. This heading is related to the *causality* relationship existing between some occurrences of actions.

Expressing causality rules with usual temporal connectives may appear very cumbersome (see, e.g., motivations given by [FS86]). To this end, our language is enriched with specific connectives which allow to specify, for example, that an action has to be issued by the agent as a unique response to the occurrence of another action (brought or not by the agent). A common pattern is based on the use of the "$\longrightarrow$" symbol which is not to be confused with the usual "$\Longrightarrow$" logical symbol. In our case, we want to denote some form of *entailment*, as it exists in Modal Logic [HC68].

In the case study, an example of causality exists between the *Produce*, the *Remove-rivet* and the *Store-bolt* actions. It relies upon the necessity of having a unique occurrence of the *Remove-rivet* and of the *Store-bolt* action (in that order) in response to each occurrence of the *Produce* action (see Fig.5).

The "$\longrightarrow$" symbol can be quantified by a temporal operator to express performances constraints (e.g. the "$\overset{\Diamond_{\leq 10'}}{\longrightarrow}$" symbol in the causality constraint on Fig.5 means that the occurrence of the *Store-bolt* action has to happen within a 10 minutes interval after the occurrence of the *Produce* action).

[7] This heading contains constraints which describe both effects of internal actions and effects of actions perceived from the outside.

[8] Please note that in an effect statement, derived components may not appear in the left part of a valuation.

Cell

BASIC CONSTRAINTS

DERIVED COMPONENTS

$Out\text{-}full \triangleq Card(Output\text{-}stock) = Capacity(Output\text{-}stock)$

INITIAL VALUATION

$Empty(Output\text{-}stock)$

LOCAL CONSTRAINTS

STATE BEHAVIOUR

$\neg\ Empty(Input\text{-}stock)$
$In(Input\text{-}stock,r) \Longrightarrow \Diamond\ \neg\ In(Input\text{-}stock,r)$

EFFECTS OF ACTIONS

$Remove\text{-}rivet(r)$: $Input\text{-}stock = Remove(Input\text{-}stock,r)$
$Store\text{-}bolt(b)$: $Output\text{-}stock = Add(Output\text{-}stock,b)$
$Manager.Remove\text{-}bolt(b)$: $Output\text{-}stock = Remove(Output\text{-}stock,b)$
$Manager.Store\text{-}rivet(r)$: $Input\text{-}stock = Add(Input\text{-}stock,r)$

CAUSALITY

$Manager.Produce \xrightarrow{\Diamond \leq 10'} Remove\text{-}rivet(r);\ Store\text{-}bolt(b)$

CAPABILITY

$\mathcal{F}$ ($Store\text{-}bolt(b)$ / $Out\text{-}full$)
$\mathcal{F}$ ($Remove\text{-}rivet(r)$ / $\neg\ In(Input\text{-}stock,r)$)

COOPERATION CONSTRAINTS

ACTION PERCEPTION

$\mathcal{K}$ ($Manager.Store\text{-}rivet$ / TRUE)
$\mathcal{I}$ ($Manager.Remove\text{-}bolt$ / $Empty(Output\text{-}stock)$)
$\mathcal{XK}$ ($Manager.Produce$ / $\neg\ Empty(Input\text{-}stock)$)

ACTION INFORMATION

$\mathcal{K}$ ($Store\text{-}bolt(b).Manager$ / TRUE)

STATE INFORMATION

$\mathcal{XK}$ ($Output\text{-}stock.Manager$ / $\neg\ Empty(Output\text{-}stock)$)

Figure 5: Constraints on the *Cell* agent

The right part of a commitment (the *reaction*) may only refer actions which are issued by the agent (i.e. actions which are not prefixed).

Left and right parts of a commitment may be composed of one or more occurrences of actions[9]. In case of more than one, occurrences may be composed in the following ways:

- "*act1* **;** *act2*" which means "an occurrence *act1* followed by an occurrence *act2*";
- "*act1* $\otimes$ *act2*" which means "an occurrence *act1* and an occurrence *act2* (at the same time)";
- "*act1* $\|$ *act2*" which means "an occurrence *act1* and an occurrence *act2* (in any order)";
- "*act1* $\oplus$ *act2*" which means "an occurrence *act1* or an occurrence *act2* (exclusive or)".

Some more complex expressions are provided to express iterative application of actions.

Capability. Under this heading, we describe the role of the agent with respect to the occurrence of its own actions. To this end, we are still using an additional extension of the classical first-order and temporal logic by making possible to express *permissions* associated with an agent. To this end, we consider three specific connectives allowing the expression of *obligations*, *preventions* and *exclusive obligations* (respectively the $\mathcal{O}$, the $\mathcal{F}$ and the $\mathcal{XO}$ connectives). The study of these connectives has been heavily influenced by some work performed in the area of *Deontic Logic* (see, e.g. [FM90], [Dubois91]).

The pattern for an obligation "$\mathcal{O}$ *(<int-action> / <situation>)*" expresses that the action has to occur if the circumstances expressed in the situation are matched (these circumstances refer to conditions on the current state).

The pattern for a prevention "$\mathcal{F}$ *(<int-action> / <situation>)*" expresses that the action is forbidden when the circumstances expressed in the situation are matched (e.g. the first constraint in Fig.5 expresses that "the cell cannot store a bolt into the stock when the stock is full", in other words, it is forbidden to the Cell to produce the *Store-bolt* action when the stock is full).

The pattern "$\mathcal{XO}$ *(<int-action> / <situation>)*" is used to express exclusive obligation, it is a shorthand for the combination of "$\mathcal{O}$ *(<int-action> / <situation>)*" and "$\mathcal{F}$ *(<int-action> /* $\neg$ *<situation>)*".

The default rule is that all actions are *permitted* whatever the situation.

Using these connectives makes possible to express the control that the agent has with respect to its internal actions.

3.4.3 Cooperation Constraints

This family of constraints specifies how the agent interacts with its environment, i.e. how it perceives action performed by other agents (**Action Perception**), how it can see parts of the

[9] In actions expressions, the special action identifier DAC may be used as a dummy action: e.g. "*a* **;** *(b* $\oplus$ DAC *)* **;** *c*" is a shortcut for "*(a* **;** *b* **;** *c)* $\oplus$ *(a* **;** *c)*".

state of other agents (**State Perception**), how it let other agents know what actions it does (**Action Information**) and how it shows parts of its state to other agents (**State Information**).

As said previously in Sect.3.3, perception and information provide the specifier a way to add a dynamic dimension to the importation and exportation relationship between agents expressed in the declaration part of the specification.

Action Perception. Beyond this heading we define how the agent is sensitive to changes occurring in its environment, which are made available to it by other agents belonging to the same society.

Action perceptions are specified using the $\mathcal{K}$ (*knowledge*), $\mathcal{I}$ (*ignorance*) and $\mathcal{XK}$ (*exclusive knowledge*) connectives.

The pattern "$\mathcal{K}$ *(<ext-action> / <situation>)*" defines the situation where, if an action is issued by the external agent, the behaviour of the current agent is influenced. For example, the first action perception constraint expresses that "the cell is always obliged to take into account the rivet storage performed by the Manager" (in other words, *Store-rivet* actions occurring in the Manager's life has necessarily to affect the history of the Cell).

The pattern "$\mathcal{I}$ *(<ext-action> / <situation>)*" defines the situation where, if such action is issued by the external agent, it has no influence on the current agent's behaviour.

The pattern "$\mathcal{XK}$ *(<ext-action> / <situation>)*" is used to express exclusive obligation, it is a shorthand for the combination of "$\mathcal{K}$ *(<ext-action> / <situation>)*" and "$\mathcal{I}$ *(<external-action> /* $\neg$ *<situation>)*".

The default rule is that all imported actions available may be perceived whatever the situation.

State Perception. Beyond this heading we define how the agent sees parts of the state of other agents belonging to the same society and which are made available to it by them. State perceptions are also specified using the $\mathcal{K}$, $\mathcal{I}$ and $\mathcal{XK}$ connectives.

The default rule is that all imported state components available may be perceived whatever the situation.

Action Information. Constraints under this heading specify how occurrences of actions performed by an agent are made available to other agents belonging to the same society. This is also a dynamic property and is expressed using the $\mathcal{K}$, $\mathcal{I}$ and $\mathcal{XK}$ connectives introduced above.

The pattern "$\mathcal{K}$ *(<int-action> . <agent> / <situation>)*" defines the situation where occurrences of an internal action are made available to a given agent[10]. For example, the action information constraint in Fig.5 expresses that "the cell always tells the Manager when it stores bolt in the stock (in other words, *Store-bolt* actions occurring in the cell life are always visible by the Manager)".

The pattern "$\mathcal{I}$ *(<int-action> . <agent> / <situation>)*" defines the situation where the occurrences of an internal action are not made visible for a given agent.

[10]Broadcast may be modelled by using here a free variable instead of an agent identifier.

The pattern "$\mathcal{XK}$ (<int-action> . <agent> / <situation>)" is used to express exclusive obligation, it is a shorthand for the combination of "$\mathcal{K}$ (<int-action> . <agent> / <situation>)" and "$\mathcal{I}$ (<int-action> . <agent> / $\neg$ <situation>)".

The default rule is that all exported actions may be visible by any agent to which it is exported, whatever the situation.

State Information. Beyond this heading we define how the agent shows parts of its state to other agents belonging to the same society. State information is also specified using the $\mathcal{K}$, $\mathcal{I}$ and $\mathcal{XK}$ connectives.

The default rule is that all exported state components may be visible by any agent to which it is exported, whatever the situation.

3.5 Case Study: the Chessmen Making Shop

In this sub-section, we report on the performance of a more real-size case study that the one handled in the previous sub-section. Because the lack of place, the full specification is presented in Appendix and we only discuss some specific aspects illustrating the possibilities offered by the  language.

In the first part, we completely rephrase the formal specification into informal terms. Comments on the formal specification are given in the second part.

3.5.1 Informal Description

The *Chessmen Making Shop* is a small manufacturing unit in charge of producing *chessmen* from wooden *cylinders*. It is composed of production *cells* (automated units) and is supervised by a *manager* (human actor).

The shop manager gives a *production order* to a cell when a chessman has to be machined. The order specifies the type of chessman to be produced (king, queen, castle, bishop, knight or pawn) and a coded reference.

A cell is composed of a number of *machine tools*, a *stock*, a *robot*, a *clamping system*, an *auto-guided vehicle* (AGV) and a *controller* (computer).

The cylinders and end products are stored in the stock. Before they can be machined, cylinders have to be clamped on a pallet. This work is done by the clamping system. The robot is used to withdraw parts from the stock and furnish them for the clamping operation. Once clamped, parts are transported to one machine by the AGV. If additional machining is required, the AGV picks the partly-machined part and transports it to another machine (once or several times) until the desired chessman is obtained. The chessman is then transported to the clamping system to be unclamped and stored by the robot in the stock (at the location where the raw cylinder was withdrawn). All these activities are coordinated by a controller. The whole production process of a chessman takes less than 20 minutes.

The stock is composed of a number of locations identified by an address. Each location can contain at most one item (cylinder or chessman). Items can either manually be entered in or

withdrawn from the stock by the shop manager or automatically by the robot. The manager is able to see if the stock is empty and an alarm is sent to him by the stock when it is full.

The robot is in charge of loading and unloading the clamping system. The grip of the robot is equipped with a sensor, so it can see which item is stored at a given location of the stock when its arm reaches that location. The same holds for the contents of the clamping system. The robot receives commands from the controller to load the clamping system with an item and commands to unload it. The commands specify the item to be transported and the address of the stock concerned by the operation, i.e. for a load (resp. unload) command, the location of the stock where the item has to be taken (resp. put).

The clamping system role is to clamp and unclamp the items it receives. It is an automatic device: it automatically clamps any item put on it by the robot and automatically unclamps any item brought by the AGV. Clamping and unclamping operations take at most one minute. The clamping system may handle one item at a time and must be unloaded between two operations. The clamping system sees what is carried by the robot and the AGV when these are located at the clamping system.

The AGV has two places where items can be put. But those places can never be occupied at the same time during transportation (there are only used to perform some exchange between an item already in the AGV and an item located on a device (machine tool or clamping system). The AGV receives commands from the controller asking it to carry one item from one location to another. In reaction to these commands, it has to move to the origin (if it was not yet there), pick up the item, move to the destination and deliver the item (if there was already an item on the destination device, an exchange must take place). The AGV sees the contents of a device (a machine tool or the clamping system) where it is located. The AGV notifies the devices when it picks up (resp. delivers) an item from (resp. to) it.

Machine tools are used to apply basic transformations to raw materials or partly-machined items. Each machine tool has some capabilities, i.e. the set of programs it is able to run. Machine tools receive commands from the controller. A command, addressed to a given machine tool, specifies the program to be run. Machine-tools notify the controller when the program they had to run has been completed. Machine tools of a cell are independent and may work simultaneously.

The controller is in charge of achieving orders it receives from the manager by issuing commands to the devices of the cell. The software running in the controller contains a “recipe book” which associate a recipe to each chessman type. A recipe consists in a sequence of production steps. Each production step is characterized by a machine and a program to be run on it. The controller keeps traces of the status of each order it is processing and the status of each component of the cell.

The communication between the controller and the machine-tools is not always reliable. In order to detect problems, the controller marks a machine as “down” when a command was sent to a machine tool and no end-of-work notification has been received within 10 minutes. The controller send then an alarm message to the manager specifying which machine appeared faulty. The controller does not use “down” machine tools anymore.

3.5.2 Comments on the Formal Specification

The formal specification corresponding to the informal description shown above is given in Appendix A. The process followed to build this specification will be presented in Sect.4. Informal comments are provided in the Albert text of the specification to facilitate understanding.

In this part, interesting properties of some parts of the specification will be highlighted and discussed: (i) concurrent behaviour and non-determinism; (ii) fine-grain visibility control; (iii) reliability of agents.

Concurrent Behaviour and Non-Determinism. As other languages Albert allows to describe systems in an operational way. But it also allows to give declarative properties, i.e. considering extensively the life of an agent.

Causality constraints are good examples of declarative properties. In the specification of the controller (see Sect.A.6), the causality relationship means that each production order has to be followed by a list of actions starting some subprocesses, but this does not preclude that nothing else may happen in the meantime. In fact, the controller may process several orders at the same time.

A common problem encountered in concurrent systems is deadlocks. Components of the cell are working in parallel so we could imagine deadlocks may happen. The role of requirements specification is just to state that deadlocks should be avoided but not to express solutions to avoid them (this will be done during design). The causality constraints mentioned above is sufficient to express that we do not want deadlocks: only lives with finishing order processing (thus without deadlocks) will be accepted as valid models of the specification.

Fine-grain Visibility Control. The visibility mechanism taking place among agents (i.e. the way they exchange information) is quite elaborate in Albert. Several levels are used to model with precision visibility control:

- *static importation/exportation mechanism*
  This syntactic mechanism constrains which state components of one agent may be referred to in the specification of another one. See, e.g., the declaration associated with the robot (Sect.A.2) where it is stated that the *Robot* agent imports the *Contents* component from the *Stock* agent. This means that references to this component may be made in constraints describing the robot.

- *state information mechanism*
  This mechanism dynamically restricts what exported state component is effectively available and to which agent (i.e. to which instance(s)) it is made available. See, e.g., the state information constraint on the stock (Sect.A.1) which restricts the visibility of the robot on its contents to the location where the robot stands (if it is on the stock).

- *state perception mechanism*
  This last mechanism may restrict furthermore the quantity of information exchanged between agents by putting constraints on when imported components, which where made available to the agent by another, are indeed perceived (accessed) by the agent. See, e.g.,

the state perception constraints (Sect.A.2) which says that the robot looks at the contents of a location of the stock only when it is standing at that location[11].

Those three mechanisms act as successive filters on channels carrying information between agents (see Fig.6).

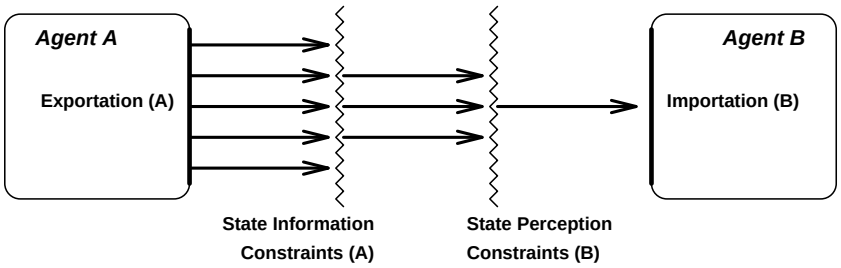


Figure 6: Fine-grain visibility control

In an expression, the value of an imported components which is not made available or is not perceived, is the special value "UNDEF ".

Reliability of Agents. A similar three levels mechanism is provided to deal with control flow between agents in . This provides the specifier a nice way to model reliability of agents.

In the informal statement of the problem it is said that "the communication between the controller and the machine-tools is not always reliable". This is quite straightforwardly modelled in the specification of the machine-tool (see Sect.A.5):

- The first action perception constraint defines some situation where the *Run* action from the controller is not seen but nothing is said about other situations. The default rule is that it *may* be seen (exactly what happen through an unreliable communication channel).
- No action information constraint is given about the *Work-done* action which is exported from the machine tool to the controller. Here again, the default rule will apply and this means that occurrences of the *Work-done* action *may* be made visible to the controller.

This way, communication (from a control flow point of view) between the controller and the machine tool is modelled as unreliable in both directions.

As for visibility of state components, constructs here act as successive filters on the control flow and an agent has always "the last word" on control information coming from the outside.

4 Methodological Guidelines at the RE Level

In the RE field as well as in other fields, a language is not usable for real applications if it is not supported by an appropriate and effective methodology. The requirements document, in its final

[11]For this example, the perception constraints (in the robot specification) matches the corresponding information (in the stock specification) but this is not necessarily the case.

version, is usually a complex document due to the number of individual agents belonging to the system and the complexity of interactions taking place among agents. Therefore, one cannot imagine that the requirements document can be written in one shot and it is found essential to provide some methodological guidance in the elaboration of successive and incremental versions of the requirements document.

As depicted in Fig.7, the elaboration of a specification can be seen as a sequence of development steps, each step being defined by the application of transformations on the current version of the requirements document and resulting in a new version of this document. The application of a specific transformation at some stage of the development depends on some *strategies* followed by the analyst. In the rest of this section, we briefly discuss and illustrate two possible elaboration strategies based on a progressive refinement of the specification. Another strategy has been described in [DDP93a] and consist in the possibility of reusing generic specification components.

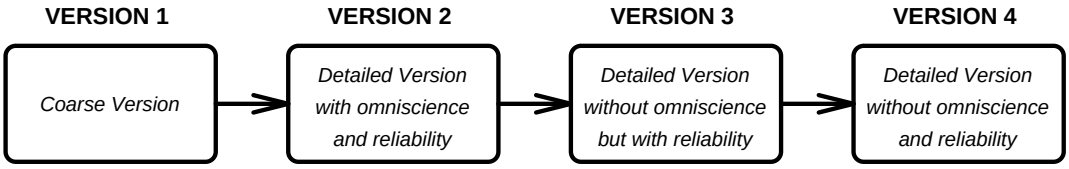


Figure 7: Example of RE process

4.1 A "Goal-oriented" Strategy

The main idea behind this strategy is based on a top-down agent *refinement* philosophy [Bjorner92] [Dubois89]. Tackling a new problem, the analyst will have to isolate and specify the main **goals** of a system first, before to refine them progressively in terms of finer requirements that can be attached to finer subsystems. To be more precise, the strategy is:

1. Identify the goals of the system and specify them in terms of a unique monolithic agent.
2. Identify new sub-agents and attach to each of them their individual responsibilities so that the behaviour of the sub-agents preserves the original goals.
3. Apply recursively the step 2 on each agent up to the identification of 'terminal' agents, i.e. components for which designers agree on the implementation of their attached responsibilities.

Within the framework of the *Chessmen Making Shop* system introduced in the previous section, the *goal-oriented* strategy has been applied to result in the second version of the requirements document (see Fig.7):

- In the first version, the identification of the problem has resulted in the specification of a system made of two agents (the *Cell* and the *Manager*). At this stage of elaboration, the *Cell* is considered as an individual agent for which the given specification states that "a production order received by the cell should cause the delivery of an appropriated manufactured cylinder within a 20 minutes delay". This specification is considered as the *goal* of the system.

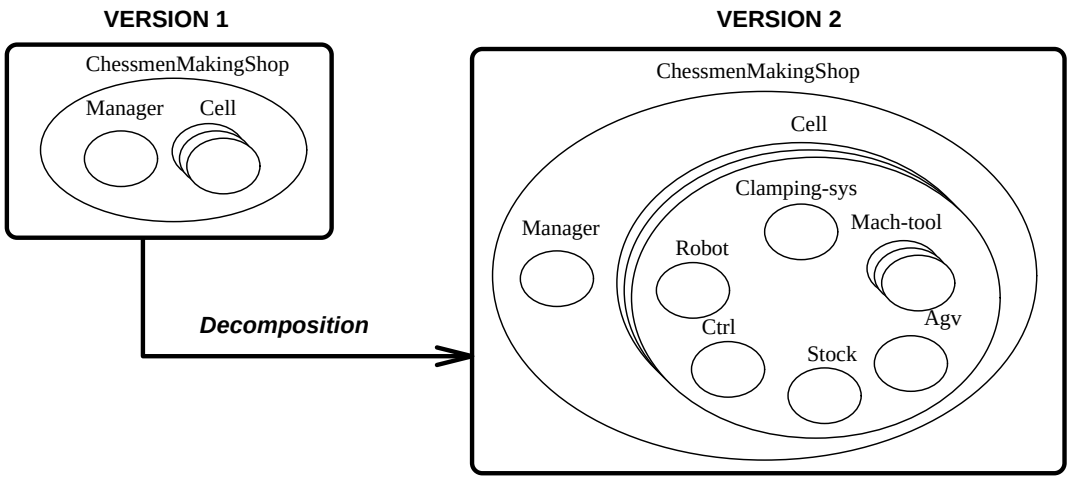


Figure 8: Application of the "Goal-oriented" Strategy

- In the second version, the application of the goal strategy defined above has resulted in a new version of the requirements document where the original *Cell* agent has been refined in terms of six finer agents (*Stock*, *Robot*, *Clamping-sys*, *Agv*, *Mach-tool* and *Ctrl*, see Fig.8).

4.2 A "Retracting-Assumptions" Strategy

In version 2 of the requirements document, the six agents being components of the *Cell* have been identified. However, due to the complexity of their individual descriptions, the analyst has imagined an idealized version of them which liberates him/her from too many specification details in the requirements document. In other words, the work of the analyst was based on some assumptions related to the behaviours of the involved agents. In the proposed strategy (inspired by the pioneering work of [Feather89]), we consider two kinds of possible assumptions:

- The first assumption is based on an *omniscience* hypothesis i.e. the perfect knowledge an agent has of the states and actions of other agents.
- The second assumption is based on a *reliability* hypothesis associated with each agent, i.e. its responsibility for performing the appropriate actions under the given circumstances.

The proposed strategy is the following one:

1. Specify a system where: (i) the internal state and actions of each agent are visible from each other and (ii) all agents are considered reliable.
2. Elaborate a new specification by retracting the first assumption, viz: (i) the internal state of each agent is not necessarily visible from the outside and (ii) all agents are still reliable.
3. Elaborate the final specification of the system by retracting the reliability assumption associated with each agent.

In our example, the specification, contained in the version 2 of the requirements document, has been built by the analyst with these two assumptions in mind. This means, for example, that

the controller can access to the *status* of the clamping system (*omniscience* hypothesis) and that the machine tool is reliable w.r.t. its communication with the controller (*reliability* hypothesis).

The application of the step 2 of the proposed strategy resulted in a new version of the requirements document (version 3). At this level, the controller has now to maintain information recording the status of the clamping system. This information aims at mirroring the real status of the clamping system but this duplication is needed since the controller has no longer access to the state of the clamping system and thereby has to derive its status from the control commands delivered to it.

Finally, in version 4 of the requirement document, the *reliability* assumption attached to agents has been removed (this version is the one fully presented in Appendix A). This, again, has the consequence to add complexity to the requirements document by having to consider and incorporate new details to deal with failures, as an example one may refer to the handling of failure in the communication between the machine tool and the controller.

5 Conclusion

Multiple experiences have shown that an engineering approach such  cannot be effectively adopted by analysts if it is not supported by tools. In the long term, we want to build an integrated requirements engineering environment for  . For the moment, two basic tools are made available:

1. **Editing facilities** are offered through the use of a structural editor that will support 'graphical' as well as 'textual' syntaxes. The graphical part of this editor is already implemented using the *GraphTalk*[12] tool which allows to manage multiple views of the same requirements specification (several figures presented in this paper have been edited using this tool).

2. **Validation facilities** are offered through an *animator* tool which can be used for the purpose of verifying the adequacy of formal requirements specifications with respect to what customers really want to do. Discussions around a set of logical formulae cannot be an acceptable basis and thereby an animator can be used for testing dynamically the possible behaviours of the described system (a classical prototyping approach could not be applied here due to the non-executable nature of ALBERT). More about the animator can be found in [DDD94].

Within short term, we have some plan to develop checking facilities developed on the basis of the formal interpretation rules associated with the language.

Two other researches directions are related to:

- at the practical level, the use of  in some other specific application domains. Preliminary investigations are performed in domain of telecommunications, Electronic Data Interchange and office automation (*workflows*);

[12] *GraphTalk* is a registered trade mark of Rank Xerox

- at the method level, we are working on the elaboration of formalised methodological guidelines for writing requirements documents as well as for integrating other kinds of requirements like *organizational* and *non-functional* requirements (see, e.g., [Yu93],[BCDS93]).

References

[BCDS93] Blyth (A.J.C.), Chudge (J.), Dobson (J.E.) et Strens (M.R.). – ORDIT: a new methodology to assist in the process of eliciting and modelling organisational requirements. *In : Proc. of the conference on organizational computing systems – COOCS'93*, éd. par Kaplan (Simon). pp. 216–227. – Milpitas CA, November 1-4, 1993.

[Bjorner92] Bjørner (D.). – Trusted computing systems: The ProCoS experience. *In : Proc. of the 14th International Conference on Software Engineering – ICSE'92*. IEEE, pp. 15–34. – Melbourne (Australia), May 11-15, 1992.

[BMR92] Borgida (A.), Mylopoulos (J.) et Reiter (R.). – *. . .And Nothing Else Changes : The Frame Problem in Procedure Specifications*. – Technical Report n° DCS-TR-281, Dept. of Computer Science, Rutgers University, 1992.

[Brunet91] Brunet (J.). – Modelling the world with semantic objects. *In : Proc. of the working conference on the object-oriented approach in information systems*. – Québec (Canada), 1991.

[Bubenko80] Bubenko (J.A.). – Information modeling in the context of system development. *In : Information Processing 80*, éd. par Lavington (S.H.). pp. 395–411. – North-Holland.

[CY91] Coad (P.) et Yourdon (E). – *Object-Oriented Design*. – Englewood Cliffs, New Jersey, Prentice-Hall, 1991.

[DDD94] Dubois (Eric), Du Bois (Philippe) et Dubru (Frédéric). – Animating formal requirements specifications of cooperative information systems. *In : Proc. of the Second International Conference on Cooperative Information Systems – CoopIS-94*. pp. 101–112. – Toronto (Canada), May 17-20, 1994.

[DDP93a] Dubois (Eric), Du Bois (Philippe) et Petit (Michaël). – Elicitating and formalising requirements for CIM information systems. *In : Proc. of the 5th conference on advanced information systems engineering – CAiSE'93*, éd. par Rolland (C.), Bodart (F.) et Cauvet (C.). pp. 252–274. – Paris (France), June 8-11, 1993.

[DDP93b] Dubois (Eric), Du Bois (Philippe) et Petit (Michaël). – O-O requirements analysis: an agent perspective. *In : Proc. of the 7th European Conference on Object-Oriented Programming – ECOOP'93*, éd. par Nierstrasz (O.). pp. 458–481. – Kaiserslautern (Germany), July 26-30, 1993.

[DHR91] Dubois (Eric), Hagelstein (Jacques) et Rifaut (André). – A formal language for the requirements engineering of computer systems. *In : From natural language processing to logic for expert systems*, éd. par Thayse (André), chap. 6. – Wiley, 1991.

[Dubois89] Dubois (Eric). – A logic of action for supporting goal-oriented elaborations of requirements. *In : Proc. of the 5th International Workshop on Software Specification and Design – IWSSD'89*. IEEE, pp. 160–168. – Pittsburgh PA, May 19-20, 1989.

[Dubois91] Dubois (Eric). – Use of deontic logic in the requirements engineering of composite systems. *In : Proc. of the First International Workshop on Deontic Logic in Computer Science – DEON'91*, éd. par Meyer (J.J.) et Wieringa (R.J.). – Amsterdam (The Netherlands), December 11-13, 1991.

[Feather87] Feather (Martin S.). – Language support for the specification and development of composite systems. *ACM Transactions on Programming Languages and Systems*, vol. 9, nº 2, April 1987, pp. 198–234.

[Feather89] Feather (Martin S.). – Constructing specifications by combining parallel elaborations. *IEEE Transactions on Software Engineering*, vol. SE-15, nº 2, February 1989.

[FM90] Fiadeiro (Jose) et Maibaum (Tom). – Describing, structuring and implementing objects. *In : Foundations of Object-Oriented Languages - REX School/Workshop*. pp. 275–310. – Noordwijkerhout (The Netherlands), May 28 - June 1, 1990.

[FP87] Finkelstein (Anthony) et Potts (Colin). – Building formal specifications using "structured common sense". *In : Proc. of the 4th International Workshop on Software Specification and Design – IWSSD'87*. IEEE, pp. 108–113. – Monterey CA, April 3-4, 1987.

[FS86] Fiadeiro (Jose) et Sernadas (Amilcar). – Linear tense propositional logic. *Information Systems*, vol. 11, nº 1, 1986, pp. 61–85.

[GB91] Gabbay (D.) et Brien (P. Mc). – Temporal logic and historical databases. *In : Proc. of the 17th International Conference on Very Large Databases – VLDB'91*. – Barcelona, September 1991.

[GBM86] Greenspan (Sol J.), Borgida (Alexander) et Mylopoulos (John). – A requirements modeling language. *Information Systems*, vol. 11, nº 1, 1986, pp. 9–23.

[HC68] Hughes (G.E.) et Cresswell (M.J.). – *An Introduction to Modal Logic*. – London, Methuen and Co., 1968.

[Hewitt91] Hewitt (C.). – DAI betwist and between : open systems science and/or intelligent agents. *In : Proc. of the international workshop on the development of intelligent information systems*, éd. par Mylopoulos (J.) et Balzer (R.). – Niagara-on-the-Lake (Canada), April 21-23, 1991.

[JSS91] Jungclaus (R.), Saake (G.) et Sernadas (C.). – Formal specification of object systems. *In : Proc. of TAPSOFT'91 Vol.2*, éd. par Abramsky (S.) et Maibaum (T.). pp. 60–82. – Brighton (UK), 1991.

[Koymans92] Koymans (Ron). – *Specifying message passing and time-critical systems with temporal logic*. – LNCS 651, Springer-Verlag, 1992.

[MBJK90] Mylopoulos (J.), Borgida (A.), Jarke (M.) et Koubarakis (M.). – Telos : A language for representing knowledge about information systems. *ACM Transansaction on Information Systems*, vol. 8, nº 4, 1990, pp. 325–362.

[MP91] Manna (Zohar) et Pnueli (Amir). – *The temporal logic of concurrent systems*. – Springer-Verlag, 1991.

[RFM91] Ryan (Mark D.), Fiadeiro (Jose) et Maibaum (Tom). – Sharing actions and attributes in modal action logic. *In : Theoretical Aspects of Computer Software*, éd. par Ito (T.) et Meyer (A.). – Springer-Verlag, 1991.

[Sernadas80] Sernadas (Amilcar). – Temporal aspects of logic procedure definition. *Information Systems*, vol. 5, 1980, pp. 167–187.

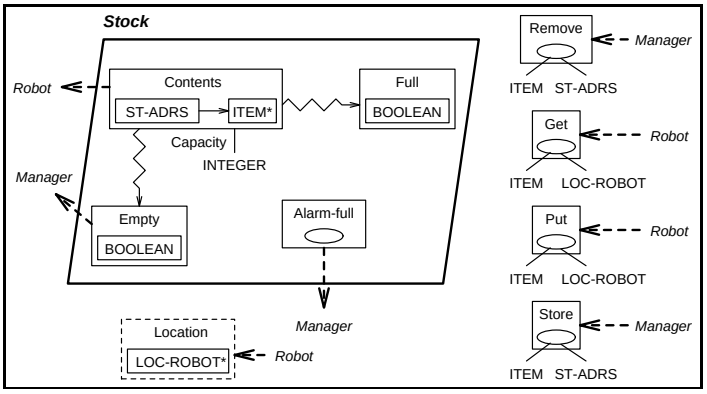
[SM88] Shlaer (S.) et Mellor (S.J.). – *Object-oriented systems analysis: modelling the world in data*. – Englewood Cliffs, New Jersey, Yourdon Press: Prentice-Hall, 1988.

[SSE89] Sernadas (A.), Sernadas (C.) et Ehrich (H.-D.). – Abstract object types: a temporal perspective. *In : Proc. of the colloquium on temporal logic and specification*, éd. par Banieqbal (B.), Barringer (H.) et Pnueli (A.). pp. 324–350. – LNCS 398, Springer-Verlag.

[Yu93] Yu (Eric S. K.). – An organization modelling framework for information systems requirements engineering. *In : Proc. of the 3rd Workshop on Information Technologies and Systems – WITS'93*. – Orlando FL, December 4-5, 1993.

A ALBERT Specification of the Chessmen Making Shop (detailed version)

A.1 The Stock



Stock

BASIC CONSTRAINTS

DERIVED COMPONENTS

Empty $\triangleq$ *Card(Contents) = 0*
Full $\triangleq$ *Card(Contents) = Capacity(Contents)*

INITIAL VALUATION

Contents[_] = UNDEF

LOCAL CONSTRAINTS

STATE BEHAVIOUR

Card(Contents) $\leq$ *Capacity(Contents)*

EFFECTS OF ACTIONS

Manager.Store(it,c): Contents = Add(Contents,it,c)
Manager.Remove(_,c): Contents = Remove(Contents,c)
Robot.Put(it,c): Contents = Add(Contents,it,c)
Robot.Get(_,c): Contents = Remove(Contents,c)

CAPABILITY

$\mathcal{O}$ (*Alarm-Full* / *Full*)

COOPERATION CONSTRAINTS

ACTION PERCEPTION

$\mathcal{XK}$ (*Manager.Store(_,c)* / *Contents[c]* = UNDEF $\wedge$ $\neg$ *Full*)
$\mathcal{XK}$ (*Manager.Remove(it,c)* / *Contents[c] = it*)
$\mathcal{XK}$ (*Robot.Put(_,c)* / *Contents[c]* = UNDEF $\wedge$ $\neg$ *Full*)
$\mathcal{XK}$ (*Robot.Get(it,c)* / *Contents[c] = it*)

STATE PERCEPTION

The stock always perceives the location of the robot.
$\mathcal{K}$ (*Robot.Location* / TRUE)

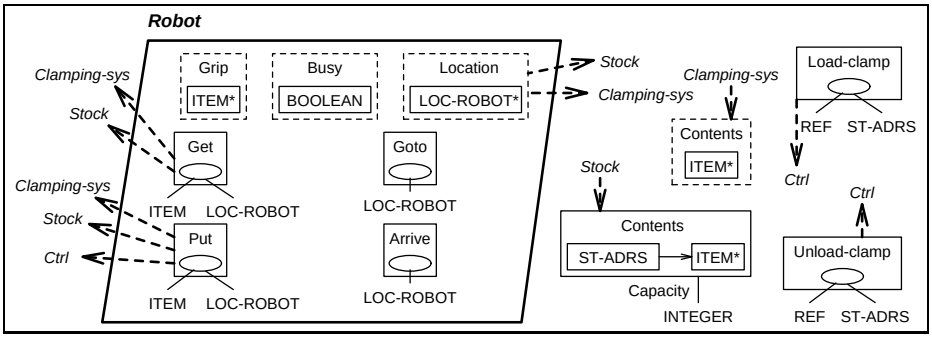
ACTION INFORMATION

$\mathcal{K}$ (*Alarm-Full.Manager* / TRUE)

STATE INFORMATION

The stock shows its contents, at a given address, to the robot if and only if the robot is located at that address.
$\mathcal{XK}$ (*Contents[i].Robot* / *Robot.Location = i*)
$\mathcal{K}$ (*Empty.Manager* / TRUE)

A.2 The Robot



Robot

BASIC CONSTRAINTS

INITIAL VALUATION

Grip = UNDEF
Busy = FALSE

LOCAL CONSTRAINTS

EFFECTS OF ACTIONS

Get(it,_): Grip = it
Put(_,_): Grip = UNDEF
Busy = FALSE
Goto(_): Location = UNDEF
Arrive(c): Location = c
Ctrl.Load-Clamp(_,_): Busy = TRUE
Ctrl.Unload-Clamp(_,_): Busy = TRUE

CAUSALITY

Ctrl.Load-Clamp(_,c) $\xrightarrow{\Diamond}$ ▌a. The robot is not yet located at the right address.
((Goto(c) ; *Arrive(c))* ⊕ DAC *)* ;
▌b. The robot is already located at the right address.
Get(it,c) ; *Goto(Clamping-sys)* ; *Arrive(Clamping-sys)* ; *Put(it,Clamping-sys)*

Ctrl.Unload-Clamp(_,c): $\xrightarrow{\Diamond}$ ▌a. The robot is not yet located at the clamping system.
((Goto(Clamping-sys) ; *Arrive(Clamping-sys))* ⊕ DAC *)* ;
▌b. The robot is already located at the clamping system.
Get(it,Clamping-sys) ; *Goto(c)* ; *Arrive(c)* ; *Put(it,c)*

CAPABILITY

$\mathcal{F}$ (*Goto(c)* / *Location = c*)
$\mathcal{F}$ (*Get(it,Clamping-sys)* / *Location* $\neq$ *Clamping-sys* $\vee$ *Grip* $\neq$ UNDEF $\vee$ *Clamping-sys.Contents* $\neq$ *it*)
$\mathcal{F}$ (*Get(it,c)* with *Is-of_ST-ADRS(c)* / *Location* $\neq$ *c* $\vee$ *Grip* $\neq$ UNDEF $\vee$ *Stock.Contents[c]* $\neq$ *it*)
$\mathcal{F}$ (*Put(it,Clamping-sys)* / *Location* $\neq$ *Clamping-sys* $\vee$ *Grip* $\neq$ *it* $\vee$ *Clamping-sys.Contents* $\neq$ UNDEF)
$\mathcal{F}$ (*Put(it,c)* with *Is-of_ST-ADRS(c)* / *Location* $\neq$ *c* $\vee$ *Grip* $\neq$ *it* $\vee$ *Stock.Contents[c]* $\neq$ UNDEF)

COOPERATION CONSTRAINTS

ACTION PERCEPTION

$\mathcal{XK}$ (*Ctrl.Load-Clamp(_,_)* / ¬ *Busy*)
$\mathcal{XK}$ (*Ctrl.Unload-Clamp(_,_)* / ¬ *Busy*)

STATE PERCEPTION

$\mathcal{XK}$ (*Stock.Contents[i]* / *Location = i*)
$\mathcal{XK}$ (*Clamping-sys.Contents* / *Location = Clamping-sys*)

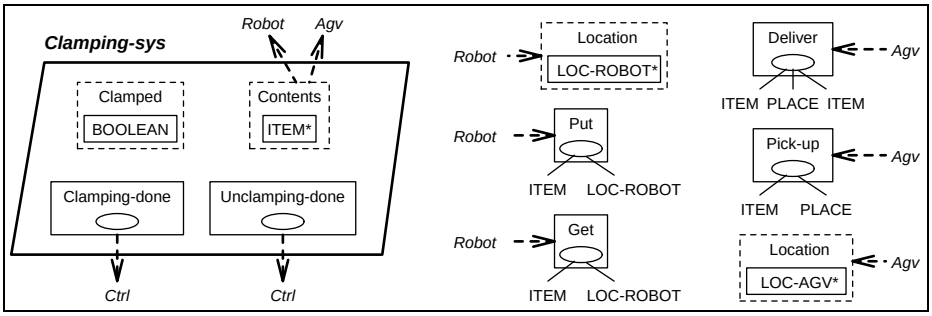
ACTION INFORMATION

$\mathcal{XK}$ (*Get(_,_).Clamping-sys* / *Location = Clamping-sys*)
$\mathcal{XK}$ (*Get(_,_).Stock* / *Is-of_ST-ADRS(Location)*)
$\mathcal{XK}$ (*Put(_,_).Clamping-sys* / *Location = Clamping-sys*)
$\mathcal{XK}$ (*Put(_,_).Stock* / *Is-of_ST-ADRS(Location)*)

STATE INFORMATION

▌The robot shows its location to the stock when it is on one of the addresses of it.
$\mathcal{XK}$ (*Location.Stock* / *Is-of_ST-ADRS(Location)*)
$\mathcal{XK}$ (*Location.Clamping-sys* / *Location = Clamping-sys*)

A.3 The Clamping System



Clamping-sys

> The clamping system is an automatic device: it clamps (resp. unclamps) any item delivered by the robot (resp. a.g.v.) without waiting for any signal from the controller.

BASIC CONSTRAINTS

INITIAL VALUATION

Contents = UNDEF
Clamped = FALSE

LOCAL CONSTRAINTS

EFFECTS OF ACTIONS

Clamping-done: Clamped = TRUE
Unclamping-done: Clamped = FALSE
Agv.Deliver(it,_,_): Contents = it
Clamped = TRUE
Agv.Pick-up(it,_): Contents = UNDEF
Robot.Put(it,Clamping-sys): Contents = it
Clamped = FALSE
Robot.Get(it,Clamping-sys): Contents = UNDEF

CAUSALITY

> Clamping and unclamping take at most 1'.

Agv.Deliver(_,_,_) $\xrightarrow{\Diamond_{\leq 1'}}$ *Unclamping-done*

Robot.Put(_,Clamping-sys) $\xrightarrow{\Diamond_{\leq 1'}}$ *Clamping-done*

CAPABILITY

$\mathcal{F}$ (*Clamping-done* / *Contents* = UNDEF)
$\mathcal{F}$ (*Unclamping-done* / *Contents* = UNDEF)

COOPERATION CONSTRAINTS

ACTION PERCEPTION

$\mathcal{XK}$ (*Agv.Deliver(_,_,_)* / *Contents* = UNDEF)
$\mathcal{XK}$ (*Agv.Pick-up(it,_)* / *Contents = it*)
$\mathcal{XK}$ (*Robot.Put(_,_)* / *Contents* = UNDEF)
$\mathcal{XK}$ (*Robot.Get(it,_)* / *Contents = it*)

STATE PERCEPTION

$\mathcal{K}$ (*Robot.Location* / TRUE)
$\mathcal{K}$ (*Agv.Location* / TRUE)

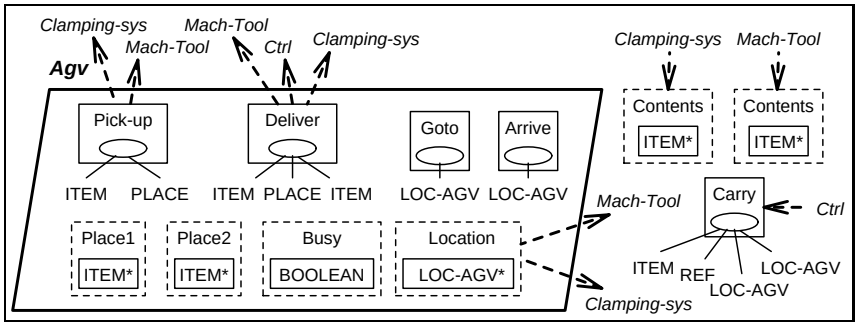
ACTION INFORMATION

$\mathcal{K}$ (*Clamping-done.Ctrl* / TRUE)
$\mathcal{K}$ (*Unclamping-done.Ctrl* / TRUE)

STATE INFORMATION

$\mathcal{XK}$ (*Contents.Robot* / *Robot.Location* = SELF)
$\mathcal{XK}$ (*Contents.Agv* / *Agv.Location* = SELF)

A.4 The Auto-Guided Vehicule



Agv

BASIC CONSTRAINTS

INITIAL VALUATION

Place1 = UNDEF
Place2 = UNDEF
Busy = FALSE

LOCAL CONSTRAINTS

STATE BEHAVIOUR

Location = UNDEF $\Longrightarrow$ *place1* = UNDEF $\vee$ *place2* = UNDEF

EFFECTS OF ACTIONS

Pick-up(it,1): Place1 = it
Pick-up(it,2): Place2 = it
Deliver(_,1,xit): Place1 = UNDEF
Place2 = xit
Busy = FALSE
Deliver(_,2,xit): Place1 = xit
Place2 = UNDEF
Busy = FALSE
Arrive(c): Location = c
Busy = FALSE
Goto(_): Location = UNDEF
Ctrl.Carry(_,_,_,_): Busy = TRUE

CAUSALITY

$Goto(c) \xrightarrow{\Diamond} Arrive(c)$

$Ctrl.Carry(\_,it,c1,c2) \xrightarrow{\Diamond} ((((Goto(c1)\ ;\ Arrive(c1)) \oplus \text{DAC}\ )\ ;\ Pick\text{-}up(it,\_)) \oplus \text{DAC}\ )\ ;$
$Goto(c2)\ ;\ Arrive(c2)\ ;\ Deliver(it,\_,\_)$

CAPABILITY

$\mathcal{F}$ (*Goto(c)* / *Location* = *c*)
$\mathcal{F}$ (*Pick-up(_,1)* / *Place1* $\neq$ UNDEF)
$\mathcal{F}$ (*Pick-up(_,2)* / *Place2* $\neq$ UNDEF)
$\mathcal{F}$ (*Pick-up(it,_)* / *Location* = UNDEF $\vee$ *Location.Contents* $\neq$ *it*)
$\mathcal{F}$ (*Deliver(it,1,_)* / *Place1* $\neq$ *it*)
$\mathcal{F}$ (*Deliver(it,2,_)* / *Place2* $\neq$ *it*)
$\mathcal{F}$ (*Deliver(it,_,xit)* / *Location* = UNDEF $\vee$ *Location.Contents* $\neq$ *xit*)

COOPERATION CONSTRAINTS

ACTION PERCEPTION

$\mathcal{XK}$ (*Ctrl.Carry(_,_,_,_)* / $\neg$ *Busy*)

STATE PERCEPTION

$\mathcal{XK}$ (*a.Contents* / *Location* $\neq$ UNDEF $\wedge$ *Location* = *a*)

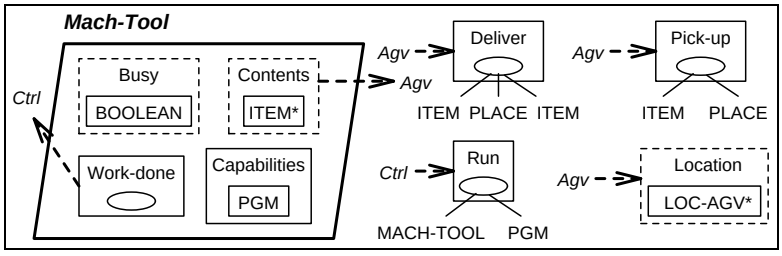
ACTION INFORMATION

$\mathcal{XK}$ (*Pick-up(_,_).a* / *Location* = *a*)
$\mathcal{XK}$ (*Deliver(_,_,_).a* / *Location* = *a* $\vee$ *a* = *Ctrl*)

STATE INFORMATION

$\mathcal{XK}$ (*Location.a* / *Location* $\neq$ UNDEF $\wedge$ *Location* = *a*)

A.5 The Machine Tool



Mach-tool

BASIC CONSTRAINTS

INITIAL VALUATION

Contents = UNDEF
Busy = FALSE

LOCAL CONSTRAINTS

EFFECTS OF ACTIONS

Work-done: Busy = FALSE
Contents = new
Ctrl.Run(_,_): Busy = TRUE
Agv.Deliver(it,_,_): Contents = it
Agv.Pick-up(_,_): Contents = UNDEF

CAUSALITY

$ctrl.Run(\_,\_) \xrightarrow{\Diamond} Work\text{-}done$

CAPABILITY

$\mathcal{F}$ (*Work-done* / *Contents* = UNDEF)

COOPERATION CONSTRAINTS

ACTION PERCEPTION

$\mathcal{I}$ (*Ctrl.Run(m,pgm)* / $m \neq$ SELF $\vee$ *Contents* = UNDEF $\vee$ *Busy* $\vee \neg$ *In(Capabilities,pgm)*)
$\mathcal{XK}$ (*Agv.Deliver(_,_,xit)* / $\neg$ *Busy* $\wedge$ *Contents = xit*)
$\mathcal{XK}$ (*Agv.Pick-up(it,_)* / *Contents = it* $\wedge \neg$ *Busy*)

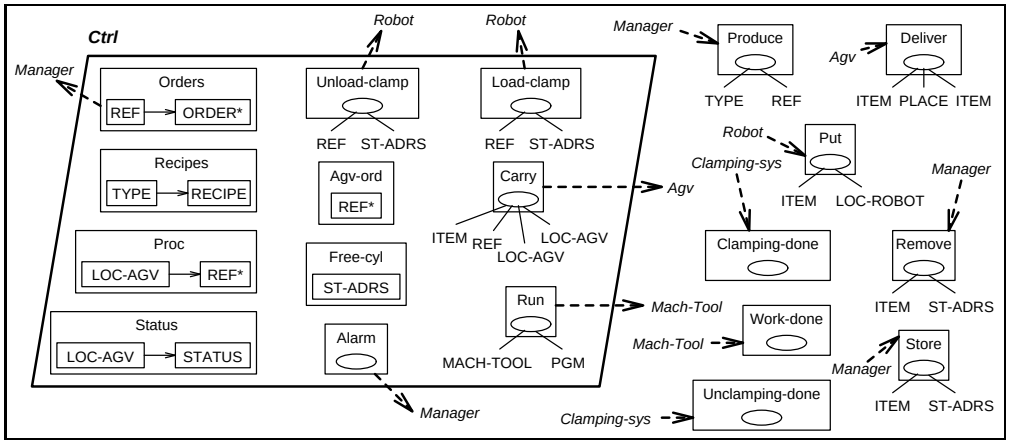
STATE PERCEPTION

$\mathcal{K}$ (*Agv.Location* / TRUE)

STATE INFORMATION

$\mathcal{XK}$ (*Contents.Agv* / *Agv.Location* = SELF)

A.6 The Controller



Ctrl

BASIC CONSTRAINTS

INITIAL VALUATION

Free-cyl = { }
Orders[_] = UNDEF
Status[_] = *Free*
Proc[_] = UNDEF
Agv-ord = UNDEF

LOCAL CONSTRAINTS

STATE BEHAVIOUR

$Orders[ref] \neq \text{UNDEF} \Longrightarrow \Diamond_{\leq 20'}\ Orders[ref] = \text{UNDEF}$
$\blacksquare_{=10'}\ Status[m] = Working \Longrightarrow Status[m] = Down$

EFFECTS OF ACTIONS

Load-clamp(ref,c): $Proc[Clamping\text{-}sys] = ref$
$Loc(Orders[ref]) = Clamping\text{-}sys$
$Free\text{-}cyl = Remove(Free\text{-}cyl,c)$

Carry(ref,_,_,c2): $Loc(Orders[ref]) = Agv$
$Status[c2] = Booked$
$Agv\text{-}ord = ref$

Run(m,_): $Status[m] = Working$
$Tdl(Orders[Proc[m]]) = Tail(Tdl(Orders[Proc[m]]))$

Alarm(m): $Status[m] = Down$

Unload-clamp(ref,_): $Orders[ref] = \text{UNDEF}$

Manager.Store(_,c): $Free\text{-}cyl = Add(Free\text{-}cyl,c)$

Manager.Remove(_,c): $Free\text{-}cyl = Remove(Free\text{-}cyl,c)$

Manager.Produce(t,ref): $Dest(Orders[ref]) = Clamping\text{-}sys$
$Loc(Orders[ref]) = Stock$
$Tdl(Orders[ref]) = Recipes(t)$

Clamping-sys.Clamping-done: $Status[Clamping\text{-}sys] = Done$
$Loc(Orders[Proc[Clamping\text{-}sys]]) = Clamping\text{-}sys$
$Dest(Orders[Proc[Clamping\text{-}sys]]) = Mach(Head(Tdl(Orders[Proc[Clamping\text{-}sys]])))$

Agv.Deliver(_,_,_): $Status[Loc(Orders[Agv\text{-}ord])] = Free$
$Proc[Loc(Orders[Agv\text{-}ord])] = \text{UNDEF}$
$Loc(Orders[Agv\text{-}ord]) = Dest(Orders[Agv\text{-}ord])$
$Proc[Dest(Orders[Agv\text{-}ord])] = Agv\text{-}ord$
$Agv\text{-}ord = \text{UNDEF}$

m.Work-done with $Tdl(Orders[Proc[m]]) = [\,]$: $Status[m] = Done$
$Dest(Orders[Proc[m]]) = Clamping\text{-}sys$

m.Work-done with $Tdl(Orders[Proc[m]]) \neq [\,]$: $Status[m] = Done$
$Dest(Orders[Proc[m]]) = Mach(Head(Tdl(Orders[Proc[m]])))$

Clamping-sys.Unclamping-done: $Status[Clamping\text{-}sys] = Done$

Robot.Put(_,_): $Status[Clamping\text{-}sys] = Free$
$Proc[Clamping\text{-}sys] = \text{UNDEF}$

CAUSALITY

$Manager.Produce(t,ref) \xrightarrow{\Diamond}$
Load-clamp(ref,c) ;
Carry(ref,_,Clamping-sys,Mach(Recipes[t][1])) ;
$\forall\ i\colon 2 \leq i \leq Length(Recipes[t])$
{ *Run(Mach(Recipes[t][i-1]),Prog(Recipes[t][i-1])* ;
Carry(ref,_,Mach(Recipes[t][i-1]),Mach(Recipes[t][i])) ;
}
Run(Mach(Recipes[t][Length(Recipes[t])]),Prog(Recipes[t][Length(Recipes[t])])) ;
Carry(ref,_,Mach(Recipes[t][Length(Recipes[t])]),Clamping-sys) ;
Unload-clamp(ref,c)

CAPABILITY

$\mathcal{F}$ (*Load-clamp(ref,c)* / $\neg\ In(Free\text{-}cyl,c) \vee Loc(Orders[ref]) \neq Stock$
$\vee\ Dest(Orders[ref]) \neq Clamping\text{-}sys \vee Status[Clamping\text{-}sys] \neq Free$)

$\mathcal{F}$ (*Carry(ref,_,c1,c2)* / $Proc[c1] \neq ref \vee c1 = c2 \vee Loc(Orders[ref]) \neq c1 \vee Dest(Orders[ref]) \neq c2 \vee Status[c1] \neq Done$
$\vee\ In(\{Working,Booked,Down\},Status[c2]) \vee Agv\text{-}ord \neq \text{UNDEF}$)

$\mathcal{F}$ (*Run(m,pgm)* / $Dest(Orders[Proc(m)]) \neq m \vee Loc(Orders[Proc(m)]) \neq m$

$\vee$ *Prog(Head(Tdl(Orders[Proc[m]]))) $\neq$ pgm* $\vee$ *Status[m] $\neq$ Booked*)
$\mathcal{F}$ (*Alarm(m)* / *Status[m] $\neq$ Working*)
$\mathcal{F}$ (*Unload-clamp(ref,_)* / *Loc(Orders[ref]) $\neq$ Clamping-sys* $\vee$ *Dest(Orders[ref]) $\neq$ Clamping-sys*
$\vee$ *Proc[Clamping-sys] $\neq$ ref* $\vee$ *Status[Clamping-sys] $\neq$ Booked*)

COOPERATION CONSTRAINTS

ACTION PERCEPTION

$\mathcal{XK}$ (*Manager.Remove(p)* / *Type(p) = Cyl*)
$\mathcal{XK}$ (*Manager.Store(p)* / *Type(p) = Cyl*)
$\mathcal{XK}$ (*Manager.Produce(t,ref)* / *t $\neq$ Cyl* $\wedge$ *Free-cyl* $\neq$ {} $\wedge$ *Orders[ref]* = UNDEF)
$\mathcal{K}$ (*Clamping-sys.Clamping-done* / TRUE)
$\mathcal{K}$ (*Agv.Deliver(_,_,_)* / TRUE)
$\mathcal{K}$ (*_.Work-done* / TRUE)
$\mathcal{K}$ (*Clamping-sys.Unclamping-done* / TRUE)
$\mathcal{XK}$ (*Robot.Put(_,_)* / *Status[Clamping-sys] = Done*)

ACTION INFORMATION

$\mathcal{O}$ (*Load-clamp(_,_).Robot* / TRUE)
$\mathcal{O}$ (*Unload-clamp(_,_).Robot* / TRUE)
$\mathcal{O}$ (*Carry(_,_,_,_).Agv* / TRUE)
$\mathcal{XO}$ (*Run(m1,_).m2* / *m1 = m2*)
$\mathcal{O}$ (*Alarm(_).Manager* / TRUE)

STATE INFORMATION

$\mathcal{O}$ (*Orders.Manager* / TRUE)