Elicitating and Formalising Requirements for C.I.M. Information Systems

Eric Dubois, Philippe Du Bois and Michaël Petit

Institut d'Informatique
Facultés Universitaires de Namur
Rue Grandgagnage, 21
B-5000 Namur (Belgium)
{edu, pdu, mpe} @info.fundp.ac.be

ABSTRACT: It is now widely recognised that *methods* are needed for covering the whole lifecycle of Computer Integrated Manufacturing (CIM) applications. In this paper, we deal with the requirements engineering phase of this lifecycle. A formal language is introduced for capturing real-time requirements expressed on CIM applications. The resulting specification is structured as a society of 'agents', each of them being characterised by (i) its responsibility with respect to changes happening in the system and (ii) the perception of the behaviour of other agents. On top of illustrating the use of the language on a small case-study, we also suggest some methodological guidance in the elaboration of requirements for large CIM applications.

KEYWORDS: O-O requirements engineering, agents, actions, formal language, CIM applications, elaboration of the requirements document.

1 Introduction

For some years, we are designing and experimenting a *formal* specification language for capturing requirements expressed on real-time complex systems. In this paper, our aim is to illustrate the use of this language by presenting preliminary results based on a real-size case study carried out in the CIM department of the CRP-HT in Luxembourg.

CIM Information System. As it is defined in [Dou90], CIM (Computer Integrated Manufacturing) stands for a "global methodological approach in the enterprise in order to improve the industrial performances". The improvement of these performances (including productivity aspects, cost reduction, fulfilment of due dates, flexibility of the manufacturing system) can only be achieved by adopting an *integrated* view of the different activities performed in the enterprise, ranging from the product design to sales management.

It is recognised that part of this integration is guaranteed by examining and controlling the complex information exchange taking place between the different activities. This led to the identification of a need for information management [Bra88, Sch88] that resulted in the implementation of *Information Systems* (IS). IS for CIM applications are characterised by :

- the mass of information to be managed and the possibly highly structured nature of the information (see, e.g., the lay-out of a product);
- the different levels of abstraction in the information. For example, information needed for the control of a factory is the aggregation of information resulting from the control of each individual equipments being part of the factory;
- the temporal (real-time) aspect of information (e.g., the control of a production machine has to be based on information messages delivered on time).

The intrinsic complexity of a CIM IS led the specialists to the conclusion that *methods* covering the whole development and maintenance life cycle of CIM IS are definitively required (not surprisingly, some years ago, a similar conclusion was drawn for the development of IS in business application systems). As a response to this need, several important projects (see, e.g., the CIM-OSA Esprit project) are investigating and designing such methods.

Formal Requirements Engineering. Requirements Engineering (RE) is now widely recognised as an essential and critical activity in the IS life cycle. This activity, which is concerned with the elicitation and the modelling of customers' needs, starts from informal wishes and elaborates a *requirements document* where the IS to be developed is defined in a precise way. Two basic trends can be identified in RE recent researches :

1. On the one hand, at the *product level* (i.e. the requirements document content), several authors plead for the use of *formal* specification languages supporting a precise interpretation (i.e. a mathematical model) of expressed requirements and advanced semantic checks through the deductive power associated with the language. Examples of such languages include RML [GBM86], GIST [Fea87], MAL [FP87], ERAE [DHR91], OBLOG [SSE89] and TELOS [MBJK90]).
2. On the other hand, at the *process level* (i.e. the set of actions applied by the analyst when he/she elaborates the requirements document). It is widely recognised that this document should include not only requirements on the IS to be implemented but also on the environment around the IS as well as on the nature of the interactions taking place between both. The emphasis on a "world oriented" or "closed system" view [Bub83, Dub89, MBJK90, Bjø92] led to the elaboration of specification for so-called *composite systems* [Fea89], i.e. systems made of multiple heterogeneous components. For example, in the CIM context, one may think about requirements on an automated production system structured according to different components like, e.g., the storage sub-system, the conveyance sub-system, the machine tool and the controller sub-system (the software piece).

Including these two aspects, under the auspices of an Esprit II project called Icarus, we have developed a formal requirements specification language called *ALBERT* (an acronym for *an Agent-oriented Language for Building and Elicitating Requirements for real-Time systems*). Some specificities of this language are :

- the possibility of structuring requirements on composite systems in terms of *agents* (like, e.g., a robot or a controller system), each agent having some contractual responsibilities with respect to the information it manages, accesses and modifies in the system;

- the existence of *structuring* mechanisms (based on parameterization and inheritance) which ensure (i) a better structuring of the requirements document (by, in particular, defining a specific vocabulary for some application domain) and (ii) the reuse of existing specification fragments.

Requirements for CIM applications are now usually modelled using classical approaches based on SADT/IDEF0 (see, e.g., the ICAM project [HDM88]) or MERISE (see, e.g., the M* project [LVB87]) or a combination of these approaches. New approaches are now emerging with the objectives of (i) increasing the degree of expressiveness, i.e. covering a wider range of requirements (see, e.g., the CIM-OSA project [GQVV90]) or (ii) using formal languages for expressing requirements (see, e.g., the OLYMPIOS project [BHM90]). We follow the same objectives in this paper. In Sect. 2, we introduce the "in-the-small" facilities offered by the ALBERT language by considering its expressiveness and its degree of formality through the handling of a fragment of an automated production control case study. Then, in Sect. 3, we report preliminary results gathered through the performance of a real-size case study from which it results that: on the one hand, we suggest how ALBERT favours the application of an Object-Oriented paradigm, on the other hand, we propose a preliminary draft for the blueprint of a *generic* architecture for CIM applications is identified. Finally, in Sect. 4, we conclude by indicating some of current research directions.

2 An Agent-oriented Requirements Specification Language

The ALBERT language ([DDP93]) is inspired by the experience that some of the authors had with the ERAE language, a language that was proposed some years ago for dealing with specifications of real-time systems [DH87, DHR91] using an appropriate *temporal logic* (an extension of first-order logic, see, e.g., [GB91]) for describing admissible histories of a system. In ALBERT, three extensions are taken into account:

1. the introduction of **agents**. That concept can be seen as a specialization of the concept of *object* and plays a key role both at the *product* and at the *process* level:
   - on the one hand, its introduction can 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 some part of the global behaviour of the whole system;
   - on the other hand, properties attached to an agent provide guidelines for the analyst who is in charge of designing the composite system; different contractual responsibilities having to be assigned to the different agents. The ALBERT language makes possible to reason on the agent responsibilities (i) for *changes* happening in the system and (ii) for the *perception* guaranteed by an agent to another one.
2. the introduction of **actions** to overcome the well-known *frame* problem [HR92], a typical problem resulting from the use of a declarative specification language;
3. the identification of **typical patterns of constraints** which support the specifier in writing and structuring complex and consistent first-order formulas. In particular, typical patterns of formulas are associated with actions.

Using the language involves two activities:

- writing *declarations* introducing the vocabulary of the considered application. Declarations are given using a graphical syntax (having a textual counterpart);
- expressing *constraints*, i.e. logical statements which identify possible behaviours of the system and exclude unwanted ones. The expression of constraints is purely textual.

Both activities are illustrated all along the rest of this section by considering a simplified fragment of an automated production control application.

Requirements are associated with a specific *cell* being part of a complex production system. This cell is in charge of the production of a *bolt* when a production request is issued. A bolt results from the transformation of a *rivet* through a given process. More specifically, the cell (see Fig.1) is made of:

- the **rivets stock**. Rivets are produced by another cell and put in this stock waiting for their processing;
- the **bolts stock**. Bolts are the resulting products of the cell activity and are stored in a stock. Bolts remain in the stock until their use for activities performed in other cells;
- the **lathe**. It is the machine transforming a rivet into a bolt by producing a thread on the rivet;
- the **robot** equipped with a gripper. Its role is twofold: on the one hand, it transports a rivet from its stock to the lathe machine; on the other hand, it transports a bolt from the lathe which produces it to the stock of bolts.

Finally, there is a performance constraint imposing that a bolt must be produced within 10 minutes.

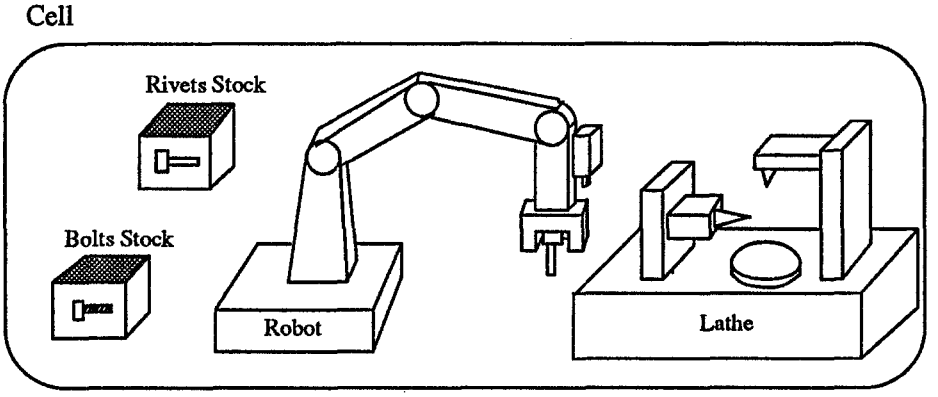

Fig. 1. Automated production control application – The Cell

2.1 Declarations

In the specification of *composite systems*, declarations identify the *agents* together with their associated *state structure.*

Agents Hierarchy. As we noted in Sect.1, the specification of a composite system is made of the specification of several agents. To be more precise, we propose to organise them in terms of a *hierarchy* where we distinguish between:

- *individual agents* corresponding to leaves in the hierarchy. These agents are *terminal* components for which a designer has to guarantee a valid implementation of its associated responsibilities;
- *complex agents* corresponding to non-terminal nodes in the hierarchy and made of finer agents. These agents are *virtual* components because they have no existence *per se* and are just aggregates of finer agents. The introduction of complex agents favours the introduction of specifications at different levels of abstraction.

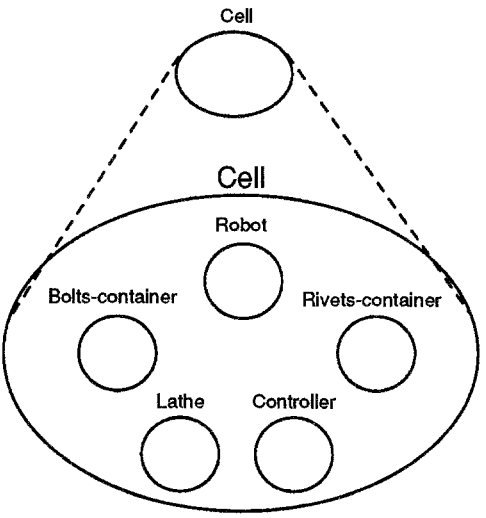

Fig. 2. Refinement of the *Cell* agent

Figure 2 proposes the graphical declaration associated with the case study where:

- *Cell* is a complex agent attached to the description of the automated production *problem.* The introduction of this *Cell* agent makes possible to introduce an intermediate level in the requirements specification document where a more abstract description of the problem can be achieved. At this level of abstraction, *Cell* is characterised as an agent in charge of transforming rivets into bolts upon requests (at this level, we have not to provide any further detail on the process according to which the transformation is performed);

- *Rivets-Container*, *Bolts-Container*, *Robot*, *Lathe*, and *Controller* are the five terminal agents cooperating together to provide a *solution* to the automated production *problem*. The last three agents play an active role. This is especially the case for the *Controller* which is in charge of synchronising and controlling the whole production process. Conversely, the two first agents (viz the *Rivets* and *Bolts containers*) have a more passive role restricted to stock keeping. At this finer level of specification, it is essential to guarantee that the abstract behaviour of the *Cell* agent is met by the interactions taking place between the five finer agents.

The existing hierarchy among agents is expressed in term of two combinators: Cartesian Product and Set. In our specific case, the *Cell* is an aggregate (tuple) of *Rivets-Container*, *Bolts-Container*, *Robot*, *Lathe*, and *Controller*.

In the sequel, we will further detail the role of the *Cell* agent (the complex agent) and the role of the *Controller* (one of the five terminal agents). For lack of place, specifications associated with the four other terminal agents are not introduced.

Actions and State Structure. The declaration part of an agent consists in the description of its state structure and the list of actions happening all along its history. In the sequel, these concepts are only briefly outlined. Further details can be found in [DDP93].

Figures 3 and 4 propose graphical diagrams (extended ERAE diagram) associated with the declaration of the structure of the *Cell* and *Controller* agents.

The structure of a state is defined in terms of *entities* (which can be grouped in populations or be individuals), *values* (which are used for characterising attributes of entities) and *relationships* between entities [Che76]. On top of these usual concepts, we are also using *data types* which correspond to :

1. predefined data types (like, *STRING*, *BOOLEAN*, *INTEGER*,...) equipped with their usual operations;
2. more complex types built by the specifier using a set of predefined type constructors (see, e.g. [BJ78]) like set, list, cartesian product,etc.

Data types are used for denoting :

- the type of value attributes;
- when required, the type of the *surrogate* used for identifying entities. This surrogate is a key mechanism that allows to refer to an agent component. For example, on Fig. 4, the individual entity *Ctrl Robot* is referring to the *Robot* agent through the surrogate mechanism. A data type is automatically associated with each class of agents. For example, each *Cell* agent has an identifier defined on type *CELL*.
  When an agent is unique (like, e.g., the *Environment* agent), then a constant is also automatically defined to refer to the identifier of that agent (*envt* in our case study).
  Inside the description of an agent, the *self* constant refers to the identifier of the described agent.

Populations of entities are denoted using squared boxes, individual entities are denoted with dashed boxes. Data types are written in upper-cases.
Actions are graphically depicted with ovals. In our terminology, we use the word 'action' both for denoting:

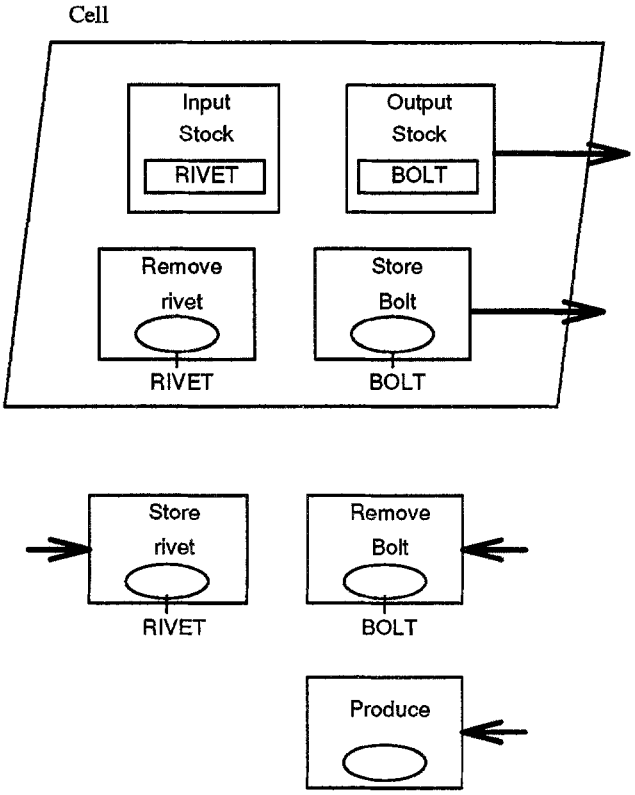

Fig. 3. Declaration associated with the *Cell* agent

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

Actions can have arguments belonging to data types (and thereby possibly referring agents surrogates).
Diagrams also include graphical notations making possible (i) to distinguish between internal and external actions and state components and (ii) to express the visibility guaranteed by the agent to the outside:

- Information within the parallelogram is under the control of the described agent while information outside the parallelogram is the perception that the agent may have with respect to the other agents of the system.
- For information in the parallelogram, boxes without arrow indicate that this information is not visible from the outside. Conversely, boxes with arrow denote information that can be perceived from the outside.

From the graphical declaration depicted on Fig.3, it can be read that the *Cell* state is

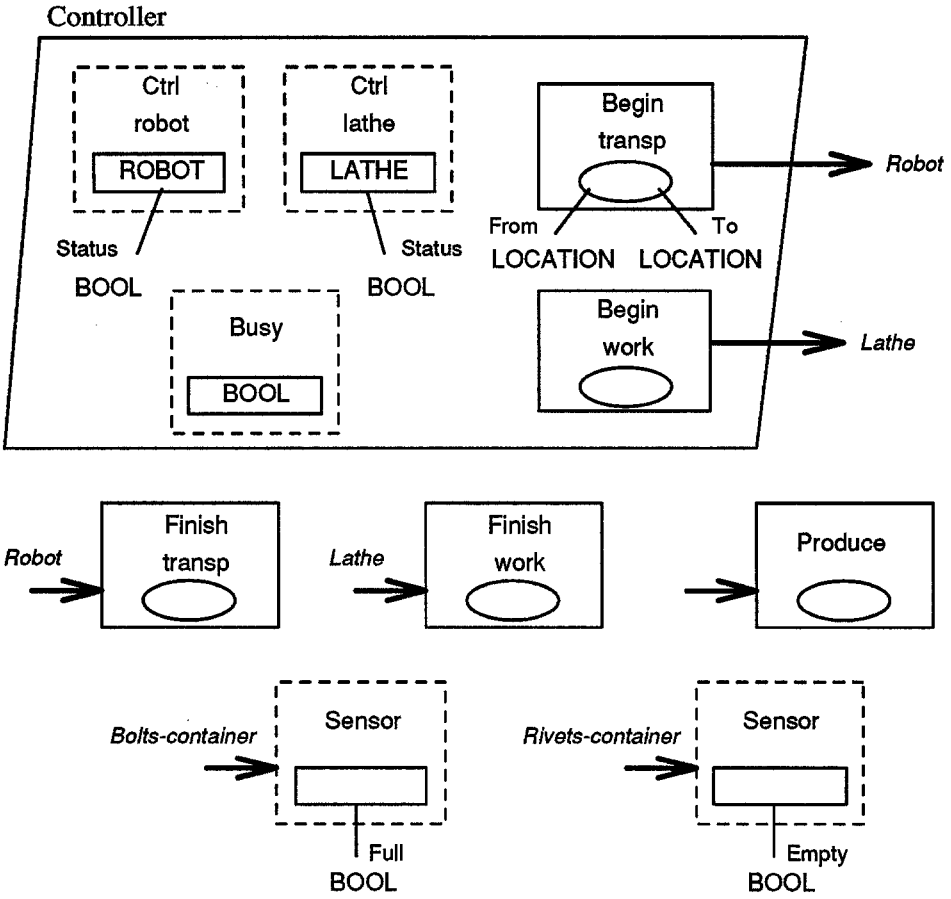

Fig. 4. Declaration associated with the *Controller* agent

made of:

1. two populations of entities respectively associated with the stock of rivets (*Input stock*) and the stock of bolts (*Output stock*). The information about the *Output stock* can be perceived from the outside (at this level, the structure of the environment is unknown);
2. two internal actions (*Remove-Rivet* and *Store-bolt*) for which the *Cell* has the initiative. The former corresponds to the withdrawal of a *rivet* from the *Input stock*, while the latter is associated with the storage of a *bolt* in the *Output stock*. Moreover, the *Cell* lets the outside having the perception of *Store-bolt* actions;
3. three actions (*Produce*, *Store-rivet*, *Remove-bolt*) are perceived by the *Cell* but have an external initiative. The *Produce* action is associated with the order of producing a new *bolt*, the *Store-rivet* action denotes the storage of a *rivet* in the *Input-stock* and the *Remove-bolt* action is related with the withdrawal of a *bolt* from the *output stock*.

The graphical declaration depicted on Fig.4 is related to the the *Controller* state structure, one of the five agents resulting from the refinement of the *Cell* complex agent. From this figure, it can be read that:

1. two individual entities (*Ctrl Robot* and *Ctrl Lathe*) denote the controlled robot and lathe. Their *status* attributes are maintained by the *Controller* for mirroring the status of the corresponding *Cell* components;
2. two actions are issued by the *Controller*. The first one (*Begin-transport*) is a transportation order sent to the *Robot* agent. The second one (*Begin-work*) is an order sent to the *Lathe* machine;
3. the cell perceives three different external actions: the *Finish-Transp* action issued by the *Robot* agent, the *Finish-Work* action issued by the *Lathe* machine agent and the *Produce* order issued by the environment (this action was already mentionned in the description of the *Cell* agent);
4. the cell perceives also the status of the two *containers* thanks to the visibility of a *sensor* associated with each of them.

2.2 Constraints

Each agent is defined by a set of possible lives modelling all its possible behaviours. A life is an (in)finite sequence of states and actions. Each state (structured in terms of *entities*) is labelled by a time value which increases all along the life. Actions occur between two successive states and can be simultaneous. On Fig. 5, a possible life of the *Cell* agent is presented.

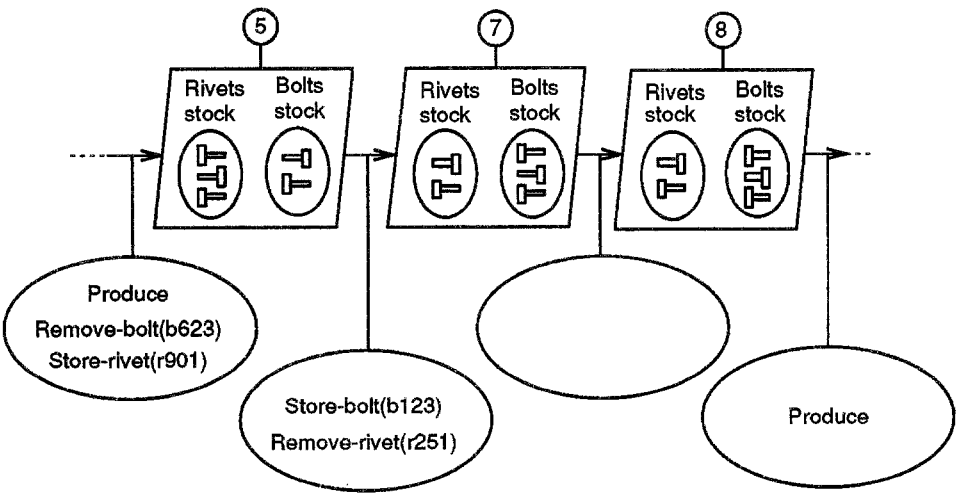

Fig. 5. Possible life of the *Cell* agent

Cell

STATE BEHAVIOUR

$\neg$ *Empty(Input-stock)*
In(Input-stock,r) $\Longrightarrow \Diamond \neg$ *In(Input-stock,r)*
* The input stock can never be empty.
* A rivet cannot stay indefinitely in the input stock.

EFFECTS OF ACTIONS

Remove-rivet(r): Input-stock = Remove(Input-stock,r)
Store-bolt(b): Output-stock = Add(Output-stock,b)
envt.Remove-bolt(b): Output-stock = Remove(Output-stock,b)
envt.Store-rivet(r): Input-stock = Add(Input-stock,r)
* Remove-rivet and Store-bolt are two internal actions;
* Remove-bolt and Store-rivet are two external actions brought by
* the outside (denoted with *envt* which stands for Environment).

COMMITMENTS

envt.Produce $\overset{\Diamond \leq 10 min}{\longrightarrow}$ *Remove-rivet(r); Store-bolt(b)*
* The Produce action (brought by the outside) has to be followed by a unique occurrence of
* the Remove-rivet action which is itself followed by a unique occurrence of
* the Store-bolt action.
* The Store-bolt action must happen less than 10 minutes after the corresponding
* Produce occurrence.

AGENTS RESPONSIBILITIES

F(envt.Remove-bolt / Empty(Output-stock))
X(envt.Produce / $\neg$ *Empty(Input-stock))*
X(Output-stock.envt / $\neg$ *Empty(Output-stock))*
* The Cell agent does not let the environment removing a bolt when the stock is empty.
* The Cell agent does not let the environment issuing a Produce order when the input
* stock is empty.
* The Environment has no access to the Output stock when it is empty.

Fig. 6. Constraints on the *Cell* agent

Constraints are used for pruning the (usually) infinite set of lives. The formal expression of constraints is based on multi-sorted first order logic, based on the concepts of variables, predicates and functions. In order to master the complexity of writing first-order formulas, we have grouped them under different headings and have considered the introduction of some specific connectives for expressing them.

On Fig. 6 and 7, we present formal specifications associated with the *Cell* complex agent and the *Controller* terminal agent. In a systematic way, on these figures, we have added informal comments to help in reading formal constraints. Hereafter, we detail the nature of the different kinds of constraints.

State Behaviour. The possible configurations of states can be restricted by constraints. These constraints are written according to the usual rules of strongly typed first order logic. In particular, they are formed by means of logical connectives $\neg$ (not), $\wedge$ (and), $\vee$ (or), $\Rightarrow$ (implies), $\Leftrightarrow$ (if and only if), $\forall$ (for all), $\exists$ (there exists). The outermost universal quantification of formulas can be omitted. The rule is that any variable, which is not in the scope of a quantifier, is universally quantified outside of the formula.

On top of constraints which are true in all states (usually referred as *invariants*), there are constraints on the evolution of the system expressing constraints on the ordering of states (like, e.g., if this property holds in this state, then it holds in all future ones) or referring states at the different times. Such constraints are written in the language by using additional *temporal* connectives which are prefixing statements to be interpreted in different states. The following table introduces these operators (inspired from temporal logic, see e.g. [Ser80, TLW91]) and their intuitive meaning (ϕ and ψ are statements):

$\Diamond\ \phi$ ϕ is true sometimes in the future (including the present)
$\blacklozenge\ \phi$ ϕ is true sometimes in the past (including the present)
$\Box\ \phi$ ϕ is always true in the future (including the present)
$\blacksquare\ \phi$ ϕ is always true in the past (including the present)
$\phi\ \mathcal{U}\ \psi$ ϕ is true from the present until ψ is true (strict)
$\phi\ \mathcal{S}\ \psi$ ϕ is true back from the present since ψ was true (strict)

There are constraints related to the expression of real-time properties. They are needed to describe delays or time-outs and are expressed by subscripting temporal connectives with a time period. This time period is made precise by using predefined functions that can be used to model the usual time units: *Sec*, *Min*, *Hours* and *Days* [KVdR89].

Effect of actions. These constraints relate the occurrence of an action to the modifications brought in the current state of an agent. Only actions which bring a traceable change are described.

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 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 agent is an external one, the name of the agent is prefixing the action) and (iii) the previous state in the history.

Controller

STATE BEHAVIOUR

$\neg$ *(Status(Ctrl-lathe))* $\wedge$ *Status(Ctrl-robot))*
* The lathe equipment and the robot equipment cannot be busy at the same time.

(Status(Ctrl-lathe) $\vee$ *Status(Ctrl-robot))* $\Longrightarrow$ *Busy*
* The Cell is said busy when the status of the robot or of the lathe is busy.

EFFECTS OF ACTIONS

Begin-transp(o,d): Status(Ctrl-robot) = true
Begin-work: Status(Ctrl-lathe) = true
envt.Produce: Busy = true
robot.Finish-transp(o,d): Status(Ctrl-robot) = false
robot.Finish-transp(lathe,bolts-shelf): Busy = false
lathe.Finish-work: Status(Ctrl-lathe) = false

COMMITMENTS

$$envt.Produce \overset{\Diamond \leq 10min}{\longrightarrow} \begin{array}{l} Begin\text{-}transp(rivets\text{-}shelf,lathe); \\ Begin\text{-}work; \\ Begin\text{-}transp(lathe,bolts\text{-}shelf) \end{array}$$

AGENTS RESPONSIBILITIES

F(Begin-transp(o,d) / Status(Ctrl-robot) $\vee$ $\neg$ *Busy)*
F(Begin-work / Status(Ctrl-lathe) $\vee$ $\neg$ *Busy)*
X(envt.Produce / $\neg$ *Busy* $\wedge$ $\neg$ *Empty(Sensor)* $\wedge$ $\neg$ *Full(Sensor))*
X(robot.Finish-transp / Busy $\wedge$ *Status(Ctrl-robot))*
X(lathe.Finish-work / Busy $\wedge$ *Status(Ctrl-lathe))*

Fig. 7. Constraints on the *Controller* agent

Commitments. 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].

The "$\longrightarrow$" symbol can be quantified by a temporal operator to express performances constraints (e.g. the "$\overset{\Diamond \leq 10min}{\longrightarrow}$" symbol means that the occurrence of an (re)action has to happen within a 10 minutes interval after the occurrence of the action).

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. 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* $\|$ *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)".

Agents Responsibilities. Under this heading, we make precise the role of the agent with respect to the occurrence of actions and to the visibility it offers to the outside. 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. Three specific *deontic* connectives are considering for expresssing *obligations*, *preventions* and *exclusive obligations*.

The pattern for an obligation "*O(<action>/<situation>)*" expresses that the action has to occur if the circumstances stated in the situation are met (these circumstances refer to conditions on the current or previous states). For external actions, an obligation defines the circumstances where, if such an action is issued by the external agent, the behaviour of the current agent state is influenced.

The pattern for a prevention "*F(<action>/<situation>)*" expresses that the action is forbidden when the circumstances expressed in the situation are matched. For an external action, a prevention defines the circumstances where, if such an action is issued by the external agent, it has no influence on the current agent behaviour.

The pattern for an exclusive obligation is "*X(<action>/<situation>)*" which is a shorthand for "*O(<action>/<situation>)* and *F(<action>/¬ <situation>)*".

The default rule is that all actions are *permitted* whatever the situation. The semantics associated with our deontic connectives is similar to the one considered by von Wright in his pioneering work on Deontic Logic [vW68].

Using these connectives makes possible to express the control that the agent has with respect to actions. At this level, our objective is somewhat similar to the one of [KM87] and [FM90]. We express the conditions under which an action issued by an external agent (like the *Store-rivet* action issued by the *Environment*) or by the agent itself (like the *Remove-rivet* action) may affect the agent history. For example, from the first statement under the responsibility heading on Fig.6 it can be read that the *Cell* agent does not let the *Environment* agent removing a bolt when the stock is empty.

These connectives are also used to bring restrictions on the visibility of states fragments or occurrences of actions to the outside. For example, the last responsibility statement on Fig.6 expresses that the *Environment* agent has no access to the *Output-stock* component of the *Cell* agent state when that stock is empty.

Perception offered by an agent to another one may vary with time. Agents responsibilities constraints are used to express this dynamical aspect.

Whatever the nature of the agent (complex or terminal), patterns of constraints are similar but play a different role in the system development life cycle:

- at the level of a terminal agent (like, e.g., the *Controller*), the specification is the starting point of the *design engineering* process where a designer will be in charge of implementing a component guaranteeing the prescribed behaviour of the agent;
- at the level of a complex agent (like, e.g., the *Cell*), the specification is an intermediate step in the process followed by the analyst. Analogously to [Dub89, DFHF91], this specification is defining the *goal* associated with the system considered as a whole and the task of the analyst is to follow a *refinement process* resulting in the description of a composite system where the combination of specifications attached to individual agents meets the goal identified. In our case study, this means that, in the final requirements document, the *Cell* goal of "*producing a bolt within 10 minutes after a request*" is achieved through the combination of the individual actions made by the *Controller*, *Robot* and *Lathe*.

3 Structuring and Reusing Requirements for CIM Applications

In the previous section, we have introduced the ALBERT language and illustrated its use on a fragment of a CIM application. Besides these "in-the-small" facilities, the language has also been used for dealing with larger CIM applications. Some conclusions of these studies are reported in this section where:

1. we discuss the elaboration of a general CIM architecture which can serve as a blueprint for the design of requirements for a specific CIM application;
2. we present a *generic* component that has been identified as a key reusable requirements specification component for building these CIM architectures.

All along this section, for the sake of brevity, only a sub-part of the formal specifications will be presented. This sub-part is related with the graphical component of the ALBERT language dealing with the declaration of *agents* and of their hierarchical structuring. For the interested reader, a complete presentation of the requirements including the declaration of the *state structure* and the expression of *constraints* may be found in [Pet92].

3.1 Towards a General CIM Architecture

CIM applications are definitively found complex because of (i) the mass of information to be managed (e.g. information about production plans, parts layouts, stocks, etc) and (ii) the different abstraction levels to be considered (e.g., relevant information at the level of the factory, at the level of each cell, at the level of each equipment, etc).

For mastering this complexity, several authors (see, e.g., [Sch88]) have proposed a general architecture for CIM applications. The objective is that this general architecture

may serve as a blueprint for the analyst being in charge of capturing requirements for specific applications. Doing so, the analyst is applying an *analogical reasoning* [SM92] by (i) identifying reusable components in the general architecture and (ii) adapting them to his/her particular context.

The description of a general CIM enterprise should result in the identification of a hierarchy (i.e. a directed acyclic graph) of specification components. We experimented that this hierarchy may be elaborated following a *functional* perspective or an *object-oriented* perspective.

The Functional Perspective. The use of classical approaches (like, e.g., SADT) leads to the elaboration of functional architectures based on:

1. at the higher level, the identification of the functional areas existing in the manufacturing company;
2. the functional decomposition of each of these areas into a set of finer functions which are chained together through data flows;
3. the recursive decomposition of these functions up to the identification of terminal functions.

Adopting such perspective leads to the identification of an architecture analogous to the one presented in Fig. 8. On this figure, the following functional areas have been considered:

- CAD (Computer Aided Design)
  This area is in charge of the design of a product. It uses information about a new product provided either by the customer or by a strategic planning department. Characteristics (like loads and tolerances) of the product are identified, technical and economical analyses are performed. An important result of this phase is the layout of the product (i.e., its decomposition into parts and the way these parts are assembled together).
- CAPP (Computer Aided Process Planning)
  This area is responsible for the elaboration of one or more recipes (called *schedules*) that can be used for producing a particular product part. It is based on information received from the CAD area about the product (and about its structure) and information on equipments features.
- PPC (Planning and Production Control)
  According to the capabilities of each production equipment and on the basis of the received orders, a production plan (including the occupation periods of each equipment) is prepared. It is established using information provided by the CAPP function about the possible schedules, information about the customer orders and the sales forecasts. Once the planned date for the production is reached, the production activities have to be started. For the different steps of the schedule, orders have to be sent to the adequate equipments following an appropriate sequence.
  The controller of the cell described in Sect. 2 implements a subset of these functions (no planning is done but the controller initiates the production activities by sending orders to the robot and the lathe machine).
- CAM (Computer Aided Manufacturing)
  This area takes in charge the execution of the different steps of the schedule.

This execution is ensured by using different equipments carrying on activities like machine-finishing, assembling, transporting, storing, packaging, ...

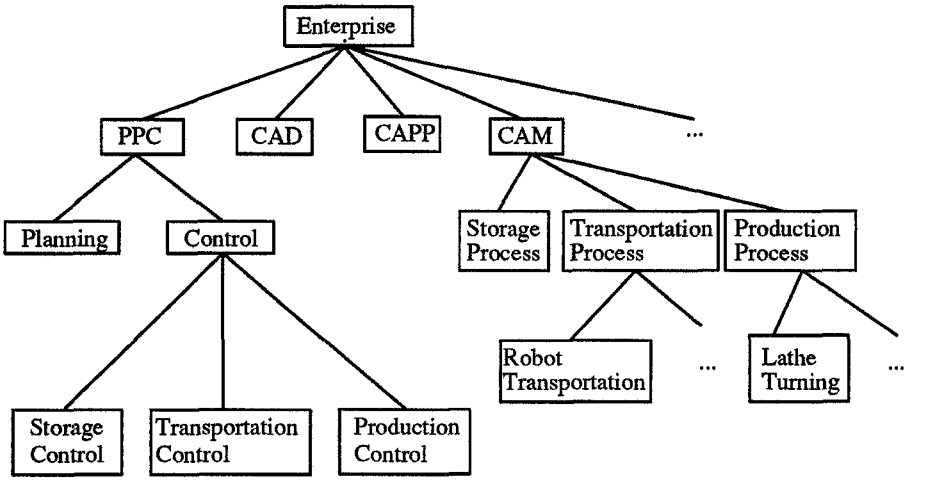

Fig. 8. An example of functional CIM architecture

The O-O Perspective. Recent researches in CIM led authors like, e.g., [Sch88, Dou90] to criticize the functional perspective and to encourage the identification of CIM architecture based on a different decomposition criterion. Similarly, in software engineering, an analogous trend has led to the adoption of an *object-oriented* perspective achieving a better cohesion around information.

In our work, we have privileged an *agent-oriented* perspective which is rather similar to the O-O one since an *agent* in the ALBERT language can be seen as a *specialization* of the usual *object* used in O-O conceptual modelling (see, e.g., OBLOG [SSE89] and O* [Bru91]). Using this agent perspective led to the identification of a hierarchy based on an *organizational* perspective where the *production process* is considered at different abstraction levels, each of them describing a *production unit*. This perspective has the consequence that functions are no longer grouped with respect to the functional area to which they belong to but are now distributed among the different production units where they are required.

On Fig. 9, one may see an example of an architecture built using an agent-oriented perspective (and expressed in terms of ALBERT declarations for agents). In this architecture, one may note that (i) there are several abstraction levels, each of them corresponding to a more or less detailed view of a level of the production process (for example, at the higher level, we consider the *Metal Machining Factory* and, at an intermediate level, we consider the *Turning Cell*) and that (ii) functional units belonging to the same functional area are distributed over the different identified production levels

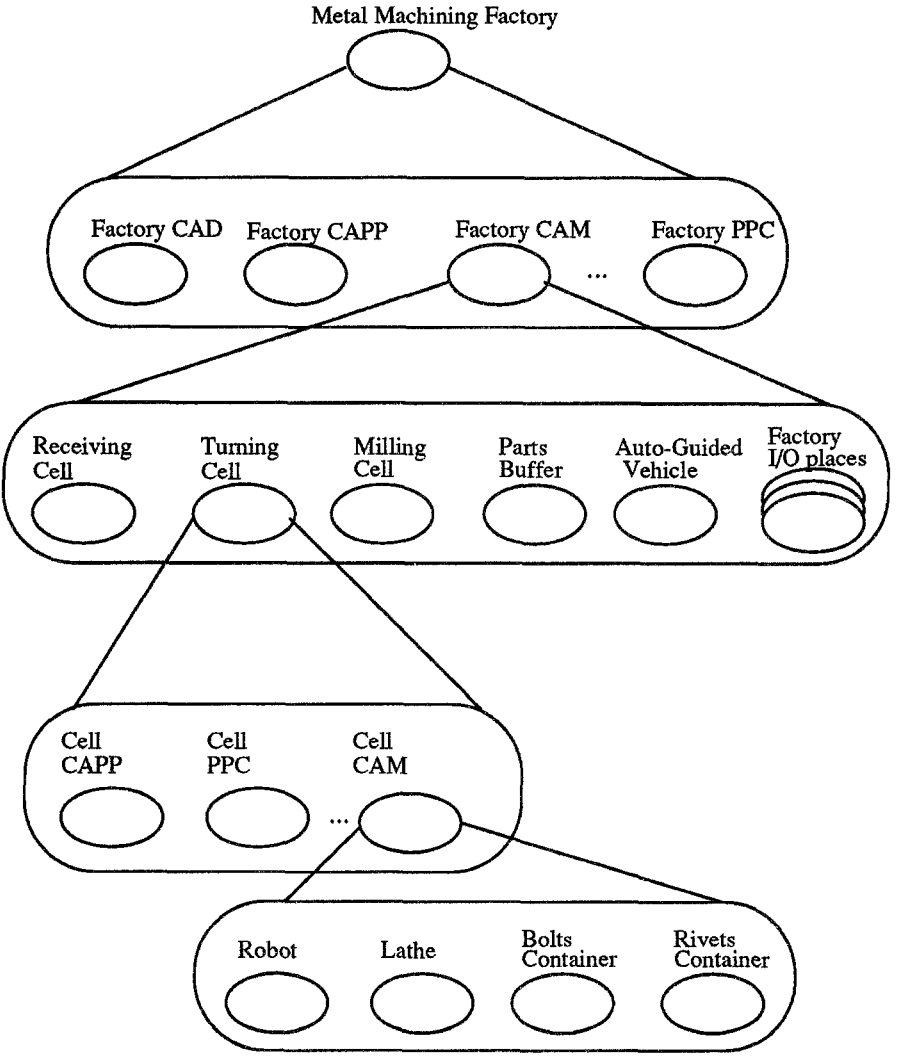

Fig. 9. An example of agent-oriented CIM architecture

(for example, a *Factory PPC* is considered at the higher level and, at a lower level, a *Cell PPC* is identified in the *Turning Cell*).

The *Turning Cell* agent described in this architecture is similar to the *Cell* agent described in Sect. 2. It is a more complete version of it where the functions of the Cell *Controller* in Sect. 2 are a subset of the *Cell PPC* functions and where the *Turning Cell* has additional capabilities (*Cell CAPP*, etc.).

3.2 Identifying a *Generic* Component

The *agent-oriented* view presented in the previous sub-section may lead to the elaboration of a CIM architecture having an arbitrarily varying number of levels according to the complexity of the *production process* considered in the manufacturing company. However, it is important to note that there are some common aspects between the different levels of the architecture. This led to the idea of identifying a *generic* component presented on Fig. 10. The role of some of the agents it contains is informally described below.

- CAM n
  The role of the CAM n component is to manufacture the product requested at this level. To do this, it consists of the following components: level-n input/output places (level-n I/O places), level-(n-1) production units, level n storage places and level n transporters.
  - The level-n I/O places are the input/output access points of the CAM level-n. They are locations where the products, entering and exiting the level n, can be placed. A level without any I/O places would be completely closed and thereby would not be able to communicate with the outside. I/O places can take different forms: a bearing area, a container, a conductor rail able to accept a pallet, ... Each kind of I/O place is dedicated to the storage of specific elements (e.g. a bare part cannot be put on a pallet conductor rail).
  - The role of a level-(n-1) production unit is to ensure a part of the production of the whole product. It can be very complex and can include many single equipments such as drilling machines, lathes, assembling stations, robots, conveyors, stocks,etc. But it is, at this level, considered as a simple unit providing services (black-box approach).
  - A level-n storage place is a location where products can be stored for a certain duration. It is very similar to an I/O place except that it is internal to the level and has no relationship with the outside.
  - The role of the level-n transporter is to transport products between two components present inside the level. Components can be of the same type (two level-(n-1) production units, two I/O places, two storage places or two transporters) or of different types (one level-(n-1) production unit and one I/O place, one storage place and one level-(n-1) production unit, etc). The type of transporter is different according to the level where it is considered. For example, at a low level (e.g. within a cell), a robot is often be used for transporting bare parts. At the factory level, Auto-Guided Vehicles (AGVs) and conveyors are often used. Trucks can be in charge of the transport between factories.
- CAD n
  The role of a level-n CAD component is (i) to identify the different parts from which the desired product is composed of, (ii) to describe, for each of them, their features (e.g., guaranteed loads, tolerances, ...) and (iii) to indicate how they have to be assembled together. The lower level parts are produced by some level-(n-1) production units present in the CAM n component and assembled by others to obtain the desired product.

- CAPP n
  The role of this component is to prepare schedules for the manufacture of products required at this level. These schedules are made of steps that will be executed by the lower levels of the CAM-n component. They also describe the order in which these steps will have to be performed.
- PPC n
  This component is responsible for the planning of the allocations of the work to the different equipments of the level-n. The resulting plan consists in the definition of occupation periods for these equipments (production units, transporters, I/O places and storage places). The PPC n component also issues the appropriate execution orders to the level n equipments and monitors the follow-up of the activities.

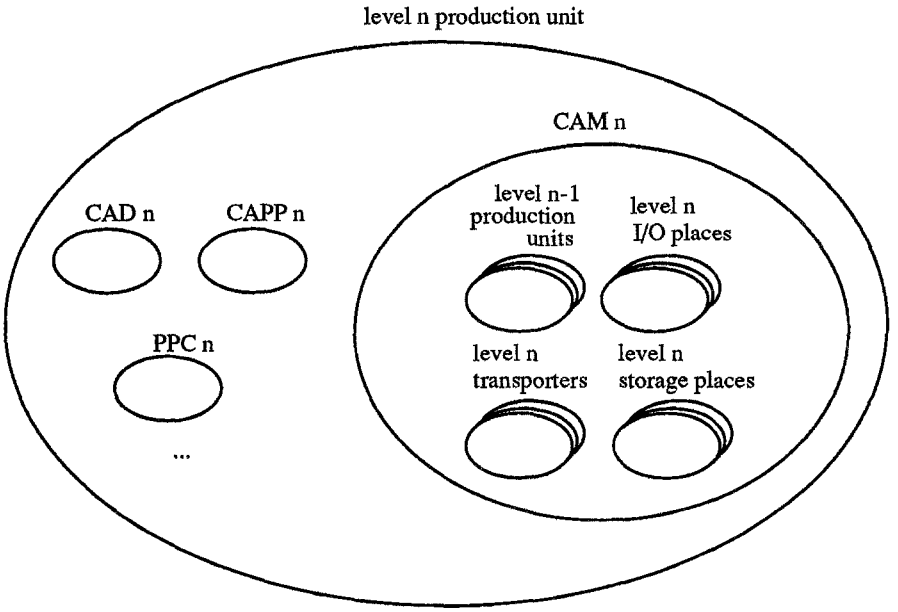

Fig. 10. Structure of the Generic component

From our preliminary experiments, it results that this generic component is central for CIM applications and thereby should take place in a library of reusable *cliches* for the CIM application domain [RW91]. Analysts can reuse it and tailor it to the needs of a particular application (a similar process was suggested by the authors in [DDR92]). For example, the two first levels of the architecture presented on Fig. 9 can be obtained by instantiating the generic component with the following parameters substitutions:

- the *Metal Machining Factory* is the *level n production unit*;
- the *Factory CAM* is the *CAM n* where:

  - the *Turning Cell*, *Receiving Cell* and *Milling Cell* are *level-(n-1) production units* (which can themselves be further decomposed by new recursive instanstiations of the generic component; see, e.g., the *Turning Cell*)
  - the *Parts Buffer* is a *level-n storage place*;
  - the *Auto-guided Vehicle* is a *level-n transporter*;
  - the *Factory-I/O-places* are *level-n I/O places*;
- the *Factory-CAD* is the *CAD n*;
- the *Factory-PPC* is the *PPC n*;
- the *Factory-CAPP* is the *CAPP n*.

4 Conclusion

For some years, we are investigating the use of *formal methods* at the Requirements Engineering (RE) level of the software life cycle. Within this context, this paper introduces the ALBERT language and provides some preliminary insights about its use in the context of Computer Integrated Manufacturing (CIM) applications. The language itself is characterized by :

- its expressiveness. Requirements on performances, actions and perceptions are structured in terms of *agents* having some contractual responsibilities for guaranteeing them. This need for introducing the concept of agent at the RE level is also emphasized in [Fea89, DFvL91, Yu93];
- its *degree of formality*. Different kinds of constraints can be expressed using some variants of typed first-order logic, viz temporal logic and deontic logic. Similar formal frameworks are also investigated in [FM90, JSS91];
- the availability of structuring mechanisms based on *parameterization* and *inheritance* [OSC89]. The use of these mechanisms is only suggested in this paper but more details can be found in [DDR92].

Using our language in the context of CIM applications led to the following researches directions :

1. the need for the identification of reusable *patterns* of requirements characteristic of this application domain. These patterns should include "in the small" facilities (like, e.g., those related to expression of different forms of *commitments*) as well as "in the large facilities" for developing blueprints of requirements fragments;
2. the investigation of *strategies* that can be followed in the incremental elaboration of a requirements document. In particular, we experimented that, due to their complexity, requirements on CIM applications cannot be built from scratch but that it can be envisaged to start from requirements on a simplified application before to move gradually towards more complex requirements. Within this framework, we are working on the development of specifications transformations related to non reliable agents and non omniscient agents;
3. finally, the study of the systematic derivation of *design specifications* from *requirement specifications* has not to be neglected. Within the context of the CIM department of the CRP-HT in Luxembourg, we consider the relationship existing between specifications written with ALBERT and design specifications expressed in OBLOG [SFS90].

Acknowledgement: This work was partially supported by the European Community under Project 2537 (ICARUS) of the European Strategic Program for Research and development in Information Technology (ESPRIT). The authors wish to thank Marc Derroitte and Robert Darimont for helpful discussions and critical comments. We are also indebted to Jean-Pol Michel for the opportunity he offered us to work on C.I.M. applications.

References

[BHM90] D. Beauchêne, A. Haurat, and J.L. Maire. Typology and modeling: a global approach in manufacturing enterprises. In *Proceedings of the international conferefence CIM'90: Integration Aspects*, pages 235–242, Bordeaux (France), June 12-14, 1990. Productic-A, Teknea.

[BJ78] D. Bjørner and C.B. Jones. *The Vienna Development Method. The metalanguage*, volume 61 of *LNCS*. Springer-Verlag, 1978.

[Bjø92] D. Bjørner. Trusted computing systems: The procos experience. In *Proc. of the 14th international conference on software engineering*, pages 15–34, Melbourne (Australia), May 11-15, 1992. IEEE, ACM Press.

[Bra88] O.H. Bray. *Computer Integrated Manufacturing (The Data Management Strategy)*. Digital Press CIM series. Hamilton Printing Company, 1988.

[Bru91] J. Brunet. Modelling the world with semantic objects. In *Proc. of the working conference on the object-oriented approach in information systems*, Québec, 1991.

[Bub83] Janis A. Bubenko. On concepts and strategies for requirements and information analysis. In *Information modeling*, pages 125–169. Chartwell-Bratt, 1983.

[Che76] P.P. Chen. The entity-relationship model: Towards a unified view of data. *ACM TODS*, 1(1):9–36, 1976.

[DDP93] Eric Dubois, Philippe Du Bois, and Michaël Petit. O-O requirements analysis: an agent perspective. In O. Nierstrasz, editor, *Proc. of the 7th european conference on object-oriented programming – ECOOP'93 (to appear)*, Kaiserslautem (Germany), July 26-30, 1993.

[DDR92] Eric Dubois, Philippe Du Bois, and André Rifaut. Elaborating, structuring and expressing formal requirements of composite systems. In P. Loucopoulos, editor, *Proc. of the 4th conference on advanced information systems engineering – CAiSE'92*, pages 327–347, Manchester (UK), May12-15, 1992. LNCS 593, Springer-Verlag.

[DFHF91] E. Doerry, S. Fickas, R. Helm, and M. Feather. A model for composite system design. In *Proc. of the 6th international workshop on software specification and design*, Milano, October 1991.

[DFvL91] A. Dardenne, S. Fickas, and A. van Lamsweerde. Goal-directed concept acquisition in requirements elicitation. In *Proc. of the 6th international workshop on software specification and design*, Milano, October 1991.

[DH87] Eric Dubois and Jacques Hagelstein. Reasoning on formal requirements: a lift control system. In *Proceedings of the 4th international workshop on software specification and design*, pages 236–241, Monterey CA, April 3-4, 1987. IEEE, CS Press.

[DHR91] Eric Dubois, Jacques Hagelstein, and André Rifaut. A formal language for the requirements engineering of computer systems. In André Thayse, editor, *From natural language processing to logic for expert systems*, chapter 6. Wiley, 1991.

[Dou90] Guy Doumeingts. Design and specification methods for production systems. In *Proceedings of the international conferefence CIM'90: integration aspects*, pages 89–103, Bordeaux (France), June 12-14, 1990. Productic-A, Teknea.

[Dub89] Eric Dubois. A logic of action for supporting goal-oriented elaborations of requirements. In *Proceedings of the 5th international workshop on software specification and design*, pages 160–168, Pittsburgh PA, May 19-20, 1989. IEEE, CS Press.

[Fea87] Martin S. Feather. Language support for the specification and development of composite systems. *ACM Transactions on programming languages and systems*, 9(2):198–234, April 1987.

[Fea89] Martin S. Feather. Constructing specifications by combining parallel elaborations. *IEEE Transactions on software engineering*, SE-15(2), February 1989.

[FM90] Jose Fiadeiro and Tom Maibaum. Describing, structuring and implementing objects. In *Foundations of Object-Oriented Languages - REX School/Workshop*, pages 275–310, Noordwijkerhout (The Netherlands), May 28 - June 1, 1990. LNCS 489, Springer-Verlag.

[FP87] Anthony Finkelstein and Colin Potts. Building formal specifications using "structured common sense". In *Proceedings of the 4th international workshop on software specification and design*, pages 108–113, Monterey CA, April 3-4, 1987. IEEE, CS Press.

[FS86] Jose Fiadeiro and Amilcar Sernadas. Linear tense propositional logic. *Information Systems*, 11(1):61–85, 1986.

[GB91] D. Gabbay and P. Mc Brien. Temporal logic and historical databases. In *Proc. of the 17th international conference on very large databases*, Barcelona, September 1991.

[GBM86] Sol J. Greenspan, Alexander Borgida, and John Mylopoulos. A requirements modeling language. *Information Systems*, 11(1):9–23, 1986.

[GQVV90] R. Gaches, B. Querenet, P. Viollet, and F. Vernadat. Cim-osa: an open system architecture. In *Proceedings of the international conferefence CIM'90: Integration Aspects*, pages 227–234, Bordeaux (France), June 12-14, 1990. Productic-A, Teknea.

[HC68] G.E. Hughes and M.J. Cresswell. *An introduction to modal logic*. Methuen and Co., London, 1968.

[HDM88] C.S. Harrison, L.G. Dove, and B. Makin. A practical approach to cim systems design. In *Proceedings of the Computer Aided Production Engineering Conference*, pages 375–380, November 1988.

[HR92] Jacques Hagelstein and Dominique Roelants. Reconciling operational and declarative specifications. In P. Loucopoulos, editor, *Proc. of the 4th conference on advanced information systems engineering – CAiSE'92*, pages 221–238, Manchester (UK), May12-15, 1992. LNCS 593, Springer-Verlag.

[JSS91] R. Jungclaus, G. Saake, and C. Sernadas. Formal specification of object systems. In S. Abramsky and T. Maibaum, editors, *Proc. of TAPSOFT'91 Vol.2*, pages 60–82, Brighton (UK), 1991. LNCS 494, Springer-Verlag.

[KM87] S. Khosla and T. Maibaum. The prescription and description of state based systems. In B. Banieqbal, H. Barringer, and A. Pnueli, editors, *Temporal logic in specification*. LNCS 398, Springer-Verlag, 1987.

[KVdR89] R. Koymans, J. Vytopil, and W. de Roever. Specifying message passing and time-critical systems with temporal logic. Doctoral dissertation, Eindhoven University of Technology, Eindhoven (The Netherlands), 1989.

[LVB87] A. Di Leva, F. Vernadat, and D. Bizier. Information system analysis and conceptual database design in production environments with m*. *Computers in Industry*, 9:183–217, 1987.

[MBJK90] J. Mylopoulos, A. Borgida, M. Jarke, and M. Koubarakis. Telos: A language for representing knowledge about information systems. *ACM Transansaction on Information Systems*, 8(4):325–362, 1990.

[OSC89] F. Orejas, V. Sacristan, and S. Clerici. Development of algebraic specifications with constraints. In *Proc. of the workshop in categorical methods in computer science*. LNCS 393, Springer-Verlag, 1989.

[Pet92] Michaël Petit. Construction et formalisation de spécifications conceptuelles pour les systèmes productiques. Master's thesis, Computer Science Department, University of Namur, Namur (Belgium), September 1992.

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

[RW91] Howard B. Reubinstein and Richard C. Waters. The requirements apprentice: Automated assistance for requirements acquisition. *IEEE Transactions on software engineering*, 17(3), March 1991.

[Sch88] August-Wilhelm Scheer. *CIM: Computer steered industry*. Springer-Verlag, 1988.

[Ser80] Amilcar Sernadas. Temporal aspects of logic procedure definition. *Information Systems*, 5:167–187, 1980.

[SFS90] Cristina Sernadas, Jose Fiadeiro, and Amílcar Sernadas. Object-oriented modelling from law. In Meersman, Shi, and Kung, editors, *The Role of Artificial Intelligence in Databases and Information Systems*. North-Holland, 1990.

[SM92] A. Sutcliffe and N. Maiden. Supporting component matching for software reuse. In P. Loucopoulos, editor, *Proc. of the 4th conference on advanced information systems engineering – CAiSE'92*, pages 290–303, Manchester (UK), May12-15, 1992. LNCS 593, Springer-Verlag.

[SSE89] A. Sernadas, C. Sernadas, and H.-D. Ehrich. Abstract object types: a temporal perspective. In B. Banieqbal, H. Barringer, and A. Pnueli, editors, *Proc. of the colloquium on temporal logic and specification*, pages 324–350. LNCS 398, Springer-Verlag, 1989.

[TLW91] C. Theodoulidis, P. Loucopoulos, and B. Wangler. A conceptual modelling formalism for temporal database applications. *Information Systems*, 16(4):401–416, 1991.

[vW68] G.H. von Wright. An essay in deontic logic and the general theory of action. *Acta Philosophica Fennica*, XXI, 1968.

[Yu93] Eric S. K. Yu. Modelling organizations for information systems requirements engineering. In A. Finkelstein, editor, *Proc. of the IEEE International Symposium on Requirements Engineering – RE'93*, pages 34–41, San Diego CA, January 4-6, 1993. IEEE Computer Society Press.