O-O Requirements Analysis: an Agent Perspective

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 : In this paper, we present a formal object-oriented specification language designed for capturing requirements expressed on composite real-time systems. The specification describes the system as a society of 'agents', each of them being characterised (i) by its responsibility with respect to actions happening in the system and (ii) by its time-varying perception of the behaviour of the other agents. On top of the language, we also suggest some methodological guidance by considering a general strategy based on a progressive assignement of responsibilities to agents.

KEYWORDS : O-O requirements analysis, agents, actions, formal language, first-order, temporal and deontic logic, elaboration of the requirements document.

1 Introduction

Requirements Analysis (also called Requirement Engineering) is now widely recognised as a critical activity in the context of software development.

Some languages where proved useful to express requirements. These include, e.g., PSL/PSA [TH77], SADT [Ros77], MERISE [TRC83] or IDA [BP83]. Languages of that family have a rigorous syntax but suffer from a lack of formal semantics. This fact leads to the development of new languages based on formal grounds like, e.g., RML [GBM86], GIST [Fea87], ERAE [DHR91] and TELOS [MBJK90].

The common characteristic of all these languages is the existence of a logical / mathematical semantics which permits unambiguous and consistent expression of requirements together with a formal framework for a rigorous investigation of the R.A. process (see, e.g. [JF90]). Differences exist between these languages and are inherent to their *expressive* power (i.e. the level of requirements that can be captured using the language) and the nature of the available *structuring mechanisms* (i.e. mechanisms which help in organizing large requirements documents).

In the last years, the O-O paradigm has been adopted at the level of programming and design specification languages. OBLOG [SSE89], Larch [GHW85], Object-Z [DKRS91] and OOZE [AG91] are examples of formal design languages supporting the O-O paradigm. Recent approaches (e.g. [CY91], [SM88]) consider the application of this paradigm at the requirements analysis level. Our paper goes along this direction but emphasises the need for formality in OORA.

We propose a formal O-O language designed for supporting requirement engineering. Our language aims at supporting the expression of (i) statements about real-world entities, (ii) performances requirements and (iii) visibility and reliability requirements. More precisely, the proposed language can be seen as an extension of the ERAE language [DHR91] complemented with:

- the concept of *agent* (as introduced for programming languages in [Sho90]). This concept, which can be seen as a specialization of the concept of *object*, is needed for structuring the requirements document according to the contractual responsibilities attached with each agent (a manual procedure, a device or a software component) with respect to the information it manages, accesses and modifies in the system;
- the concept of *action* (see, e.g., MAL [FP87]), introduced to define the role of agents w.r.t. changes happening in the system. Secondarily, the introduction of this concept is also motivated by our aim to overcome the *frame problem* existing in a declarative language [BMR92];
- the concept of *perception*, introduced to model the knowledge the agent has about its environment (i.e. the behaviour of other agents).

In Sect.2, the requirements engineering activity is first described and the concept of "composite system" is introduced.

The language itself is presented in Sect.3. It is called ALBERT (an acronym for "Agent-oriented Language for Building and Eliciting Requirements for real-Time systems") and supports the possibility of writing *graphical* declarations and complement them with *textual* constraints.

On top of a language, one definitively requires some *methodological* guidance for an incremental elaboration of a complex requirements document (see, e.g. [Fea89, FF89]). In Sect.2, we briefly describe a general strategy where the progressive elicitation of requirements is supported by identifying the *goals* of the system first before to refine them in terms of local responsibilities attached to each individual agent. In Sect.4, we illustrate how this strategy is supported in our language.

All along this paper, we will illustrate our ideas on a simplified fragment of a Computer Integrated Manufacturing (C.I.M.) 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.

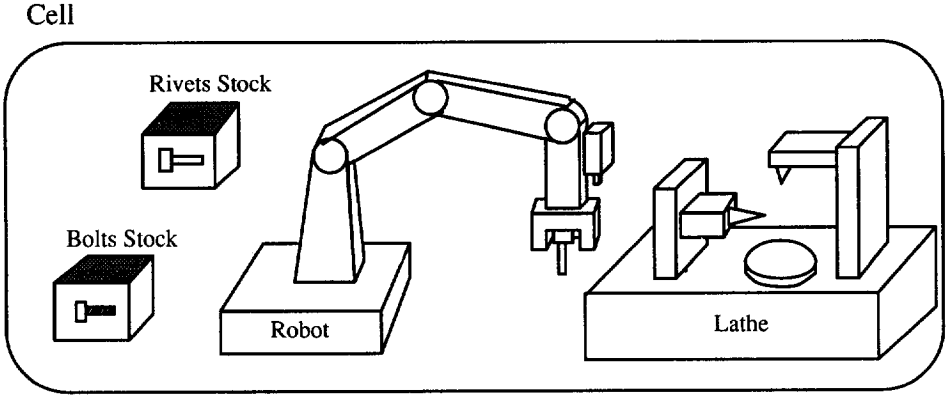

Fig. 1. The C.I.M. case study

2 Requirements Engineering of Composite Systems

In this section, we briefly details the activity of requirements engineering and its role within the software life-cycle; then we will define the concept of composite systems.

2.1 Requirements Engineering vs. Software Engineering

The majority of recent software development methods distinguish between two basic activities: *requirements engineering* and *software engineering* (see Fig.2). The former starts from informal wishes expressed by one or several customers and elaborates a so-called *requirements document* where the system to be developed is defined in a precise way. This document is the starting point of the software engineering (or design) activity, which develops a software system meeting the requirements in an efficient way.

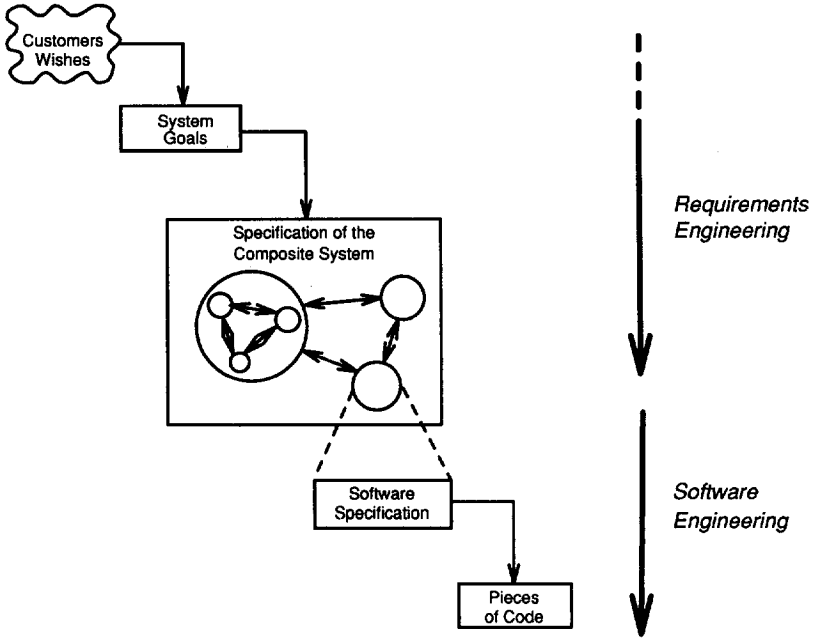

Fig. 2. Requirements Engineering vs. Software Engineering

2.2 The Four Tasks of Requirements Engineering

The requirements engineering (R.E.) activity is not as trivial as writing down the customers' wishes because these wishes are often unprecise, incomplete, ill-structured, and even inconsistent. Therefore requirement engineering implies a constant interaction between the customers and the analyst in order to progressively clarify the requirements.

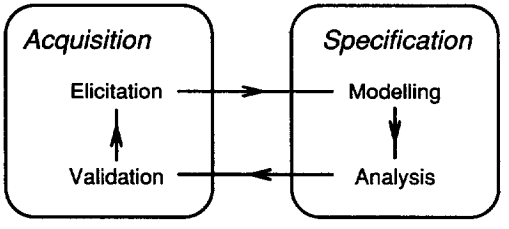

Fig. 3. Tasks of the R.E. activity

As suggested on Fig.3, R.E. can be subdivided in four interacting tasks [DHR91]:

- During the **elicitation** task, the analyst collects information about the customers' problem. This can be performed through several ways: interviews and discussions, observations, study of available documentation,...
- During the **modelling** task, the analyst processes the information collected during the previous task and elaborates a model, i.e. a formal representation of the problem. This model is a new fragment added to the requirements document elaborated so far.
- During the **analysis** task, the analyst detects problems (contradictions, inconsistencies,...) resulting from the incorporation of the new requirements fragment. These problems either relate directly to the the fragment itself or to the integration of this fragment with the rest of the requirements.
- During the **validation** task, the objective of the analyst is twofold. If problems were met during the analysis task, the analyst has to communicate them in a comprehensible way to the customers and try to solve them. Otherwise, the analyst controls the adequacy of their formalised descriptions by reformulating it to the customers in an appropriate way.

2.3 Properties of R.E. Languages

From the activities outlined above, we can derive some basic properties for an adequate language supporting requirements engineering. These properties are *expressiveness*, *structuring* and *formality*.

Expressiveness. To support effectively the elicitation and also the validation task, a language should be sufficiently expressive to support a natural mapping between all kind of things that should be described in a requirements document and the various concepts available in the language. Definitely, capturing requirements should not be a cumbersome coding task.

Structuring. Requirements documents are often quite large and, thereby, complex interactions exist between different pieces of descriptions. To facilitate the modelling task and to make the resulting document manageable, requirements should be organised into separable units which can be combined in a controllable way to yield to the complete specification.

Moreover, structuring mechanisms are also essential to support the reuse of specification components and the maintenance of requirements documents.

Formality. The formal aspect of a language depends on the availability of rigorous rules of interpretation which guarantee the absence of ambiguity. Besides, rules of deductive inference are needed to make possible the derivation of new statements from the given ones.

The deductive power of a language allows the development of tools supporting the analysis task (semantic checkers) and the validation task (prototype generators and inference tools).

2.4 Composite Systems

In the context of the development of real-time software and information systems, the requirements document, in its final stage of elaboration, should include not only specifications on the software piece to be implemented but also on the *environment* around this software as well as the nature of the interactions taking place between both (see, e.g., [Bjø92]). Let us consider two specific examples:

- in the context of an information system, requirements may include references to manual procedures to be installed (like, "this department is in charge of encoding the orders within the day of their arrival") or to real-world entities (like the following *accuracy* requirement [MCN92]: "the stocks quantities recorded in the information system should reflect the real-world stocks quantities with some delta").
- In the context of real-time software, requirements may include references to specific hardware/devices to be used (like, "the existence of an analog device which measures pulse and skin resistance in a patient monitoring system" [Fea89]) or to *performance* requirements (like, "in a furnace system, when the water temperature exceeds 100°C degrees, the alarm has to be triggered on within a 10 seconds delay").

In this paper, we consider requirements associated with the kind of systems described above that are sometimes referred as **composite systems** [Fea87] made of multiple *agents*. For example, one may think about a composite system including manual procedures, hardware, specific devices and software components interacting together. For example, in the context of the C.I.M. case study introduced in Sect.1, one may envisage (i), at the higher level, the system as being made of *Cell* agents and an *Environment* agent, and (ii), at a lower level, each cell as being made of multiple agents, viz the *Rivets container*, the *Bolts container*, the *Robot*, the *Lathe* and the *Controller*. The last three agents play an active role in the *Cell*. 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 (see Fig.4).

In the final version of the requirements document for a composite system, we have to be able to identify *responsibilities* attached with the different agents (computerised or not) and make them precise because they form a contractual part for the designers (not only the software designer) in charge of implementing them and guaranteeing their desired behaviour.

On top of the detailed behaviour attached to each agent identified, we also need to have a more abstract view of the global *goals* [Dub89, DFHF91], i.e. objectives associated with the system considered as a whole (black box approach). Goals are expressed in terms of system-wide properties, regardless of how responsibility is decomposed among agents.

The requirements engineering activity should thus encompass a phase of "organisational design", i.e. the work of organising a system into agents (subsystems), assigning responsibilities to them and describing their interconnections.

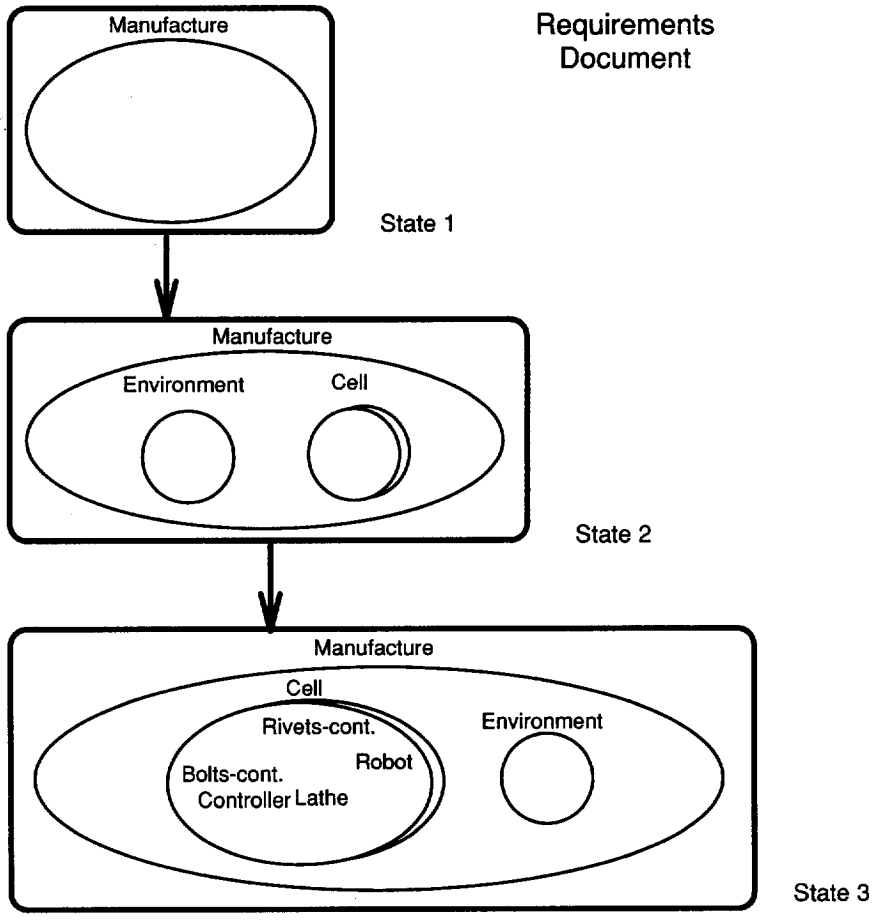

Fig. 4. Development of the Requirements Document

To conclude, we plead that an adequate method for supporting the gradual elaboration of the requirements document should include:

1. the specification of the *problem* expressed in terms of goals associated with the whole system to be developed;
2. the specification of a *solution* to the problem in terms of the description of a composite system where the set of requirements attached to individual agents meet the goals introduced.

In Sect.4, we will further detail the set of activities supporting the model sketched above.

3 ALBERT: the Language

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 are defined in [Du 92]. Their comprehensive presentation is however 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.5).

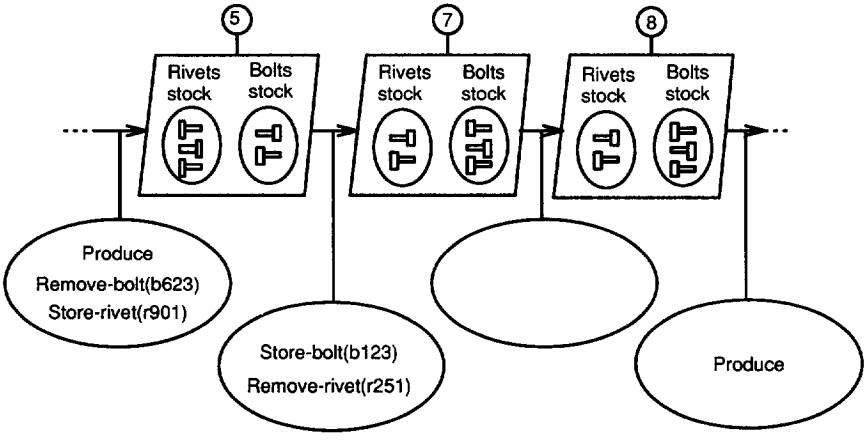

Fig. 5. A possible life of the *Cell* agent

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 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, HR92], 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:

- writing *declarations* introducing the vocabulary of the considered application,
- 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

The declaration part of an agent consists in the description of its states structure and the list the actions its history can be made of.

Agents are considered as *specialised* 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* [Bru91]).

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];
- 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 equality;
- more complex types built by the analyst using a set of predefined type constructors like set, list, Cartesian product,... (see, e.g. [BJ78]) and elementary types; on top of operations inherited from their structure, new operations can be defined on these new types;
- types corresponding to agent identifier.
  Agents includes a key mechanism that allows the identification of the different instances. A type is automatically associated to each class of agent. This correspond to the type of agents identifiers within that class. E.g., each *Cell* agent has an identifier of type *CELL* [2] [3].

Figure 6 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*;
- *Produce*, *Store-rivet*, *Remove-bolt*, *Remove-rivet* and *Store-bolt* are five actions which may happen in a *Cell* history. Actions can have arguments[4]; for example, each occurrence of the *Store-rivet* action has an instance of type *RIVET* as argument.

The diagram also includes graphical notations making possible (i) to distinguish between internal and external actions and (ii) 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

[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)

[2] 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).

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

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

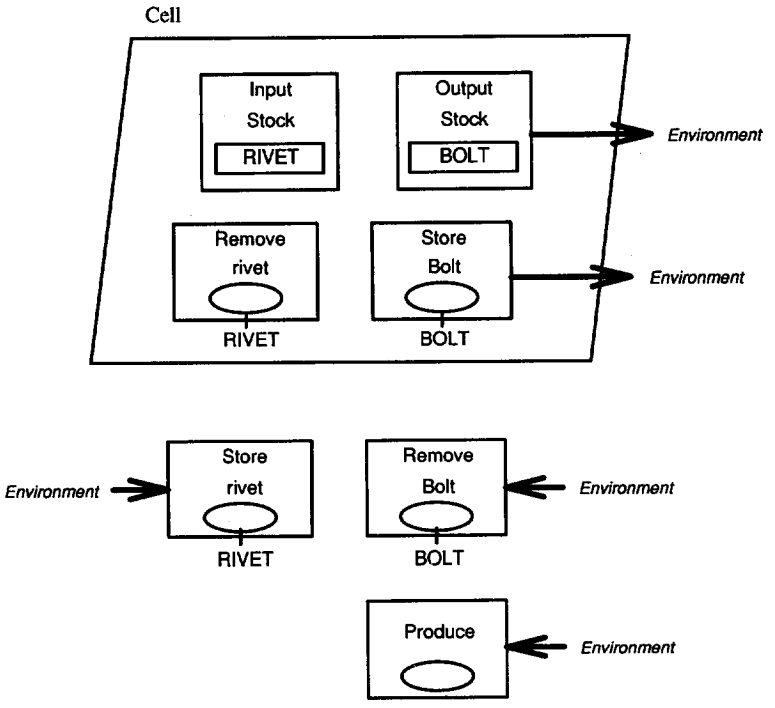

Fig. 6. Declaration associated with the *Cell* agent

read that *Cell* has the initiative for *Remove-rivet* and *Store-bolt* actions while it lets the *Environment* having the initiative of *Store-rivet* and *Remove-bolt* actions [5];

(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 are exported to the outside. From the graphical declaration, it can be read that the *Environment* may have knowledge of the *Output-stock* state and of the *Store-bolt* actions.

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

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

3.4 Constraints

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

Unlike in O-O design languages, e.g. OBLOG, the Albert semantics is not operational. A life must be extensively be 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.

Figure 7 introduces the specification associated with the behaviour of the *Cell* agent and refers to the graphical declaration introduced in Fig.6.

Cell

STATE BEHAVIOUR

$\neg$ *Empty(Input-stock)*
In(Input-stock,r) $\Longrightarrow \Diamond \ \neg$ *In(Input-stock,r)*

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)

COMMITMENTS

envt.Produce $\overset{\Diamond \leq 10min}{\longrightarrow}$ *Remove-rivet(r); Store-bolt(b)*

RESPONSIBILITY

PERCEPTION

F(envt.Remove-bolt / Empty(Output-stock))
X(envt.Produce / $\neg$ *Empty(Input-stock))*

PUBLICITY

X(Output-stock.{envt} / $\neg$ *Empty(Output-stock))*

Fig. 7. Constraints on the *Cell* agent

In order to provide some methodological guidance to the analyst, properties are grouped under six different headings: **State Behaviour, Effects of Actions, Responsibility, Perception, Publicity,** and **Commitments.**

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 constraint on Fig.7 expressing that the input stock may never be empty). 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 (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 constraint on Fig.7 expressing that a rivet may not stay indefinitely in the input stock). Writing these constraints require 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. 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 (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,* ... [KVdR89].

Effects of Actions. Beyond this heading, we describe the effects of actions[6] which may alter states in lives (see on Fig.7 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).

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

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 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.7, we express that the effect associated with the action *Store-rivet* issued by the external agent *Environment* 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.

Responsibility. 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 O, the F and the X connectives). The study of these connectives has been heavily influenced by some work performed in the area of *Deontic Logic* (see, e.g. [FM90], [Dub91]).

The pattern for an obligation "*O(<internal-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 "*F(<internal-action> / <situation>)*" expresses that the action is forbidden when the circumstances expressed in the situation are matched (e.g. "the cell cannot remove a rivet from the stock when the stock is empty", in other words, it is forbidden to the cell to produce the *Remove-rivet* action when the stock is empty).

The pattern "*X(<internal-action> / <situation>)*" is used to express exclusive obligation, it is a shorthand for the combination of "*O(<internal-action> / <situation>)* and *F(<internal-action> /¬ <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.

Perception. Beyond this heading we define how the agent is sensitive to the outside (i.e. to changes or state components which are made available to it by other agents belonging to the same society).

Perceptions are also specified using the O, F and X connectives.

The pattern "*O(<external-action> / <situation>)*" defines the situation where, if an action is issued by the external agent, the behaviour of the current

agent's state is influenced. For example, in our case study, one may think about the following constraint: "the cell is obliged to take into account a production request issued by the environment when the input stock is not empty" (in other words, *Produce* actions occurring in the environment has necessarily to affect the history of the cell when the input stock is not empty).

The pattern "*F(<external-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 (e.g. "the cell cannot remove a rivet from the stock when the stock is empty", in other words, it is forbidden to the cell to produce the *Remove-rivet* action when the stock is empty).

The "*X*" connective is defined for perceptions in the same way as for responsibilities.

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

Publicity. Constraints under this heading specify how information (i.e. occurrences of actions or state components) is made available by an agent to other agents belonging to the same society. This is also a dynamic property and is expressed using the O, F and X connectives introduced above.

For example, one may imagine that, in our case study, "the storing of a bolt is made visible to the environment if it happens between 9AM and 12AM" and the publicity statement on Fig.7 expresses that the *Environment* agent has no access to the "Output-stock" component of the *Cell* agent state when that stock is empty.

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].

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.7).

The "$\longrightarrow$" symbol can be quantified by a temporal operator to express performances constraints (e.g. the "$\overset{\Diamond \leq 10min}{\longrightarrow}$" symbol in the commitment on Fig.7 means that the occurrence of the *Store-bolt* action has to happen within a 10 minutes interval after the occurrence of the *Produce* 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)".

Some more complex expressions are provided to model iterations.

3.5 Complex Agents

The specification of a composite system is made of the specification of several agents. To be more precise, we propose to organise these in terms of a *hierarchy* where we distinguish between :

- *complex agents* corresponding to the nodes in the hierarchy and made of finer agents;
- *individual agents* corresponding to leaves in the hierarchy and which are not further decomposed.

In the sequel, we describe the *declarations* and *constraints* that can be associated with complex agents. In the next section, we will. make precise the role played by complex agents during the elaboration of a requirements document.

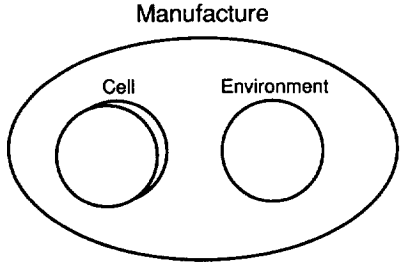

Fig. 8. Declaration associated with the *Manufacture* agent

Declarations of Complex Agents. Figure 8 proposes the graphical declaration associated with the CIM case study where *Manufacture* is a complex agent while *Cell* and *Environment* are individual agents. The *Environment* agent has been introduced for sake of simplicity but a more realistic specification would have been considered *Environment* as being itself composed of a *Production & Planning* unit, a *Quality Assurance* unit,...).

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 *Environment* agent and several *Cell* agents (which form a "class").

Constraints on Complex Agents. A complex agent has no specific behaviour except the aggregation of the local behaviours of individual agents which it is composed of.

In the specification associated with a complex agent, we will only express global properties derived from properties attached to individual agents. These properties are called **goals** and correspond to the declarative part of the statements presented in the previous sub-section, viz constraints expressed under the **State Behaviour** and the **Commitments** headings.

On Fig.8, it would have been possible to attach *goals* to the *Manufacture*. The concept of *goals* will be illustrated in the next section.

4 The Elaboration of the Requirements Document

4.1 The Elaboration Process

The requirements document, in its final stage of elaboration, is usually a complex document due to the number of individual agents belonging to the composite system and the complexity of the interactions taking place among these agents. Therefore, one cannot imagine that the R.E. activity performed by the specifier only consists in the transcription of the customers' wishes in terms of the requirements document. It is essential to provide some methodological guidance in the *process* of a progressive elaboration of the requirements document.

The elaboration of a specification can be seen as a sequence of development steps, each step being defined by the application of a transformation on the current state of the requirements document and resulting in a new state of this document [DvL87]. The application of a transformation at some stage of the development depends on some strategies followed by the specifier.

In some recent work [Dub91, DDR92], we have proposed the following strategy:

1. Express specifications on the problem, i.e. on the system considered as a monolithic one (i.e. black-box approach).
2. Identify new sub-systems and attach to each of them their responsibilities so that the refinement preserves the original prescribed behaviour.
3. Apply recursively the step 2 up to the identification of 'terminal' sub-systems, i.e. components for which designers (see Sect.2) agree on the implementation of their attached properties.

Applying this strategy to the *Manufacture* application results in the possible sequence of development steps already suggested on Fig.4 where:

- in state 1, the *Manufacture* is considered as an individual agent in our language;

- in state 2, the *Environment* and *Cell*'s are the individual agents, the *Manufacture* is now considered as a complex agent;
- in state 3, the final version identifies new individual agents composing the *Cell* agents (now considered as complex agents).

In the next sub-section, we further detail the process followed by the specifier when he/she goes from the state 2 to the state 3 of the requirements document.

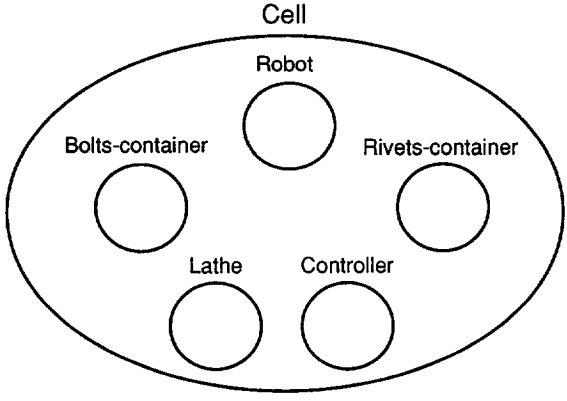

Fig. 9. Declaration associated with the *Cell* agent (II)

4.2 Identifying Agents Responsibilities

In the previous section, we introduced Fig.8 which represents the state of the requirements document at the state 2 of the process. This specification is not a terminal one because the cell is more a "virtual" agent rather than a "concrete" one (i.e. which can be implemented by a designer). To this end, the specifier has to follow a *refinement* process (analogously to the one followed in software design, see, e.g., [BJ78]) which will result in the final version of document presented on Fig.9.

At this level, five new terminal agents are identified:

1. the *lathe* corresponds to the machine in charge of the transformation of a rivet into the bolt. This agent is a device;
2. the *Input-container* is a passive agent in charge of keeping rivets stored on it;
3. the *Output-container* is a passive agent with a role similar to the previous one, viz a bolts keeper;
4. the *Robot* is an active agent in charge of transporting rivets and bolts between containers and the lathe machine;

5. the *Controller* is the software piece which plays the important role of managing the other agents so that the production goal attached to the cell is reached.

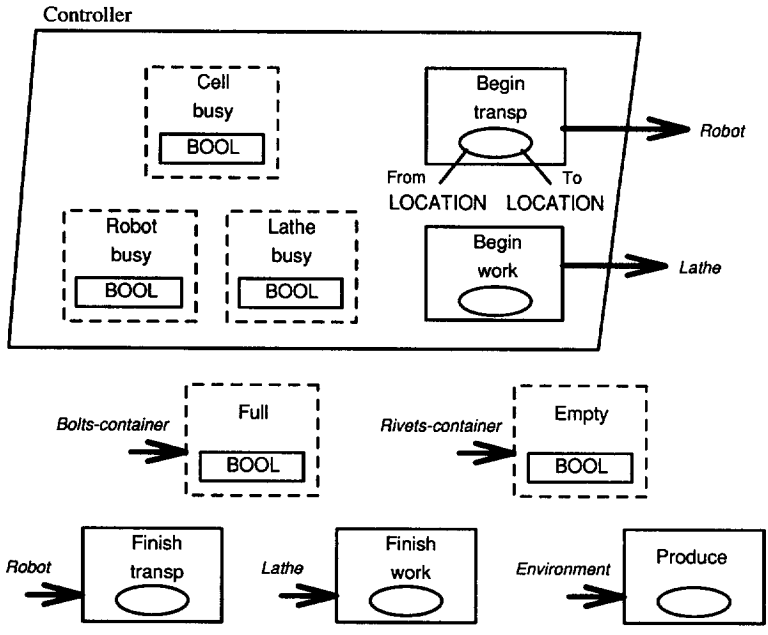

Fig. 10. Declaration associated with the *Controller* agent

For lack of place, the formal specification characterising the behaviour of each agent is not further detailed. On Fig.10 and Fig.11, we are just making precise the role of the *Controller*. From this specification, it results that the controller is in charge of managing the *Produce* orders by issuing the appropriate orders to the different other agents and by controlling their behaviour. For example, the *Controller* is issuing transportation orders to the *Robot* and is aware of the emptiness of the *Rivets-container* (which is equipped with a sensor).

4.3 Reasoning on Agents Responsibilities

In the specification presented on Fig.9, *Cell* is identified as a complex agent and thereby *goals* are associated with it. These goals correspond to a subset of the specification presented on Fig.7 and restricted to declarative constraints (**State Behaviour** and **Commitments**).

The refinement process followed by the specifier is a *correct* one if the goals attached with the *Cell* complex agent are met by the joint behaviour of the

Controller

STATE BEHAVIOUR

$\neg$ *(Lathe-busy* $\wedge$ *Robot-busy)*
(Lathe-busy $\vee$ *Robot-busy)* $\Longrightarrow$ *Cell-busy*

EFFECTS OF ACTIONS

Begin-transp(o,d): Robot-busy = true
Begin-work: Lathe-busy = true
envt.Produce: Cell-busy = true
robot.Finish-transp(o,d): Robot-busy = false
robot.Finish-transp(lathe,bolts-shelf): Cell-busy = false
lathe.Finish-work: Lathe-busy = false

COMMITMENTS

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

RESPONSIBILITY

F(Begin-transp(o,d) / Robot-busy $\vee \neg$ *Cell-busy)*
F(Begin-work / Lathe-busy $\vee \neg$ *Cell-busy)*

PERCEPTION

X(envt.Produce / $\neg$ *Cell-busy* $\wedge \neg$ *rivets-shelf.Empty* $\wedge \neg$ *bolts-shelf.Full)*
X(robot.Finish-transp / Cell-busy $\wedge$ *Robot-busy)*
X(lathe.Finish-work / Cell-busy $\wedge$ *Lathe-busy)*

PUBLICITY

Fig. 11. Constraints on the *Controller* agent

five individual agents [FH91]. For example, the goal of "*producing a bolt within 10 minutes after a request*" has to be achieved through the combination of the individual actions made by the *Controller*, *Robot* and *Lathe*.

Due to the formality of our language, a formal proof can be exhibited. For sake of brevity, it is not detailed in this paper but the interested reader may refer to [Dub89] for further discussion.

5 Conclusion

Analogously to some recent contributions in Distributed Artificial Intelligence [Hew91] or in the Design of Distributed Software [SSE89, FM90], we claim that a formal language for capturing requirements should encompass the specification of a composite system, i.e. a system made of multiple agents. Different authors propose different concepts for modelling agents and reasoning on their properties (see, e.g. [Fea87, Bjø92, DFvL91]). In this paper, we introduce ALBERT, a formal language for capturing some of them. This language includes a certain number of features also proved useful in languages like TROLL [JSS91] or DAL [RFM91]. It also incorporates recent results gathered in O-O conceptual modelling by considering agents as a specialization of the object concept (agents being characterized by some specific dynamic properties).

For lack of place, we have not illustrated the structuring mechanisms (based on parameterization and inheritance) which can be used to achieve (i) a better structuring of the requirements document (in particular, by defining a specific vocabulary for some application domain) and (ii) the reuse of existing specification fragments. More details on that topic can be found in [DDR92].

Within the framework of an Esprit II project, a semantics has been given to our language in terms of GLIDER [DDRW91] (A General Language for an Incremental Definition and Elaboration of Requirements). The GLIDER language is based on a so-called *loose (algebraic) semantics* [OSC89] which supports the expression of constraints using, among other things, typed first-order formulas and *streams* (i.e. infinite sequences). In this project, a set of supporting tools are also under development including, in particular, a graphical editor for our language. Finally, our research plans are in three directions:

- the validation of the language concepts and the enhancement of the proposed methodology through the study of conclusions from large industrial experiments currently done in a Computer-Integrated-Manufacturing (C.I.M.) environment and in the field of advanced communications applications;
- at the product level, the identification of a set of orthogonal patterns supporting the expression of different forms of *commitments*;
- at the process level, the study and the formalisation of a set of typical *transformations* and of their *strategies* of use. Examples of them include the development of specifications for non reliable agents and/or non omniscient agents.

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.

References

[AG91] A.J. Alencar and J.A. Goguen. Ooze: An object oriented Z environment. In P. America, editor, *Proc. of the 5th european conference on object-oriented programming – ECOOP'91*, pages 180–199. LNCS 512, Springer-Verlag, 1991.

[BJ78] D. Bjørner and C.B. Jones. *The Vienna Development Method. The meta-language*, 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.

[BMR92] A. Borgida, J. Mylopoulos, and R. Reiter. ...and nothing else changes: The frame problem in procedure specifications. Technical Report DCS-TR-281, Dept. of Computer Science, Rutgers University, 1992.

[BP83] F. Bodart and Y. Pigneur. *Conception assistée des applications informatiques. Première partie: Etude d'opportunité et analyse conceptuelle.* Masson, Paris, 1983.

[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.

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

[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.

[DDRW91] Eric Dubois, Philippe Du Bois, André Rifaut, and Pierre Wodon. Glider user manual. Intermediate Deliverable SpecFunc-028-R, ESPRIT Project Icarus 2537, June 1991.

[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.

[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.

[DKRS91] Roger Duke, Paul King, Gordon Rose, and Graeme Smith. The object-z specification language – version 1. Technical report 91-1, SVRC, Dept. of Computer Science, The University of Queensland, Queensland (Australia), May 1991.

[Du 92] Philippe Du Bois. Using glider for the formal definition of a requirements language for specifying composite systems. Technical report, Computer Science Department, University of Namur, Namur (Belgium), May 1992.

[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.

[Dub91] Eric Dubois. Use of deontic logic in the requirements engineering of composite systems. In J.J. Meyer and R.J. Wieringa, editors, *Proc. of the first international workshop on deontic logic in computer science*, Amsterdam (The Netherlands), December 11-13, 1991.

[DvL87] Eric Dubois and Axel van Lamsweerde. Making specification processes explicit. In *Proceedings of the 4th international workshop on software specification and design*, pages 161–168, Monterey CA, April 3-4, 1987. 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.

[FF89] Anthony Finkelstein and Hugo Fuks. Multi-party specification. In *Proceedings of the 5th international workshop on software specification and design*, pages 185–195, Pittsburgh PA, May 19-20, 1989. IEEE, CS Press.

[FH91] Stephen Fickas and Rob Helm. Acting responsibly : Reasoning about agents in a multi-agent system. Technical Report CIS-TR-91-02, Dept. of Computer and Information Science, University of Oregon, Eugene OR, 1991.

[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 and its logic. In M.L. Bodie and J. Mylopoulos, editors, *On knowledge base managment systems*, Topics in information systems, pages 471–502. Springer-Verlag, 1986.

[GHW85] John V. Guttag, James J. Horning, and Jeannette M. Wing. Larch in five easy pieces. Technical Report 5, Digital systems research center, Palo Alto CA, July 1985.

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

[Hew91] C. Hewitt. DAI betwist and between: open systems science and/or intelligent agents. In J. Mylopoulos and R. Balzer, editors, *Proc. of the international workshop on the development of intelligent information systems*, Niagara-on-the-Lake (Canada), April 21-23, 1991.

[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.

[JF90] W. Lewis Johnson and Martin Feather. Building an evolution transformation library. In *Proceedings of the 12th international conference on software engineering*, Nice (France), March 1990. IEEE.

[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.

[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.

[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.

[MCN92] J. Mylopoulos, L. Chung, and B. Nixon. Representing and using nonfunctional requirements: a process-oriented approach. *IEEE Transactions on software engineering*, SE-18, June 1992.

[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.

[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.

[Ros77] Douglas T. Ross. Structured analysis (sa): a language for communicating ideas. *IEEE Transactions on software engineering*, SE-3(1):16–34, January 1977.

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

[Sho90] Y. Shoham. Agent-oriented programming. Technical report STAN-CS-90-1335, Robotics Laboratory, Computer Science Dept, Stanford University, Stanford CA, 1990.

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

[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.

[TH77] D. Teichroew and E. Hershey. A computer-aided technique for structured documentation and analysis of information processing systems. *IEEE Transactions on software engineering*, SE-3:41–48, 1977.

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

[TRC83] H. Tardieu, A. Rochfeld, and R. Colletti. *La méthode MERISE: principes et outils*. Les Editions d'Organisation, Paris (France), 1983.