MASTER IN COMPUTER SCIENCE

A proposal for A requirements engineering method dealing with organisational, non-functional and functional requirements

Bissener, Michel

Award date:
1997

FACULTÉS UNIVERSITAIRES NOTRE-DAME DE LA PAIX, NAMUR
INSTITUT D'INFORMATIQUE
RUE GRANDGAGNAGE, 21, B-5000 NAMUR (BELGIUM)

A Proposal For A Requirements Engineering Method Dealing With Organisational, Non-Functional And Functional Requirements

Michel Bissener

Supervisor : Professor Eric Dubois

Thesis submitted in conformity
with the requirements for the degree of
'Licencié et Maître en Informatique'

August 1997

Abstract

Methods are often required for the development of software. Besides these software engineering methods, the requirements engineering activities should also be supported by a method.

Furthermore, in addition to the functional requirements which have for a long time been the focus of scientific research, organisational and non-functional requirements have become increasingly important. Requirements documents should not only limit their scope to the specification of the functionality that a system should provide. Non-functional requirements of a system just as the organisational environment (the place and time) a system is introduced should be modelled.

The thesis proposes a methodology that deals with organisational, non-functional and functional requirements. On the one hand, the functional requirements are specified in the agent-oriented specification language ALBERT. On the other hand, we use the *i** framework to deal with organisational issues and non-functional requirements. These non-functional requirements, after they have been explored more in detail, are partially or completely composed of definable requirements. In addition, the methodology is illustrated and validated by a case study: the Mail Order problem.

Des méthodes sont souvent exigées pour le développement de logiciels. Le développement des besoins des utilisateurs devrait aussi être supportée par une méthode.

De plus, à côté des exigences fonctionnelles, qui furent longtemps la cible de la recherche scientifique, les exigences organisationnelles et non fonctionnelles ont gagné en importance. Ainsi, la documentation des besoins des utilisateurs ne devrait pas seulement inclure les exigences fonctionnelles mais aussi les besoins non fonctionnelles et organisationnelles.

L'objectif de ce travail de fin d'études est de proposer une méthode qui traite les exigences fonctionnelles, non fonctionnelles et organisationnelles. De l'un côté, les exigences fonctionnelles seront spécifiées dans le langage de spécification ALBERT. De l'autre côté, les besoins non fonctionnels et les questions organisationnelles seront modélisés en *i**. La méthode sera ensuite illustrée et validée par une étude de cas.

Acknowledgements

I would like to thank all the people that helped me in the writing of this thesis:

My supervisor, Professor Eric Dubois, for his advice and the many re-readings and corrections by which he helped me to realise the thesis.

Professor Eric Yu and Professor John Mylopoulos, from the University of Toronto (Ontario Canada) for their advice and support during the stay in Toronto.

Michaël Petit for his important help when writing the specifications, for the constructive comments and for being so busy in re-reading the thesis.

Finally, I would like to thank all the people that showed interest in the thesis.

Contents

CHAPTER 1
INTRODUCTION

The development of an information system is generally divided into two phases as depicted in figure 1: the Requirements Engineering phase and the Design Engineering phase.

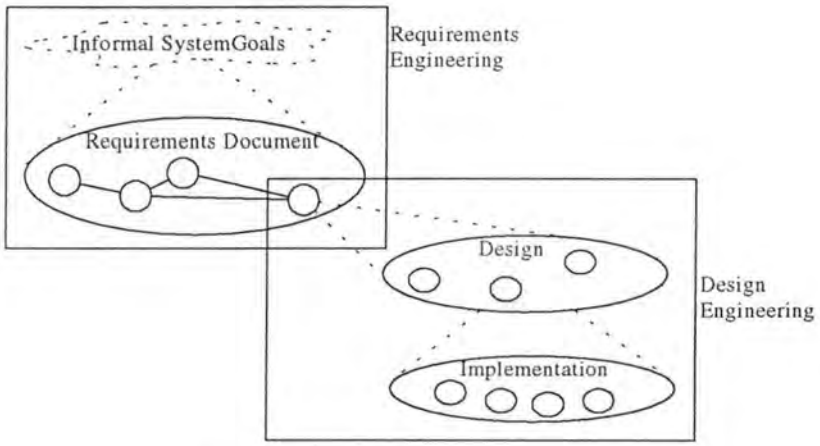


Figure 1: The IS development cycle

The **Requirements Engineering** (RE) phase starts with the informal statements representing goals, services, viewpoints of the clients. These statements should be put into an organisational context, identifying their owners and their motivations. The requirements document should define the system to be developed in a precise way. This document includes specifications of expected human, hardware and software behaviours.

At the **Design Engineering** (DE) level, we start with the software specification (defined in the requirements engineering phase). This latter is used for the development of a logical architecture of the software which is later implemented by writing code.

RE: Definitions and characteristics

Requirements Engineering has for a long time been recognised as a, if not the, crucial part of the system development process. Many definitions have been given for this first phase in the life cycle of a software system (it should be noted that this is *not* an exhaustive list):

- *The discipline for developing a complete, consistent, unambiguous specification -which can serve as a basis for common agreement among all parties concerned- describing what the software product will do (but not how it will do it; this is to be done in the design specification).* [Boehm,79]
- *Specifying the requirements document of a software, it is to define in a complete and non-ambiguous way the external characteristics of the software offered to its users and the way according to which the software is integrated in the system.* Meyer [MDL,97]
- *During the requirements stage, it is necessary to analyse, and thus to understand the problem to be solved. Problem analysis is the activity that encompasses understanding the needs of the users as well as all the constraints upon the solution.* Davis [MDL,97]
- *(1)The Process of studying user needs to arrive at a definition of system, hardware, or software requirements. (2)The process of studying and refining system, hardware or software requirements.*[IEEE,91]
- *The part of development in which people attempt to discover what is desired.*[Gause et al,89]

From the definitions above, we can deduce some characteristics. These define our view of requirements engineering:

- **Requirements Engineering is a process.**

This is implicitly indicated in all the definitions and explicitly remarked by the IEEE. RE is much more than simply stating requirements in a formal or informal notation. Just as for the life-cycle for the design of a system, there exist different phases for Requirements Engineering: elicitation, modelling, analysis and validation[Dubois et la, 95].

The division into steps indicates already that requirements engineering is not merely the transcription of the user's statements. Furthermore, not only one solution has to be considered. Ideally, an important set of possible solutions should be imaginable at the system design stage.

In order that this process is not left to the analyst's discretion, a rational process should be provided for elaborating the requirements. The aim of the thesis is to propose a method for elaborating a requirements document.

- **Requirements Engineering produces a document.**

This is what most of the definitions have in common. Almost any definition says that the RE phase should finish with a complete, consistent and unambiguous document, specification or model defining what is desired or needed. One should note that we do not say that RE takes requirements, that were already stated informally by the client, as input. We rather believe that requirements should be collected in co-operation with the client during the RE phase.

- **Requirements Engineering relates worlds.**

As already established above, RE brings together the client world and the analyst/developer world. We made the assumption that the different customer and users have the same goals and needs, which is often, or almost always, untrue for the real world. This, of course, complicates the situation as customers and users might have conflicting goals. Usually, each world is new to the domain of the other one's; which considerably complicates the transition.

Ideally, customers, users and analysts should, *together*, find out what the users' and customers' wishes are and refine them into a detailed description.

- **Requirements Engineering is a means.**

Consequently, requirements engineering can be considered as a means for supporting the interactive development. Analysts are forced to understand the problem domain and customers are forced to describe rigorously their needs. This elaboration can also be of profit for the analysis of organisational issues. In addition, one can say that analysts should be supported by CASE tools in achieving this knowledge.

Methodology

The aim of the thesis is to describe the process of elaborating the requirements document. When we focus more on the Requirements Engineering phase we can depict a more detailed approach for specifying the information system. Figure 2 represents this approach.

The Business Enterprise Model (problem domain) identifies the organisational settings of the problem domain plus the additional goals/objectives to be delivered by the future system.

From these global goals we specify a system which is first defined by the necessary knowledge about the environment and the control upon the environment.

The system is then split into different concrete agents assuming each a different responsibility for accomplishing the overall goals. Their internal structure and their interaction with the system's environment is designed. These agents vary from software units over hardware units to humans.

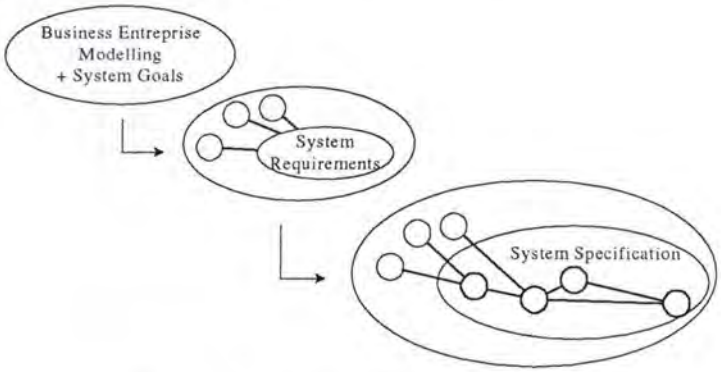


Figure 2: The Development Approach

Functional, Non-Functional and Organisational Requirements

The thesis also deals with functional and non-functional requirements. Even if there has been a discussion about their differentiation (about the existence and about the meaning), they are generally divided into these two classes. The difference, however, is often influenced by the viewpoints of people and the membership to one class often depends upon the language (and its expressiveness) one uses to denote the requirement. For example, time constraints can be regarded as functional requirements in certain languages whereas other classifications include them in the set of non-functional requirements. An interesting and more detailed classification of non-functional requirements can be found in [Pohl,96].

Functional Requirements

Most of the research and literature in the past focused on functional requirements, which define the functionality a system has to provide. This approach is deeply influenced by a mathematical background: a function is a correspondence of a set S towards of a set T, binding each element of S to one and only one element of T. "Given a set of numbers, the user wants the system analyst/developer to give him a function that provides him with this or that result." They are what could traditionally be defined as the 'what' of the system. A functional requirement of a program could be: "Calculate the salary of an employee".

Non-Functional Requirements

Gaining increasingly importance, the non-functional requirements aim basically at constraining the choice of solution regarding a problem. Since then, non-functional requirements tell us how a result should be produced. Typically, non-functional requirements are requirements such as user interface constraints, operating constraints, maintainability, security, portability, standards,... Our example chosen previously can clarify the already discussed problem about the difference of the two types of requirements. Let "Calculate the salary of an employee in α seconds" be the requirement. Is this requirement now a functional or non-functional requirement?

We relate the difference between functional and non functional more to the quantifiable respective the qualitative aspects of the requirement. Quantifiable requirements can be precisely defined. This definition does not necessarily suppose a formal language. Any type of language (even natural language) may be well suited for the expression of the condition. Quality requirements describe properties which can, a priori, not be defined. In our example, if 'α' is a defined value, the requirement is functional. If the value is yet unknown (e.g. 'α' means " as quickly as possible"), it is a non functional requirement.

According to this differentiation, non functional requirements, if they are explored more in detail, often turn out to be partially or completely composed of functional (quantifiable) ones.

The framework presented in this thesis deals with functional requirements, basically addressed by the Albert language, and non-functional requirements, represented in the *i** model.

Organisational Requirements

The *i** framework also supports the modelling of organisational issues. Information systems are not intended to work individually. They are put into an environment where they have to co-operate with other organisational actors. Information systems, however, often fail because they neglect these organisational settings. Consequently, the organisational requirements should be inspected as soon as possible.

Organisation of the thesis

Chapter 2 presents the first specification framework, *i**, which is used to understand organisational processes. The *i** framework consists of two models: the Strategic Dependency Model and the Strategic Rationale Model. We define the different concepts used to built these models and explain them on an example..

Chapter 3 introduces the second framework. We use the ALBERT language to specify functional requirements. The concepts of agents, state components, actions and information/perception will be described and explained on an example.

Chapter 4 introduces the methodology itself. The three main steps are defined and related problems are discussed.

Chapter 5 explains the different concepts introduced by the methodology by the means of a case study (a mail order example).

Chapter 6 draws conclusions of the thesis and identifies further work.

CHAPTER 2
THE *i** FRAMEWORK

In this chapter, we describe the first modelling language we intend to use for the elaboration of our requirements specification: *i** (read i-star), a framework for modelling strategic relationship. *i** has been first introduced in Eric Yu's Ph.D. thesis [Yu,95]. Yu identifies four application domains for *i**: Business Process Reengineering, Organisational Impacts Analysis, Software Process Modelling and last but not least Requirements Engineering. We will concentrate more on this latter.

1. Why *i**

Human behaviour is generally described as a process, i.e. a series of actions that brings about a result. These human processes have been mainly influenced by the industrial revolution and now the computer revolution. In order to rationalise these processes it is necessary to model the existing process.

A lot of the existing process modelling frameworks deal with the flow of actions and/or entities that are necessary for the execution of the process. We can distinguish between two basic components of such process models :

- the Object Flow

The Object Flow Model partially describes the behaviour of a process by showing how these objects flow from one entity to another. Objects might be of an informational nature as well as of a physical, concrete nature. A popular specialisation of such a model is the Data Flow Model, where we represent a process by the information that flows between entities. Most of the methods analysing processes use such a Data Flow Diagram like in SADT, ...[DeMarco,79]

- the Action Structuring

The Action Structuring Model could be defined as a set of concepts and rules that should structure the process into a temporal sequence of actions. Actions can be atomic or not. Non-atomic are further decomposed in a hierarchical way.

Most of the existing process models are combinations/specialisations of these two generic models. A process is represented by a set of atomic and non-atomic actions and for each of these actions some input and output objects are associated. The output of an action may become the input of the following action (and so on) or a final output of the process. This description outlines the strong influence of the industrial and computer science world in process modelling: indeed Scheer [Scheer93] identifies two paradigms in modelling organisational processes. The classical, industrial approaches, influenced by the manufacturing industry, highlighted the flow of material that circulated in the plant. People were purely regarded as material processors and the challenge of process modelling was seen in shortening the product development cycle or the production costs. Then, with the massive introduction of the computer into our society processes, organisational modelling concentrated on representing procedures and data. As Scheer rightly remarks people appeared "as data records, input-output mechanisms, or substitutes of programs".

The main problem with these models is that they do not represent the rationales of different actors and their actions: why actions are performed or why objects are transformed.

This is the main motivation for the *i** model that puts more emphasis on the intentional aspects of the organisational model.

The *i** framework consists of two models: the strategic dependency model "describing a particular configuration of dependency relationships among organisational actors" and the strategic rationale model "describing the rationales that actors have about adopting one configuration or another." In the following, we briefly introduce these two models. For a more detailed description, refer to Eric Yu's Thesis [Yu,95].

2. The Strategic Dependency Model

The Strategic Dependency Model consists of a set of actors where each actor is related to some other actors by a dependency link expressing a dependency relationship. An *actor* is an active entity that, in order to meet some goals, carries out actions by exercising its know-how.

The dependency relationship is called *dependum*, the depending actor is called *depender* and the actor who is depended upon is the *dependee*. The dependency relationships can be analysed in terms of ability and vulnerability. When an actor depends on another actor he becomes at the same time able and vulnerable on this actor to achieve a certain goal: he can achieve that goal by delegating it to another actor, but if this actor fails in attempting this goal, the depender is negatively affected. Actually, organisations are often based on delegating tasks and so organisational actors depend on other organisational actors.

In the Strategic Dependency Model, we find four types of dependency links:

- goal dependencies
- task dependencies
- resource dependencies
- softgoals dependencies.

2.1 The Goal Dependency Relationship

We use this relationship when 'the depender depends upon the dependee to bring about a certain state in the world.' It is not explicitly defined what the dependee has to do to achieve this goal, in contrast to the task dependency. The actor has the freedom to choose a way to fulfil the condition. By expressing such a dependency, we say that the depender assumes that the requirement or state of the world will hold but he becomes vulnerable since the dependee may fail to bring about that condition.

The notions of goal, condition or state in the world we used above, are clear-cut, black-white notions of goal achievement. It is either achieved or not; there can only be a positive of negative answer to the request if the condition is satisfied. To clarify the concepts, we will illustrate it by an example. Let us take for instance a mail order company: an Order Processing Clerk who is responsible for handling incoming orders depends on the shipping clerk for that the desired item will be shipped to the customer. The depender (the Order Processing clerk) is only interested in the fact that the item is shipped and lets the shipping clerk the freedom to choose the appropriate way for achieving the condition. Figure 3 represents this relationship.

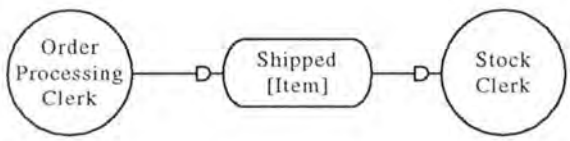


Figure 3: A Goal Dependency

2.2 The Task Dependency Relationship

This relationship expresses the fact that the depender depends upon the dependee in order to perform an activity that is specified by the depender. In other words, the depender has given instructions how to perform the task and the depender may suffer from the fact the task is not performed in the specified sense. Furthermore, by this dependency the depender only represents the 'how' (i.e. the specification) and not the 'why' (i.e. the rationale) to the dependee. Although the concept of a task might seem quite similar to an action in traditional workflow models, there is an important semantic difference between these two concepts. In opposition to a workflow model, where a task is an activity that manipulates some data, the Strategic Dependency Model only considers activities that have a strategic importance or are helpful in clarifying the

intentional behaviour of the actor. It is also important to note that the task specification should be considered as a constraint on the actual task rather than a complete and consistent description of the task to perform.

Again, let us show an example: if, for whatever reason, the Order Processing clerk would insist that the shipping clerk should send the item using a particular delivery company the depender may suffer if the task is not executed as specified. We would then represent this dependency by a task dependency relationship (see figure 4).

Figure 4: A Task Dependency

2.3 The Resource Dependency Relationship

A resource dependency expresses the fact that an actor depends upon another actor to provide him with a certain resource. This resource can be an informational or physical entity of the real world. As for the task dependency relationship, the resource relationship only expresses strategic aspects of the 'data' and thus cannot be compared to the standard data flow model. A resource dependency is only modelled if the depender becomes vulnerable, i.e. he would suffer, if the resource is unavailable. A striking example may clarify the previous notion: a diabetic depends on the druggist for the resource 'insulin' and would be in sincere danger whenever the druggist would run out of stock whereas a 'normal' person would not suffer if the druggist runs out of toothpaste. Applied to the mail order example, the Order Processing clerk who has to receive information about the customer's account balance would suffer because he could not continue to process the mail orders (see figure 5).

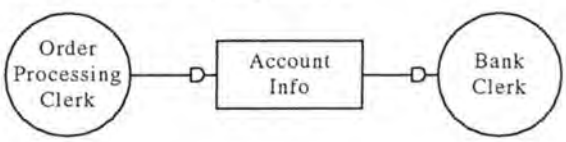


Figure 5: A Resource Dependency

2.4 The Softgoal Dependency Relationship

The first of the dependencies we described were the goal dependencies. Goals were described as clear-cut; they are achieved or not. But sometimes, the depender has only a vague notion of the goal that has to be achieved or "the condition to be attained are elaborated as the task is performed". Often, we cannot define the exact meaning, condition of a goal that has to be achieved. In addition, the goal and consequently its achievements are left to interpretation. For this situation, we use softgoal dependencies.

If we take the resource 'Account Info' from the figure 5, the Order Processing Clerk would also be negatively affected if the account information would not be accurate and

we don't yet know what an accurate information would look like. Consequently, we can add the softgoal 'Accurate [Account Info]' (see figure 6).

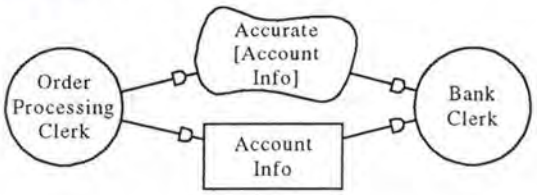


Figure 6: A Softgoal Dependency

2.5 Dependency Strength

The model also provides a weighting of dependencies. The three degrees are Open, Committed and Critical and are represented by an "O" for open, a "X" for critical. These "marks" can be placed on both sides of the dependency. Committed dependencies are unmarked.

Critical Dependencies denote dependencies where the depender is seriously affected "in that all known course of action would fail". In this case, the depender is also "concerned about the viability of the dependee's dependencies." (see figure 7)

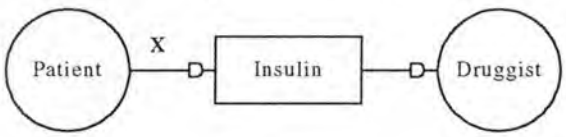


Figure 7: Critical Ressource Dependency

Committed Dependencies affect the depender significantly "in that some planned course of action would fail". On the dependee side, the committed dependency expresses that the depender "will try its best to deliver the dependum".(see figure 8)

Figure 8: Committed Goal Dependency

Open Dependencies represent the fact that the depender is affected to some extend but not seriously. For the dependee the open dependency is an assertion that he can deliver the required goal, softgoal, resource or perform the specified task. (see figure 9)

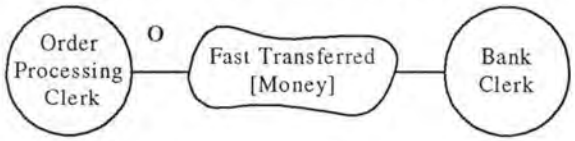


Figure 9: An Open Softgoal Dependency

2.6 Actor Specialisation

The term Actor "refers generically to any unit to which intentional dependencies can be ascribed". To describe more precisely organisational issues, the *i** framework differentiates between roles, agents and positions. Since we will frequently use this type of notation, we will give the definitions from Eric Yu's thesis.

- *A **role** is an abstract characterisation of the behaviour of a social actor within some specialised context or domain of endeavour. Its characteristics are easily transferable to other social actors. Dependencies are associated with a role when these dependencies apply regardless of who plays the role.*
- *An **agent** is an actor with concrete, physical manifestations, such as a human individual. We use the term agent instead of person for generality, so that it can be used to refer to human as well as artificial (hardware/software) agents. An agent has dependencies that apply regardless of what roles he/she/it happens to be playing. These characteristics are typically not easily transferable to other individuals, e.g. its skills and experiences, and its physical limitations.*
- *A **position** is intermediate in abstraction between a role and an agent. It is a set of roles typically played by one agent (e.g., assigned jointly to that one agent).*

In figure 10, the 'Bank Clerk' is represented as an agent who plays two roles: 'Account Information Provider' and 'Money Transfer Processor'. Another possibility is to represent two physically different agents who play each one role: the 'Information Clerk' who has to provide the customer with information about their accounts and the 'Operation Clerk' who plays the role of transferring money.

One should, however, note that agents only do not necessarily refer to individuals.

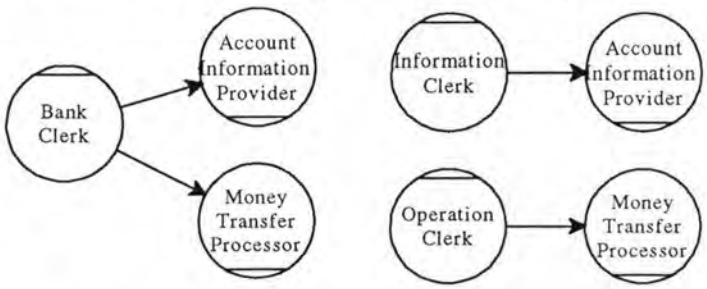


Figure 10: Two possible actor-'specialisations'

The following graph summarises the different concepts and the graphical notations:

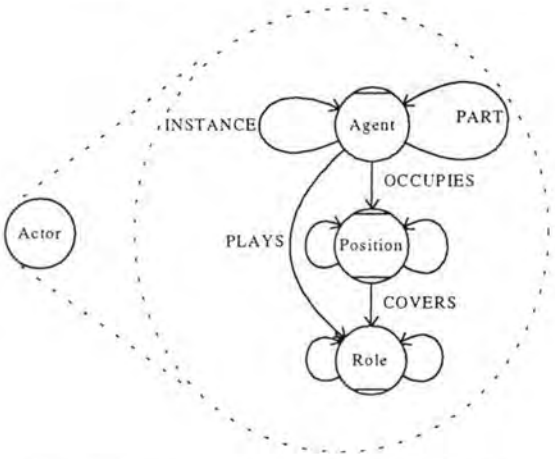


Figure 11: Actor, Agent, Role and Position

3. The Strategic Rationale Model

While the Strategic Dependency Model 'only' describes the external relationships between the actors of an organisation, the Strategic Rationale Model goes more into detail with the description of a process. The Strategic Rationale Model extends the Strategic Dependency Model by describing 'the intentional relationships that are internal to actors'.
This is mainly done by means of two classes of relationships that are internal to one actor:

- task decomposition, where a particular task is decomposed into a finer set of concepts and
- means-ends links, where the relationship between an end - which can be a goal, a resource, a task or a softgoal- and a means for achieving the end.

3.1 The Task Decomposition links

In an actor's context one can describe the tasks that he has to perform in order to meet a dependency relationship. These tasks can be structured by the means of task decomposition links. By this mechanism, a task can be divided into sub-tasks, sub-resources, sub-goals and sub-softgoals. Each sub-component has the same definition as for the Strategic Dependency model.

A **task** can be decomposed into:

- A **task**, expressing "a particular way of doing something" and thus constraining the high-level task. A practical example (see figure 12) of a task that is decomposed into a sub-task, would be to constrain the "Process Order" task, performed by the Order Processing Clerk, by a task "Verify Order". This would mean that the Order Processing Clerk has to verify the order in a particular way.

- A **resource**, that can be a physical or informational entity considered as important (i.e. the availability is not certain). In Figure 12, the verification by the Order Processing Clerk of the order requires the list of actually sold items.
- A **goal**, that has to be achieved in order to perform the high-level task. The defintion of goal is still the same as indicated in the Strategic Dependency section, i.e. a condition or state of affairs in the world. Contrary to the task-task decomposition, the way of doing things in order to achieve the goal is not specified. The goal decomposition is often used to indicate that there may be different alternatives for achieving the condition. Thus different alternatives can subsequently be explored. For our example, the processing of the order could also depend upon the fact that the Order Processing Clerk could also require that the account of the customer is above 1000$.
- And finally a **softgoal**, where the condition that should be brought upon is not yet explicitly defined. Softgoals are used to express quality goals for that task and can then be used for the evaluation of alternatives. In our example, one could indicate that there should be a fast turnaround for the task and thus also for the sub-task and sub-goals are influenced by this softgoal..

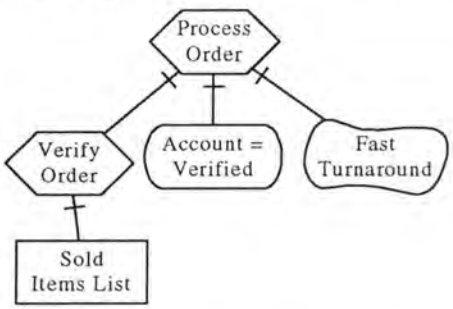


Figure 12: A Task Decomposition

3.2 The Means-Ends links

Different alternatives can be represented in the Strategic Rationale Model by Means-Ends links, where "the end can be a resource, a task, a goal or a softgoal, whereas the means usually is a task." There exist 5 types of means-end relationships:

- **Goal-Task** means-ends relationships have as a means a task and as an end a goal. In figure 14, the Order Processing Clerk has different possibilities in obtaining the goal "Account Verified": firstly, he can look by himself if the account is all right and secondly, he can ask a bank clerk whether the account is in order.
- The **Resource-Task** link identifies a resource as an end that has to obtained and a task as a means by which the resource can be produced. (see Figure 13)

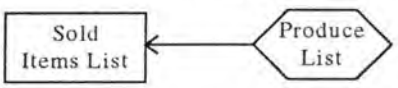


Figure 13: A Resource-Task Link

- A **Task-Task** link has a task as a means and as an end. Unlike the decomposition of tasks, where the higher level task can be defined as an aggregate of its sub-tasks, there exist many ways (alternatives) of constraining a task furthermore. This is possible since the description of a task can be incomplete, and thus making it possible to choose between alternatives.
- A particular configuration is the **Softgoal-Task** Link, where the means is a task and the end a softgoal. These links identify **contributions** (negatively, positively, enough or not enough) to softgoals, that can then be evaluated for the decision process.
- Softgoals also can be decomposed into sub-softgoals by **Softgoal-Softgoal** links until they can be addressed by some tasks (shown by a contribution link in figure 14).

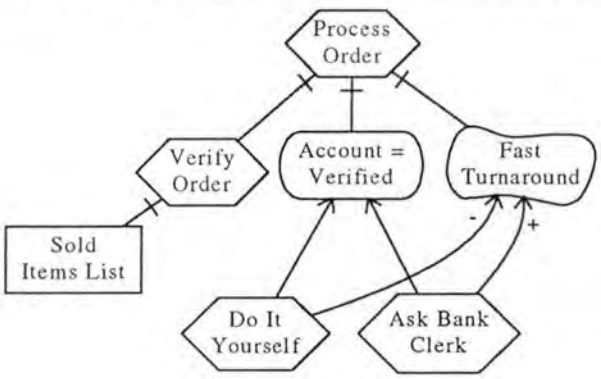


Figure 14: A Means-Ends Link

3.3 Routines

The Means-Ends Links described above identified different alternatives for achieving one end. Once an alternative is chosen, it can be developed furthermore by the routine representation, which is a sub-graph with only one Means-Ends Link. This link then refers to the representation of one possible solution for achieving the end.

4. Knowledge Base Representation

Both the Strategic Dependency Model and the Strategic Rationale Model have been embedded into the conceptual knowledge representation language Telos. Consequently, it benefits from the conceptual structuring mechanisms such as generalisation, aggregation and classification and linguistic extensions through the definition of meta-attributes.

5. Process Modelling

After having defined the vocabulary of *i**, let us highlight two main characteristics:

- Contrary to other process models, *i** captures besides the 'how' also the 'why' of the process. By offering the means-ends mechanism which is goal oriented, *i** provides the analyst with the representation of rationales. If we take for example figure 14, we can know how the goal "Account = Verified" is achieved. Secondly, *i** provides the explicit representation of alternatives and the reason for the choice of one of them. This means-ends mechanism also offers the opportunity to explore and analyse in depth different alternatives.
- Contrary to other process models, where only the decomposition of tasks is taken into account, *i** enlarges this by letting the analyst choose between different types of decompositions. Thus, a task can be decomposed into a sub-task, a sub-resource, a sub-goal or a sub-softgoal. This decomposition, in addition, concentrates on strategic importance, and so the decomposition does not necessarily have to be complete.

6. Application Area: Requirements Engineering

On reviewing the suitability of *i** in the Requirements Engineering domain, Yu concentrates on the 'Early Requirements' phase, where the requirements document is elaborated. Indeed, most formal specification languages assume that there exist already requirements statements in an informal manner or natural language. But this statements do not express how and why these goals should meet some organisational goals. Yu identifies four major concerns that should be met by the early requirements:

- Develop a deep understanding about a domain. *i** could help here by showing the rationales that are behind processes and does not only concentrate on entities and activities as most other modelling languages do. In addition, the representation of rationales and motivations may also lead to a better description of the process itself.
- Help in coming up with initial requirements. The exploratory way in which requirements are elaborated and analysed, the analysis of opportunities, vulnerabilities and strategic concerns helps in finding out the 'real problems'.
- Trace the changes of the requirements back to the organisational changes. This exploratory way also helps in tracing back to the original goals.
- Share knowledge about domains. As we already saw in the previous section, *i** is implemented in Telos, a conceptual knowledge representation language. The use of a knowledge-based approach facilitates the collection, organisation, use and reuse of domain knowledge across cases and across domains.

CHAPTER 3
THE ALBERT FRAMEWORK

This chapter presents the second language used in our methodology. ALBERT (an acronym for **A**gent-oriented **L**anguage for **B**uilding and **E**liciting **R**eal-**T**ime requirements) was developed within the Esprit ICARUS project by Eric Dubois and his team at the University of Namur. Several aspects can be used to describe their motivations:

- the introduction of agents together with their properties (internal states, responsibility for actions, perceptions of the environment). Furthermore, agents can be grouped into classes or societies. This "object-oriented" approach can also help in structuring large specifications.
- the introduction of actions to overcome the well-known frame problem [Borgida et al,92],
- the identification of typical patterns of constraints which support the analyst in writing complex and consistent formulas. Especially the cooperation constraints are of great importance since several agents have to interact in order to fulfil the overall goal.

In 1995, Philippe Du Bois presented in his thesis the second version of the language. At the present time the revision and extension of the language is still going on. We will present the latest version: ALBERT II(version 2.0). For a more detailed description, one can look at [DuBois,95], [DuBois,97].

1. About ALBERT

The ALBERT language has been elaborated with respect to the following characteristics [DuBois,95], [Dubois et al, 95]:

- degree of formality,

- expressiveness and naturalness,
- intended application domain.

1.1 Formality

The first Requirements Engineering specification languages like SADT,... were semi-formal languages that integrated two kinds of descriptions:

- a structured description expressed in terms of the language. Likewise the SADT model differentiates between data and transformation. Although these concepts were related to a formal language, these formal languages consisted mainly in a concrete, formal syntax of the concepts: only the representation of the concepts and the possible combinations between the concepts were defined. The major drawback with these languages is that the concepts do not always have a precise semantics. Consequently, the interpretation of such descriptions is often left to the analysts.
- an unstructured, textual description. These languages have a restricted set of concepts and relationships between these concepts and they cannot express every property. As a result the analyst has to use informal (even not syntactically formal) descriptions in natural language to express these properties.

Furthermore, in order to built more efficient CASE tools, which have more sophisticated checks and assistance, it is necessary to provide modelling languages with formal semantics. Having a formal language, one can look for inconsistencies or validate through animation, for instance.

ALBERT is fully formal as its semantic relies on a variant of temporal logic.

1.2 Expressiveness and Naturalness

ALBERT was designed considering the fact that a Requirements Engineering language should be expressive, i.e. have a rich "ontology of concepts by which the requirements can be expressed". This ontology is always influenced by the domain in which the modelling takes place. This environmental issue has as a consequence that not only software artefacts should be modelled during the RE phase but also the environment of a system. Users of the system, devices of the system and other hardware components are just as important as the pure specification of the software for the requirements documentation and are modelled as agents in the ALBERT specification.

Furthermore, the concept of naturalness is highlighted in ALBERT. In order to restrict over-specification, the mapping between the users requirements and the specification statements should be natural, i.e. the language should offer possibilities to map straightforwardly the informal statements provided by customers, onto formal statements expressed in the RE language.

1.3 Application Domain

Like in the acronym indicated, ALBERT is primarily proposed for modelling real-time distributed systems. Like other languages, ALBERT enables the modelling of composite and distributed systems where activities "run in parallel with possible communication/interaction".

1.4 Models[1] of a Specification

The purpose of ALBERT is to describe admissible behaviours of a composite system. This description, called *specification* of the system, must abstract irrelevant details. If the specification describes faithfully the system, admissible behaviours of the system will be *models* of the specification. The model does not refer to the whole specification but to a mathematical interpretation structure associated with a logical theory.

The semantics of ALBERT are defined as a set of rules that define the set of admissible lives from a given specification. The precise rules can be found in [DuBois,95].

A model of an agent specification refers to one of its possible lives and consists in an alternate sequence of *changes* (represented by ovals in figure 15) and *states* (represented by ovals in figure 15).

Each change is tagged with a real-time value which increases throughout the sequence. The *time stamps* of changes are not necessarily equidistant. Time stamps reflect a point of time where something happen in the system. They are represented by circles in figure 15.

A state is structured according to the information handled in the considered application in terms of state components. The value of state components stays unchanged between two adjacent changes.

A change is composed of simultaneous events. An event corresponds to:

- either the occurrence of an instantaneous action (in figure 15: '<Add(a)>')
- either the beginning of the occurrence of an action (in figure 15: '<Remove(a)')
- or the end of the occurrence of an action (in figure 15: 'Remove(a)>').

Only the occurrence of an action may induce a change of a state. The value of a state at a given time in a certain life can therefore always be deterministically derived from the changes occurred so far and the initial state.

Figure 15 shows a portion of a possible life having as state components a stock and an 'on hold' stock that have to be manipulated. Two actions may change the value of the state components 'Stock' and 'OnHold': 'Add(x)' and 'Remove(x)'. For this example, 'Add(x)' is an instantaneous action and only has a post-effect. 'Remove(x) can last over a

1 The word Model is used here in a different sense as we use it in the rest of the work when we talk about the model, i.e. the specification itself or the i* model.. In Albert, we use the word Model in the sense of a mathematical interpretation structure associated with a logical theory.

period of time and has a pre- and post-effect. The 'Effect of Actions' constraints look like this:

EFFECT OF ACTION

Add(x):

[ ]

Stock := Add(Stock, x)

Remove(x): *Stock := Remove(Stock,x)*

[ ]

OnHold := Add(OnHold,x)

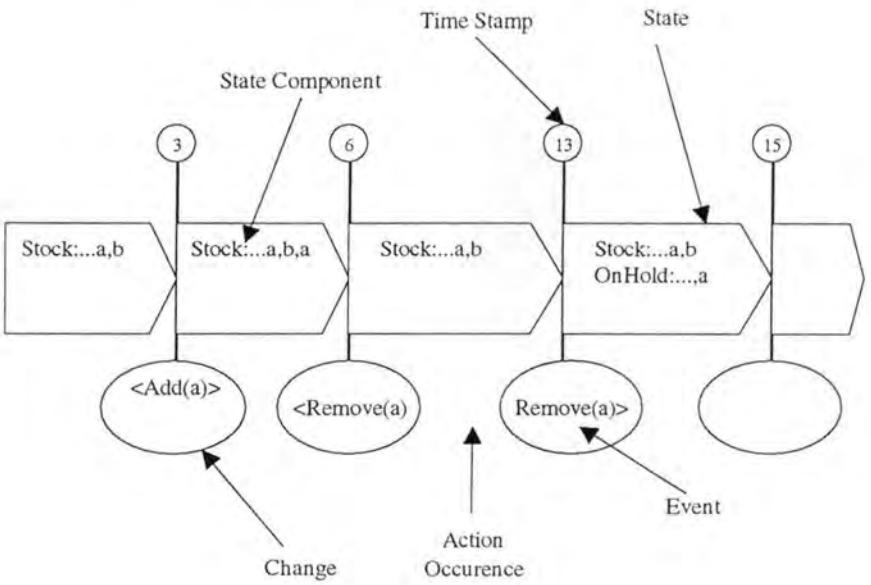


Figure 15: A possible life of an agent

1.5 State-based and Life-based Constraints

Many of the existing software design languages only support state-based constraints. State-based constraints express a relation either between two consecutive states, or between a state and the change happening in that state, or between a change and the resulting state. So, describing constraints only consists in describing properties of transitions. This is found to be inappropriate for specifying requirements because this induces the 'over-specification' problem and complicates the traceability between the specification and the informal statements. Consequently, ALBERT was designed to offer both operational and declarative (life-based) styles of specification. Life-based constraints relate changes or states spread all over an agent life.

In comparison with the previous version of ALBERT, where already state-based and life-based constraints were well supported, the differentiation between the two styles of specification has been emphasised in the new version where the so-called "Local

Constraints" group of headings has been split into the "Declarative Constraints"(for life-based constraints) and the "Operational Constraints"(for state-based constraints) families.

2. Specifying in ALBERT

An ALBERT specification is made of the definition of data types and associated operations, the definition of societies, the definition of the agents with their properties (state components, actions and constraints). For the description of the language, we consider the following structure:

- declaring the data types and their operations,
- declaring societies,
- declaring the structure of agents (states and actions),
- constraining the agents.

Running Example. In the following, we introduce to the different concepts. To clarify the different concepts, we use the specification of a Stock Clerk as a running example: On receiving an order from the Office Clerk (represented in our example by an 'O') to ship items, the Stock Clerk removes the item from the stock and ships it to a customer (via a mail company 'Mail').

2.1 Declaration of data types and the associated operations

ALBERT is a typed language relying upon the use of abstract data types for structuring information (state components & action arguments). Furthermore, operations on these data types can be defined. These operations should be understood as mathematical functions rather than "ALBERT-actions". There are several forms of data types; namely predefined data types and data types defined by the user. These latter can be elementary or constructed.

BASIC TYPES
ITEM
ITEMTYPE
CONSTRUCTED TYPES
INVOICE = CP[OrderId: INTEGER, Person: STRING, Item: ITEMTYPE, Sum: INTEGER]

Figure 16: Declaration of Data Types

2.1.1 Predefined Data Types

Predefined basic Data Types are:

BOOLEAN	CHAR	STRING
INTEGER	RATIONAL	DURATION

These data types are fitted out with their usual operators like ¬ (not), ⊗ (and), ⊕ (or) for the Boolean operators or + (plus), - (minus) or mod (rest by the division) for the integer. For an exhaustive list refer to the reference manual [DuBois,97].

2.1.2 User-Defined Elementary Data Types

Since a requirements specification language should be expressive, one cannot be satisfied with the predefined data types. The user of ALBERT is given the opportunity to define himself *basic data types* which, by default, only have the two equality operators (= and ≠) as predefined operators. It is also important to note that basic type names are all uppercase and that they should be unique for the whole specification. From a practical point of view, basic types are declared by writing them down under the BASIC TYPES heading (see our example in figure 16).

2.1.3 User-Defined Constructed Data Types

ALBERT also offers the facility to construct more complex data types upon the predefined and user-defined elementary data types. This construction can be built by using a set of predefined type constructors:

- CP for the declaration of a Cartesian Product,
- SET for the declaration of a set (unorder),
- BAG for the declaration of a bag or multiset,
- TABLE for the declaration of a table (indexed bag),
- UNION for the declaration of a union (merging of items) and
- ENUM for the declaration of an enumeration of values.

Using these constructors, one can declare new types under the CONSTRUCTED TYPES heading. Just as for elementary data types, constructed type names are all uppercase and they should be unique for the whole specification. In Figure 16, we represented such a constructed data type: 'Invoice' that is a cartesian product with 4 fields.

2.1.4 User-Defined Operations

Because the newly declared types are only, by default, provided with the '=' and the '≠' operator, there is a mechanism for providing the specification with operations associated to the user defined types. "They are mathematical functions returning a result from zero or more arguments" and they can be constrained by the use of first order logic formulas. User defined operations appear under the OPERATIONS heading and are defined for the whole specification.

2.2 Declarations of Societies

As already mentioned, ALBERT is a language for specifying composite systems and thus it is very logical that the language offers a way to define hierarchies of agents that differentiate between:

- individuals or classes of agents,
- societies.

An individual agent represents one and only one instance in the model whereas classes have at least one instance of the agent in the model. Classes of agents can be understood as a set of individual agents having the same behavioural properties although each agent of a class can be identified. By the means of these components, one can construct a hierarchy of agents, where the terminal leaves are individual agents or classes of agents. These leaves can then be regrouped into societies. Thus, societies form aggregates of different agents (individuals, classes and societies). In an ALBERT specification every agent or society must be part of one and only one society (with the exception of the root agent/society).

Agents, classes and societies have a graphical representation and a textual syntax has been defined. In our example (figure 17), the society 'Company' is made of a stock clerk, an office clerk and a mail agent.

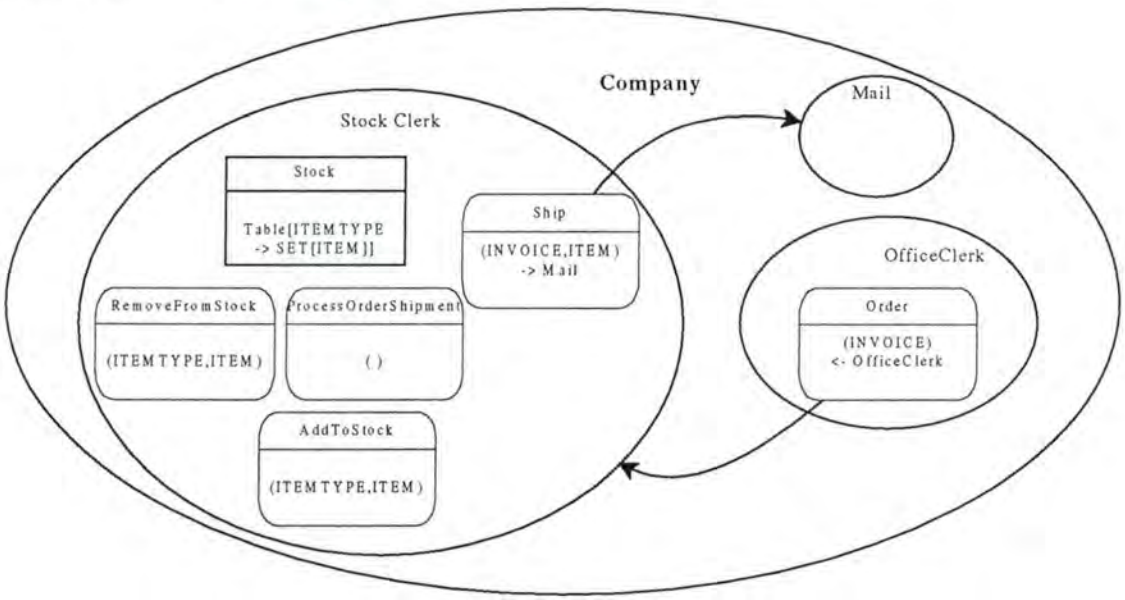


Figure 17: Partial Declaration of a Society

2.3 Declaration of Agents

ALBERT is an 'object oriented' approach in the sense that state components and actions are encapsulated in one unit (agent). Although, there is a big difference with object oriented programming languages like C++ or Pascal 5.5: mechanisms as heritage, specialisation, hierarchy of types and polymorphism are not provided. Furthermore, an

operational specification and a declarative specification style is supported by ALBERT. The word 'agent' has been preferred since the entities of a system have responsibilities and perceptions.

The declaration of an agent consists in two things:

- declaring their state components and
- declaring their actions.

This can be done textually (figure 18) and graphically (figure 19).

STOCKCLERK

DECLARATIONS

STATE COMPONENTS

Stock table-of *SET[ITEM]* indexed-by *ITEMTYPE*

ACTIONS

ProcessOrderShipment

RemoveFromStock(ITEMTYPE,ITEM)

AddToStock(ITEMTYPE,ITEM)

Ship (INVOICE,ITEM) → *Mail*

Figure 18: A Textual Agent Declaration

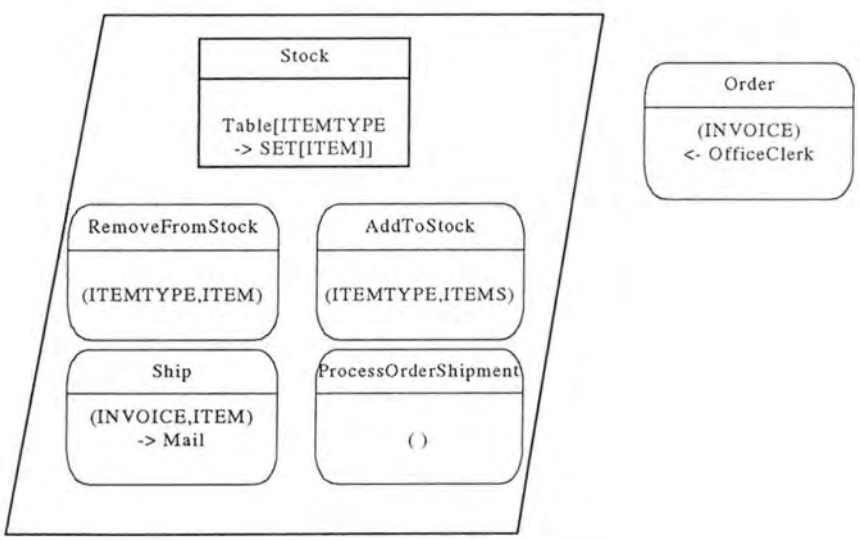


Figure 19: A Graphical Agent Declaration

2.3.1 Declaration of the state components

By the means of state components, we describe the knowledge the agent has about himself and its environment. A state of an agent is structured by state components. An agent can have zero or more state components which can be individuals or populations. A population can be a set, a sequence or a table.

State components can be declared textually:

	Textual representation
Individual	*Name* instance-of *TYPE*
Set	*Name* set-of *TYPE*
Sequence	*Name* sequence-of *TYPE*
Table	*Name* table-of *TYPE2* indexed-by *TYPE1*

Components are time varying by default but can be declared to be constant. Constant components are graphically displayed by bold line boxes and textually by prefixing the component with a star (*).

In order to simplify the description of expressions, one can use derived components. These derived components must be declared by relating the component to its 'causes'. ALBERT does not accept any recursive declaration of derived components. Again, there exists a textual notation (*StockEmpty* derived-from *Stock)* and a graphical one which is represented by an arrow from the depending state components towards the derived component.

By default, agents do not perceive state components belonging to other agents. Another fact that has to be explicitly declared is the static importation/exportation mechanism; (i.e. whether components may be made visible to other agents). When state components are declared to be exported to another agent, the latter may perceive this state components. If a state component is made visible for a society, it is visible for all the agents and societies within the society. A component of a class of agents can also be exported to the members of this class.

The textual declaration under the heading STATE COMPONENTS describes the agents to whom the state components are shown. Graphically, the exportation/importation mechanism is shown by an arrow to the inside/outside of the agent parallelogram.

Note that the declaration of an exported/imported state component does not necessarily imply that the state components are shown/perceived by other agents since this perception/information may vary over time (depending on the state of agents). The dynamic counterpart of the Importation/Exportation properties is the Perception/Information mechanism, which is expressed in the co-operation constraints: STATE PERCEPTION and STATE INFORMATION.

2.3.2 Declaration of the actions

State components cannot change by themselves and one has to introduce actions that can change them. This should, however, not lead to the conclusion that agents must have actions. Just as for the state components, agents may be declared without having any actions. Actions are declared graphically by a rounded-corner box, within which the types (as a sequence) of the eventual arguments are also declared. Textually, actions are declared by writing them under the ACTIONS heading.

Actions can be instantaneous or last over a period of time. Actions have a duration by default and instantaneous actions are represented in the declaration with a preceding star

(*). Actions can be elementary or decomposed into more elementary actions. This decomposition fact is not represented in the declaration part.

Actions can also be declared as being exported to other agents. If an action is made visible for a society, it is visible for all the agents and societies within the society. An action of a class of agents can also be exported to the members of this class. Graphically, actions that are exported towards other agents are declared with an arrow that is drawn to the outside of the agent box. In the textual form, the respective action is followed by an arrow '→' and the importing agents and societies.

In our example (see figure 18 and 19) the 'Ship'-action is exported to the Mail-Company.

2.4 Constraints

"Constraints are used for pruning the infinite set of possible lives of an agent." They are classified into 14 headings and regrouped into 4 families:

- basic constraints,
- declarative constraints, which adopt a life-based point of view,
- operational constraints, which adopt a state-based point of view,
- co-operation constraints, that describe the interface with the agent's environment.

STOCK-CLERK

DECLARATIONS

STATE COMPONENTS

Stock $\overset{table\text{-}of}{}$ *SET[ITEM]* $\overset{indexed\text{-}by}{}$ *ITEMTYPE*

ACTIONS

ProcessOrderShipment

Ship (INVOICE,ITEM) → *Mail*

RemoveFromStock(ITEMTYPE,ITEM)

AddToStock(ITEMTYPE,ITEM)

BASIC CONSTRAINTS

INITIAL VALUATION

Stock[_] = {}

DECLARATIVE CONSTRAINTS

ACTION COMPOSITION

{ O.OrderShipment, RemoveFromStock, Ship}

ProcessOrderShipment ↔ *O.OrderShipment(inv)* <> (

(RemoveFromStock(Item(inv), it) <> *Ship(inv, it))*

⊕ *DAC)*

OPERATIONAL CONSTRAINTS

PRECONDITION

RemoveFromStock(i,_) : Card (Stock[i]) > 0

EFFECTS OF ACTIONS

AddToStock(i,it) : *[] Stock[i] := (it ∪ Stock[i])*

RemoveFromStock(i,it) : *[] Stock[i] := (it \ Stock[i])*

COOPERATION CONSTRAINTS

ACTION PERCEPTION

K (O.OrderShipment(_) / TRUE)

ACTION INFORMATION

K (Ship(_,_).Mail / TRUE)

Figure 20: ALBERT Constraints

2.4.1 Basic Constraints

Under this header we describe the initial values for state components and the derivation rules for derived components. We adopt thus the following categories:

- Initial Valuation

  The constraints written under the INITIAL VALUATION header indicate the value of state components in the first state of the trace. These constraints are optional so that there can exist state components for which no initial valuation is given. But for every state component that is initialised there can only be one initialisation and a derived component cannot be initialised.

  In our example (see figure 20) the constraint expresses that at the beginning of a possible life of an agent, the stock is empty.

- Derived Components

  A derived component is a facility by which we can express complex mathematical relationships between state components of the same agent. It is primarily a mechanism to simplify expressions. This derivation rule is obligatory for every state component declared as a derived component and the components used for describing the rule do also have to be indicated in the declaration of the derived component. A constant component can only be derived from constant components.

  It is important to note that components can only be derived from internal state components and hence no imported state components can appear on the right-handed side of the derivation rules. Neither can derived state components appear on the right-handed side of the rule.

  This type of constraint is not used by our running example but we might introduce for example a derived state component that is true if and only if the stock is empty: 'StockEmpty'.

  DERIVED COMPONENTS

  StockEmpty[i] ≙ Empty(Stock[i])

2.4.2 Declarative Constraints

To stress the fact that ALBERT deals both with constraints that are valid for the whole agent life and constraints on possible transition of states in an actors life-cycle, the previous "Local Constraints" group was split into declarative and operational constraints. Declarative constraints relate distant states / actions of the whole life of an agent.

- State Behaviour

  A first type of declarative constraints is used for expressing the 'behaviour' (i.e. the possible evolution) of a state component during the whole life-cycle of the agent. These constraints are expressed using real time temporal logics. Temporal logics use the usual first order logic connectives (see figure 21) and additional real time temporal connectives like those introduced in the TRIO language.(see figure 22).

And	$\otimes$	if and only if	$\Leftrightarrow$
or	$\oplus$	for all	$\forall$
implies	$\Rightarrow$	there exists	$\exists$

Figure 21: First Order Logic Connectives

ϕ always true	`Alw` ϕ	ϕ true for one moment	`Som` ϕ
ϕ always true in the future	`AlwF` ϕ	ϕ true for one moment in the future	`SomF` ϕ
ϕ always true in the past	`AlwP` ϕ	ϕ true for one moment in the past	`SomP` ϕ
ϕ true until ψ becomes true	ϕ `Until` ψ	ϕ true until ψ becomes true and ϕ becomes false at that moment	ϕ `Until!` ψ
ϕ true since was true	ϕ `Since` ψ	ϕ true since ψ was true and ϕ was false at that moment	ϕ `Since!` ψ

Figure 22: Real-Time Connectives

Using these connectives, one can build different constraints on state components and thus restrict the set of possible lives. One can also specify that a constraint applies only during the occurrence of an action which appears between brackets "[ ]" in front of the formula. If no action appears between brackets, the constraint has to be true at any time.

In our example (figure 20), the number of items never can be negative and during the action 'RemoveFromStock' (i.e. for all state between the beginning and the end of the occurrence of the action) the stock should not be negative or equal to zero.

- Action Composition
  The relationship between occurrences of actions is described under this header. Action composition constraints allow the analyst to express processes by decomposing 'complex' actions into more 'elementary' actions. Elementary actions are called elementary when they cannot be decomposed furthermore.

  The decomposition of an action expresses the possible temporal ordering of sub-actions by using the following connectives:

Types	Notation
Sequential Composition	$a \leftrightarrow a_1 < d > a_2$ *where d is a duration*
Repetitive Composition	$a \leftrightarrow \{ a_1 \}^k$ *where k is the number of repetition*
Parallel Composition	$a \leftrightarrow a_1 \mid\mid a_2$
Simultaneous Composition	$a \leftrightarrow a_1 \mid\Leftrightarrow\mid a_2$
Costarting Composition	$a \leftrightarrow a_1 \overset{\rightarrow}{\mid} a_2$
Cofinishing Composition	$a \leftrightarrow a_1 \overset{\leftarrow}{\mid} a_2$
Alternative Composition	$a \leftrightarrow a_1 \oplus a_2$

  On the left-hand side of these constraints can only appear internal actions whereas on the right-hand side both internal and external actions can appear. Consequently, processes may be spanned over several agents by the export/import mechanism of actions.

  The "`with`" clause allows to specify constraints about the arguments of the actions.

  Actions can also be restricted to appear only within a composed action; this is expressed by listing theses actions at the top of the Action Composition template between brackets "{ }".

  It is also important to note that action compositions cannot represent cycles. A special action identifier `DAC` may also be used in alternative compositions as dummy action which then represents 'nothing'.

  In our example (see figure 20) the action 'ProcessOrderShipment' is decomposed into an 'OrderShipment', 'RemoveFromStock' and 'Ship' action. The process begins by a request action from the 'Office Clerk', that is followed by the removal of the respective item and the shipping action. All the actions listed above can only occur inside their respective composed action.

- Action Duration

  Constraints about the duration of internal action occurrences may be expressed into this template. Several operators can be used: "=" (equal), "≠" (different), "≤" (less), "<" (strictly less),"≥" (greater),">" (strictly greater). In addition, the "`with`" clause can be used.

  The following expression states that the 'RemoveFromStock' action lasts longer than 1 minute.

ACTION DURATION

|RemoveFromStock(_)| > 1 '

2.4.3 Operational Constraints

This family regroups the headings that are evaluated in an historical way; the evaluation of the formula does not consider the whole life-cycle but only the past.

- Preconditions

  This heading regroups formulas expressing conditions that have to be verified for an occurrence of an action. The scope of the condition can be restricted using a "`with`" clause.

  In our example (see figure 20), the 'RemoveFromStock'-action can only occur when the stock for the respective item is not empty.

- Effect of Actions

  Because state components cannot change by themselves, this heading is extremely important since the constraints appearing under this heading do express the effect of action occurrences upon internal state components. Actions, that can have an effect upon state components, can be internal or imported: also other agents may change components of one agent.

  The general notation is as follows:

  Action-Name: Pre-Valuation [Formula] Post-Valuation

  The constraint is divided into pre-effects and post-effects.

  Usually, post-effects are used for describing the effect of an action and thus the effect generally appears at the end of an occurrence of an action. If the formula is verified just before the end of the action, the state components just after the end of the action takes the value as expressed in the post-valuation. The post-valuation is evaluated with the values of components as they were just before the beginning of the occurrence of the action. If the formula is false, then the state components are left unchanged.

  It is also possible to express pre-effects: pre-effects describe the change of a state component just after the beginning of the action occurrence and they are evaluated with respect to the value of the state components just before the occurrence of the action. Pre-effects are temporary: they are undone when the action ends unless the post-effects state otherwise.

  In figure 20, we only use the post-effect as a means for expressing the change by the actions 'AddToStock' and 'RemoveFromStock' upon the state component 'Stock'.

- Triggerings

  This heading is used to express obligations of the occurrence of an action; i.e. a condition under which an action has to occur and which has to hold for a defined period of time. We did not define any triggering in the figure 20. An example might be to trigger an alarm for the supplier of the stock if the stock has been empty for one day.

  TRIGGERINGS

  StockEmpty / 1 day $\rightarrow$ *Alarm.Supplier*

2.4.4 Cooperation Constraints

In the declaration part, we already talked about importation and exportation. These were static properties. Their dynamic counterpart can be found under the cooperation constraints header, which is used to express how agents interact with their environment in a time varying manner. Not only state components may be made visible towards other agents and be perceived by other agents but also other agents can be informed of the occurrence of internal actions and external actions can be perceived.

Three connectives are offered for the expression of cooperation constraints:

- $\mathcal{K}$: the knowledge pattern, which defines the condition for the perception/information of a state component/action. If the condition is true then the state component/action is perceived; if the condition is false, the state component/action is perceived or not.
- $\mathcal{I}$: the ignorance pattern, which expresses the condition for which external state components/actions are not perceived.
- $\mathcal{XK}$: the exclusive obligation pattern, expresses the condition under which and only under which the state components/actions are perceived.

The conditions are expressed using temporal formulas which may refer to state components.

The following table summarises the different concepts:

	$\mathcal{K}$	$\mathcal{I}$	$\mathcal{XK}$
ϕ	Transfer !	No transfer !	Transfer !
$\neg\,\phi$	Transfer or not ?	Transfer or not ?	No transfer !

When no information/perception constraints are expressed, the exported actions/state components may or may not be visible or perceived.

- State Perception

  Here we express under which conditions state components belonging to other agents are perceived.

In figure 20, no state components are made visible towards the stock clerk: he has no knowledge about external state components.

The "`with`" clause can also be used to limit the scope of the constraint.

- State Information

  Under this heading we define when state components are made visible for other agents.

  In figure 20, no state components are made visible towards the other agents. Hence, the state component 'Stock' and their value stays private to the Stock Clerk.

  The "`with`" clause can also be used to limit the scope of the constraint.

- Action Perception

  Actions belonging to other agents can also be perceived. Depending upon the evaluation of the formula and upon the type of cooperation connective, the actions are perceived or not.

  In our example (see figure 20), the 'OrderShipment'-action is always perceived. This is represented by an '$\mathcal{K}$' and the constant 'TRUE' in the formula.

  The "`with`" clause can also be used to limit the scope of the constraint.

- Action Information

  The counterpart of the Action Perception constraints are the Action Information constraints.

  In figure 20, the stock clerk informs the mail agent about the occurrence of the 'ship' action.

  In practice, the action perception and information mechanism can be used to define processes that span different agents.

  The "`with`" clause can also be used to limit the scope of the constraint.

CHAPTER 4
THE METHODOLOGY

The two previous chapters presented the two specification languages the thesis is based upon. The first, ALBERT II, aims at highlighting the 'what', the functional requirements of the system and the second, *i**, concentrates on the modelling of the 'why', the organisational issues and the non-functional requirements. Although, the languages are the basic tools for writing a requirements specification, their use does not necessarily lead to a 'good' requirements specification; especially when the analysts are inexperienced with the language or in developing user-oriented specifications. In addition, the elaboration of requirements specification is a particular critical task because two worlds are confronted: the user/customer world and the analyst (engineer) world. Furthermore, the analyst is often responsible for resolving conflicting situations between diverging viewpoints of the customers and direct users. In this chapter, we are going to describe certain basic steps a requirements document should go through before we can finally talk about a specification. We divide our system development method into three main stages: first the problem domain modelling stage, where the model of interest is represented (section 1), then the system requirement stage where the functional and non functional requirements of the system are stated (section 2) and finally the system specification stage where we map the model back to the real world and specify the system internals. (section 3)

1. Problem Domain Model and Objective

The goal of this stage is to produce the definition of the problem domain. This problem domain should supply us with the basic elements, vocabulary which will serve for a first definition of the problem and the objective.

In section 1.1, we will talk about some general aspects of the problem domain. Section 1.2 will describe the approach we choose in ***i**** for modelling the real world. Section 1.3

and 1.4 are going to deal with respectively the identification and the description of the behaviour of the ALBERT agents. Section 1.5 finally will show the introduction of the objectives of the system using the elements of the problem environment.

1.1 Preliminary: The Problem Domain

The necessity for defining the Problem Domain has been well recognised for a long time (see e.g. [Jackson,83]). This is different from an approach like, e.g. SART, which focuses on the modelling of the system internals. Jackson describes several advantages about the early modelling of the real world.

To start with the functional specification of the system internals is necessarily ambiguous because many undefined terms are used. The early modelling of the real world introduces a set of words whose meaning is related to the real world entities and is thereby less ambiguous. This argument will also lead us to the conclusion that every noun introduced from the real world should be completed by an informal definition.

Furthermore, when analysts get knowledge about the environment, their communication with the user is improved. Indeed, a lack of knowledge or a misunderstanding of the domain is often the root for dissatisfaction on the client's side. In addition, considering that analysts are most often 'reconfigured' programmers, they often deliver technical specifications which will be of no help for users. However, the requirements specification should be regarded as the contract that circulates between the client world and the analyst world and be understandable for each side. The introduction of the real world modelling only considers aspects of the client's world.

A principal motivation for Jackson is that the model of the real world is more stable than the functional specification. The functions, an Information System should support, change rapidly, but the model will not change so often because the functions can be based upon the model. Jackson makes here the comparison with a road map and journeys. The road map can be matched with the model of the real world and journeys with functions of an Information System. A change of the map will affect the search for journeys, contrary to a change in a journey which will have no affect on the map. Just as for the map, the building of the model of the real world is concerned with elaborating the basic elements with which a future function can be 'calculated'. Consequently, model and function are inevitably interconnected. The functional approach only treats the model implicitly although it should be treated explicitly.

In addition, this approach also goes along with the traditional Mintzberg-approach [Mintzberg et al,91] for decision making. Indeed, the elaboration of requirements document can be compared with a decision making process: 'for a given problem define what to do to solve the problem'. This decision process starts with the perception of organisational issues, problems or crises. After this accumulation of stimuli, a diagnostic routine is initialised. Existing information is gathered and new information is created. Then only, the development phase is started by the search routine and the design routine.

Although Jackson's argumentation is valid, it might also be reviewed; we believe that more is needed.

The modelling of the processes that the future system should support might increase the knowledge about the process itself, but one can seriously doubt whether this modelling, prior to the software engineering phase leads to an important increase of understanding about the domain. Does the partial knowledge of a program's functions necessarily imply the understanding of the program? Is it not also important to describe the rationales, motivations and intentions that are behind a behaviour? Rationales of a process should also be considered and explicitly stated. Otherwise the interpretation of the intention of a process will lead to ambiguities. However, these rationales are often not restricted to the entity responsible for a behaviour.

We furthermore believe that the requirements specification should not only represent the environment communicating with the future system. The system might not only have responsibilities or non-functional requirements towards the entities[2] that directly interact with the system. Dependencies may also exist without existing communication. Besides this, dependencies might also be of interest which are beyond the direct requirements. These can then be used to explain the *'why'* of the requirements of the system. Indeed, the requirements that a user has upon a system are often not based upon simply internal motivations. External motivations, responsibilities towards other agents, might influence these internal intentions.

Although a detailed (formal) description of entities in organisations might be 'nice', it often results in a model that is too cumbersome. We will use here the properties of *i**. Indeed, *i** does not have to be complete and limits its scope to strategic aspects inside an organisation unlike data flow charts, for instance, that are too detailed. However, for the immediately interacting actors of the real world, we consider a more detailed approach because their behaviour may be changed. Interacting actors are the ones that will communicate with the future system. The modelling of their behaviour will be the role of ALBERT.

We also believe that the client[3] side of an information project is not always consistent about what is desired. Customers, assigning the project, might have other, conflicting, requirements than the users, who will interact later on with the system. We believe that multiple viewpoints have to be considered. These could be represented by possibly conflicting dependencies towards the new system.

With respect to the discussion above, we divide the purpose of this first step into two goals to achieve:

- a model of the organisational issues that goes beyond the pure modelling of system-interacting entities of the real world, modelled by the *i** framework and
- a model of the interaction issues, modelled by the ALBERT framework.

[2] In general, we use the term entity when we do not want to refer to an *i**-actor or an ALBERT-agent. In this context, entities refer to the real world.

[3] We identify the client side by the customers, assigning the information project, and the users, interacting directly with the future system.

1.2 Identification of the real world in *i**

As said above, ***i**** and particularly the Strategic Dependency model is specialised in modelling the strategic relationships in organisations. We will not only concentrate on the actors that might directly interact with the future system. This first step has more to do with organisational modelling than anything else. Defining business objectives and their decompositions in organisational responsibilities and dependencies is important because:

- information projects should fit with existing business strategy,
- information projects should provide a competitive advantage,
- information projects should try to exclude organisational risks.

The modelling of these actors will provide us with the 'indirect' objectives and the non-functional requirements that did not actually lead to the desire to produce a solution to a problem but that provide the objectives that a future solution has to satisfy.

The first step is also characterised by the first contact with the organisation. The analyst is potentially new to the domain. This implies a lack of knowledge and understanding about the problem domain and hence about the possible solution that should satisfy the client.

The typical ***i**** approach (from the Strategic Dependency Model to the Strategic Rationale Model) can be characterised as 'outside-inside': first, the actors with their relationship are identified and only then the internal behaviour of the actors is described. On the one hand this has the advantage that the analyst does not have first to restructure the 'internal mess' of the actor's behaviour. On the other hand, the internal analysis often reveals further dependencies. Indeed, how can we say that we depend upon someone if we do not know what for? An actor can have an important number of dependencies but we are only interested in certain relationships, related to some specific processes.

1.2.1 Actors in *i**

As we noticed already in the introduction of ***i****, the Strategic Dependency model consists of a set of actors inter-linked by dependency relationships.

Intentional Actors. Typically, ***i**** identifies intentional actors in organisations. Consequently, this step should not be too difficult: actors are tangible entities or groups of such entities. Actors can be individuals, such as the bank clerk 'John', classes (set of actors having more or less the same properties), the class of bank clerks for instance, or aggregates, such as the 'Bank Company'. At this early stage, we do not differentiate between agents, position or roles. Actors may hence assume several responsibilities; each represented by an 'incoming' dependency link.

Non-intentional Actors. (like existing information systems, sensors, software,...) Although ***i****-actors are characterised by intentions and motivations, we believe that non-intentional entities should be included in the model of the real world. Indeed, with the increasing introduction of technology in any area of life, the human being has become more and more dependent upon these technologies. It is hence not erroneous to talk

about dependencies upon technical systems. Furthermore, these non-intentional actors can also, by delegation of responsibilities, depend upon intentional and non-intentional actors.

Properties of Actors. Regardless of intentional or non-intentional actors, one can establish certain general properties:

- Actors are identifiable

This notion is represented in nearly every object identification methodology, such as Booch [Booch,86] or JSD [Jackson,83]. We mean by this, that every actor is capable of receiving a globally unique identifier. The concept of identifier is important because actors are not clearly separated by their properties (components and actions): actors can be different although they share the same properties. Besides, properties can change in the course of time which would imply that actors change in time.
This does, however, not necessarily mean that actors have to be a single component. Sets of people such as a developer team or the bank clerks of a financial institute can also receive an identifier.

- Actors have states

The current state do partially determine an actors' future behaviour.

- Actors have a behaviour

We understand by behaviour of an actor the fact that the actor suffers and/or performs a set of actions. This does not necessarily mean that his behaviour also has to be modelled in the *i** model.

- Actors have responsibilities and dependencies

Along with the concept of behaviour goes the concept of responsibility. It is perhaps the most important criteria for identifying actors in the real world. We mean by responsibility the internal or external motivations/intentions that are behind a behaviour[4]. In addition, responsibilities can be a means for regrouping explicitly modelled behaviours. There can be actors for which no internal responsibility exists; which have no idea why they have a certain behaviour. But they often have responsibilities towards another actor (although they might not know it). When a responsibility is external we talk about a dependency: the actor A_1 has a responsibility towards another actor A_2, who on the other hand becomes dependent upon the other actor A_1. We see here that both concepts 'actor', 'responsibility' and 'dependency' are strongly related.

Gathering actors. One cannot limit the analysis of the actors to those identified within the requirements document eventually elaborated by the client. Because this latter is often restricted to the actors that will interact with the future system, one has to analyse a broader range of documents.

Typical approaches for gathering information about actors are:

- interviews with personnel,
- analysis of organisational charts,

[4] One should however not conclude that the explicit modelling of the behaviour of an agent automatically leads to an explicit representation of the responsibility and vice versa.

- analysis of flow charts,
- analysis of existing processes,...

1.2.2 Dependencies in *i**

Dependencies are the other model components in the Strategic Dependency Model. In the previous paragraph, we considered already the strong relation between actors and dependencies. The identification of the actors and dependencies will in practice run much more in parallel than in the sequence showed here. Some dependencies might be defined even before the totality of the existing actors are identified.

We have to consider that there exists a multiplicity of dependencies between actors because of the following reasons:

- We did not specialise actors into agents, positions or roles.
- Actors can regroup societies or sets of smaller identical, with respect to their type, real world entities (classes).

Gathering Dependencies. A first step to find out dependencies might be to

- ask actors upon their responsibilities,
- analyse interactions between actors,
- analyse existing flow charts,...

The *i** chapter identified four types of dependencies:

- the goal dependency,
- the resource dependency,
- the task dependency and
- the softgoal dependency.

To differentiate between these different types of relationships and the 'direction' of the relationship, one should consider:

- *Who* is *responsible* if there are problems, *who* receives *ability* and *who* becomes *vulnerable* upon the dependency. These points should help in finding out which actor is the depender and which actor is the dependee of the relationship. The actor who is responsible is generally the depender and the dependee basically achieves ability but also becomes vulnerable upon the dependum. The notion of responsibility is also important because of the existence of delegation of tasks. If something (a task to perform, a goal to achieve or a resource to produce) is delegated to another actor, two situations are possible: (1) the delegating actor keeps the responsibility and (2) with the delegation of the task, goal, softgoal or resource the responsibility is partially or totally delegated. For instance, if a marketing manager who was previously responsible for the editing and the typing of an invitation letter, receives a typist to

assist him, the manager is still responsible for the editing of the letters, the typist however may also assume some responsibility for spelling mistakes or typing errors.

- Whether the depender gives instructions for achieving something. If this is the case, the probability is important that there exists a task dependency. We talk about task dependencies when the goal has to be achieved in a particular way. This differentiation is also important if we talk about computer systems. These latter are often supplied by the user with some 'input-data'. If this data has to be provided in a specific way as through a user interface or under a certain format, we will link the actors by a task dependency and not a resource dependency as one might first think.
- If the condition and its achievement can be clearly identified. Goals are defined as clear cut, black or white notions whereas softgoals are characterised by the difficulty of assessing the satisfaction of the condition to be attained.
- Whether the depender shows a particular interest in the process of meeting the dependency. This is another hint to differentiate between a task dependency or a goal/resource dependency. If he is interested, we often talk of task dependencies.

Documentation in *i.**

Although, we gave here some hints to model actors and dependencies, the choice of the actors and dependency will most often rely on the analyst. Furthermore, the list of questions to ask are by no means complete. It should be understood as a first impulse of possible reasoning. The differences between the different concepts might also sometimes seem unclear and left to interpretation. Therefore, for every actor, a precise description of the actor itself and its responsibilities should be added to the Strategic Dependency model. Furthermore, each dependency link should be accompanied by a description and an argumentation validating the chosen type of link.

1.2.3 The System Chain and the Model Boundary

At the beginning of this section, we stated that the ***i**** model cannot be restricted to the modelling of the interacting environment. The problem that arises then is to define boundaries. How far do we trace back the motivations? Jackson [Jackson et al,96] notes that 'almost every goal is a subgoal with some higher purpose' and compares engineering with religion since both are about goal satisfaction. He concludes that a subject matter, i.e. an area where the alternative goal-satisfaction can take place, must be defined. We agree with this concept but we note that this can seriously restrict the choice for alternatives. Consequently, we have to deal very carefully with the selection of actors. Especially when we are searching for a solution for a problem and assist the client in elaborating and selecting alternatives for his problem, a restrictive view of the problem domain might be too deterministic for the evaluation of alternatives. Sometimes, clients have already a specific idea about a solution when they charge a company with an information project although a better solution could also be explored.

In accordance with Jackson, we do not believe religious or psychological aspects are of concern here. Although sometimes useful for elaborating client's needs, it is hard to deal with them because they are difficult to elicit, document and analyse. In contrast with

these individual goals stand the business goals[5] which should be defined long before an information project is initiated. They should also appear in the Requirements Document inasmuch as the specification can be understood as the contract that exists between two parties.

Porter [Porter et al,85] uses the value chain as a means for analysing the role of information technology in enterprises. The value chain can be defined as a set of activities (value activities) that the organisation performs to do its business. Each activity increases the value of a product/service and the final value is measured by the amount the client is ready to pay. An organisation can create competitive advantage by reducing cost or by differentiating its products. Porter extends the notion of value chain that is internal to one organisation to a chain of activities that spans both the supplier and the customer side of the organisation: the value system. We will use this value chain as a boundary to identify actors and motivations within an organisation if the system is intend to support value activities within the organisation.

A generic model might be represented as in figure 23.

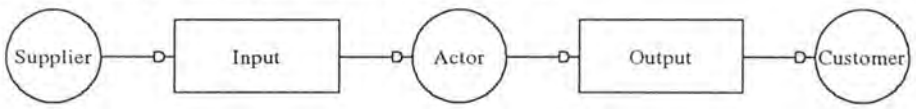


Figure 23: A 'generic' Dependency Model

In general, every organisational actor has somewhere a supplier and a customer side. Input and Output can both represent a resource or a service to provide.

Furthermore, information systems are more and more developed to support inter-organisational relationships such as Electronic Data Interchange (EDI) or Efficient Customer Response (ECR) and influence thus activities that span over several organisations. The motivations of indirect actors just as the directly interacting actors hence span different organisations. The Porter model will help us with the identification of potential actors of the future system that will support the system chain. The model can also help for the identification and evaluation of alternatives.

For a private company, one could use this value system and extend it with the shareholders and trade unions. The strategy, an organisation chooses for surviving, can be defined by making use of the four following competitive forces:

- The suppliers, which provide the organisation with input (products or services).
- The customers, that are provided with the organisation's output (products or services).
- The shareholders, represented by the board of directors, which are interested in the possibly greatest difference between the input value and the output value. They define also the policy of the company.

[5] These business goals should not be restricted to goals with respect to the achievement of profit. Business strategies should also be included.

- The trade union, representing the workers. Workers could also be put in the class of suppliers (of work) but because there exists a contractual difference between them, we treat them differently. The workers/employees are interested also in getting a part of the difference between the organisation input value and the organisation output value. Contrary to the shareholders, whose part of the input/output margin is variable, the part that goes to the workers is fixed.

A possible[6] model of a simple business company might look like the representation in figure 24:

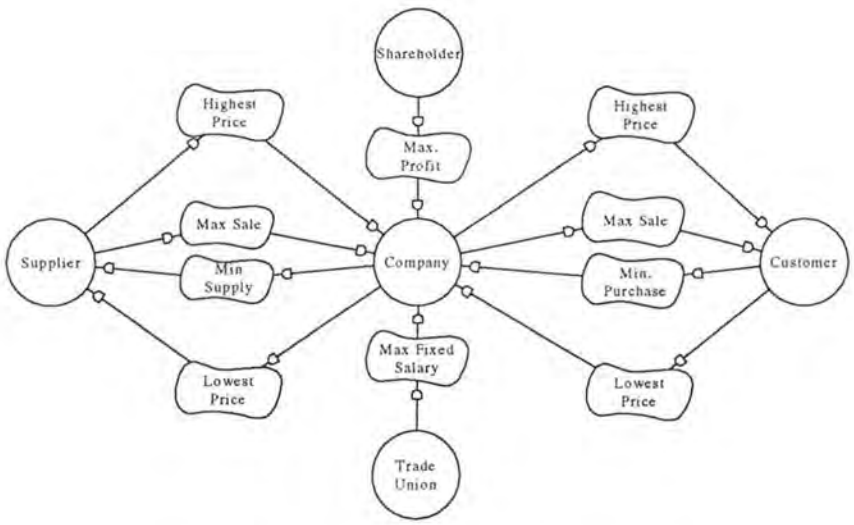


Figure 24: A Business Dependency Model

The need for this broad gathering of intentions and motivations is discussed below:

'Higher level' goals that would usually be represented as internal goals of actors are often 'substituted dependencies'. If we take, for example, a worker who performs a task inside a company. This task should contribute to a goal or softgoal 'High Profit' that is typically depicted as internal to the worker's context. The truth, although, is that this goal is in fact not internal. Both, the company and the clerk, have concluded a contract where it is stated that the clerk should perform the task in order to provide the company with profit. This fact should then rather be modelled by a dependency link between the 'Company' and the 'Clerk'. We can define the real actors that are related to a desire.

Organisational and strategic business goals are *the* objectives an information project has to respect. In comparison with business objectives, individual goals are often of minimal importance. If a specified functional requirement cannot be achieved according to some organisational and business goals, the future existence of the requirement can be thoughtfully questioned.

Consequently, we have to express goals that are internal to organisations. These combinations of these goals give the ultimate motivation for every business's objective.

[6] This model should not be regarded as a generic model for representing every business company. It is only intended to give an idea about the broad range of dependencies that might be considered.

The value chain identifying the internal actors define our *i** Model Boundary if the problem is inside an organisation and the value system identifies the actors if the problem is spanning different organisations.

1.2.4 'Specialisation' of Actors

The 'Actor' notion, that *i** offers for modelling organisational issues, is a very generic term. It can be used to refer to many kinds of real world entities: the bank clerk 'Mr Simpson', the bank clerks, a director of the bank, a software product, a financial company, etc. This 'Actor' concept can be divided into agents, positions and roles. The structuring of the different concepts is hierarchical: an agent is described as a set of agents, positions or roles; a position as a set of positions and roles and role is the basic unit for describing a behaviour. Furthermore agents, roles and positions can have instances and subparts. It should be noted that actors are intentional by themselves and the set of intentions of their subparts do not define their whole intentions.

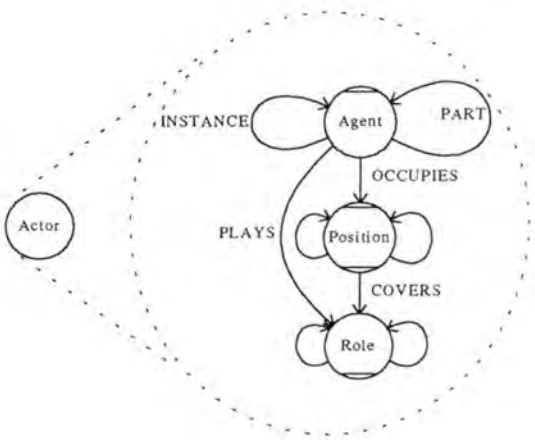


Figure 25: Actor Specialisation

The different relationships between these three concepts are:

- INSTANCE: an actor (Agent, Position or Role) can be an instance of another actor (Agent, Position or Role). This relationship is typically used for the representation of classes and its instances. The use of this relationship, however, will be restricted because classes in ALBERT refer to sets of entities having exactly the same properties which is not necessarily the case in *i** where instances of actors can assume responsibilities that are different from its classes dependencies. Thus, if we want to map an *i**-class to an ALBERT class, we have to be very careful that an instance of a class in *i** may specify more than its class.
- PART: an actor (Agent, Position or Role) can also be a part of another actor (Agent, Position or Role). ALBERT identifies the concept of society. By the means of the 'PART'-mechanism, we can define in *i** aggregates of different entities. Usually, we define in *i** these aggregates by agents or positions that are decomposed. This point like the previous point is only 'briefly' considered in chapter 2 and we will detail it a

little bit more because we think that it can be a useful mechanism for understanding how organisations are structured and influenced by Information Systems. An example of an organisational representation is given in Figure 26.

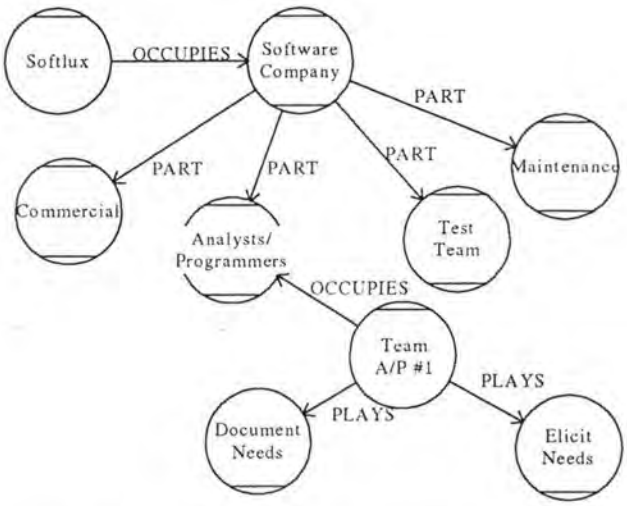


Figure 26: A possible Decomposition of a Company

In this example, a Software Company is divided into commercial, analysis/ programming, test and maintenance departments. One should note that with respect to the incompleteness property of *i**, one does not have to represent all the subparts of the company and that the intentions/behaviour of the company might not be completely described by its subparts. With respect to the same property, we can also stay at the 'Software Company' stage and leave the internal structure open. The model would hence be less expressive but more readable. Consequently, we can say that it always depends upon the analyst's need how far he wants to develop the organisational model into agents, positions and roles.

- OCCUPIES: an agent occupies zero or more positions.
- PLAYS: an agent plays zero or more roles.
- COVERS: a position covers zero or more roles.

Restructuring Dependencies. This remodelling of the Strategic Dependency Model will have an influence on the network of dependencies. Every dependum has to be rethought since the incoming dependencies have to be linked to one of the possible decompositions. In addition, new dependencies might be created between the 'decomposed' actors.

We connect dependencies with a *Role* node when these dependencies do not suppose a special social actor to play the role. The dependency could be accomplished by any actor and it does not presuppose special skills or experiences. This goes in opposition with the dependencies that will be connected to an *Agent* node. These dependencies rely on a special actor who should accomplish the dependency. His skills and experiences are of importance. *Positions* are usually decomposed into its typically assigned roles.

Roles typically assume only one or a set of more or less related incoming dependencies whereas actors, agents and positions can deal with more different types of incoming dependencies.

1.3 Identification of the ALBERT Model Boundary

The elaboration of the ALBERT agent begins with the identification of the actors in the *i** model that are potentially influenced by the introduction of the new system. This phase is strongly related with the identification of the problem itself. We cannot yet define precisely the problem because we do not have the necessary vocabulary and it is difficult to identify the actors that are influenced by the (until now not defined) problem. We assume here that already a vague notion of the problem and eventually an idea of a possible system (solution) exists. The *i** model has to be focused onto the potentially interacting actors and the decomposition of the actor node into agents, position or roles should be explored[7]. We continue this process by the following assertion:

> *An i* Actor, Agent or Position 'interacting with the system' corresponds to an ALBERT Agent/Society.*

1.3.1 Agents in ALBERT and *i**

Agents in *i** represent actors that have special characteristics and skills and can have several responsibilities towards other actors. The ALBERT concept 'Agent' restructures large specifications into finer pieces defining together the whole behaviour. ALBERT is elaborated for specifying composite systems. Furthermore, agents are characterised by their behaviour and the communication between agents. This behaviour may be due to several responsibilities.

The statement above is of importance. It has a huge influence on our modelling of the real world. One should first note that when we talk of a mapping between the *i** actors to ALBERT agents, we only consider actors that will interact with the future system. The *i** model will deal with a much greater variety of actors, agents, positions or roles. Consequently, the mapping is by no means complete. In addition, the mapping will always depend upon the analysts/developers and a general deduction rule is impossible to precise. Hence, the statement should not be understood as a rule for directly mapping *i** agents to ALBERT agents. It should rather be interpreted as an advice not to map ALBERT agents to *i** roles. *i** roles are then mapped to some statements inside of an agent (at least for the real world modelling).

We decided to bind the *i** actor, agent or position concept to the ALBERT agents when we talk about decomposed actors because of the following reasons:

[7] One should note that this last step is not absolutely necessary. The analyst can also proceed with the identification of ALBERT agents without having decomposed the *i** Actors. The ALBERT Agents can then be bound to the generic *i** Actors.

- Because we are dealing with the modelling of real world entities, we believe that the binding between ALBERT agents (or actors or positions) and *i** agents is more intuitive in opposition to the binding between ALBERT agents and *i** roles. ALBERT agents can assume several responsibilities and behaviours which seems to be much more related to the concepts of agent or position in *i**. Ideally, we say in *i** that one role responds to one responsibility; one type of dependency link. But positions or agents can cover or play several roles.
- Another, more practical reason is that when we will later define ALBERT agents' behaviours (see section 1.4), if we would model each ALBERT agent as an *i** role, we would have to introduce explicit communication to co-ordinate the two behaviours. This will lead to the introduction of statements that are of no importance for the understanding of the process and can lead to an over-specification; a property that is not desired in the Requirements Specification. In addition, another type of over-specification is the introduction of state components that are shared between the two ALBERT-agents. Because a state component can only be assigned to one agent, we would have to introduce supplementary statements that express a relation between the two different state components in the ALBERT specification which actually refer to *one* real world entity. The following example might help in understanding our viewpoint: assume that an Actor 'StockClerk' in our *i** model has a resource 'Stock' and his role is to ship sold items and update a mirror of the stock. This actor is then modelled by an *i**-Agent 'StockClerk' who has two resources 'Stock' and 'Mirror' and who now plays two roles: 'ShipItems' and 'UpdateMirror'. The first role corresponds to the behaviour of shipping a specified item and the second describes the updating of a mirror of the real stock to reduce the inconsistency between the reality and its mirror. Mapping the two roles to two ALBERT agents would yield the following specification:

SHIPITEMS

ACTION COMPOSITION

ShippingProcess(item) $\leftrightarrow$ *Remove(item)<> SignalRemoval(item)*

ACTION INFORMATION

$\mathcal{K}$*(SignalRemoval(_).UpdateStock: TRUE)*

UPDATEMIRROR

ACTION COMPOSITION

UpdateProcess(item) $\leftrightarrow$ *ShipItems.SignalRemoval(item)<>MinusMirror(Item)*

ACTION PERCEPTION

$\mathcal{K}$*(UpdateStock.SignalRemoval(_): TRUE)*

This communication and shared state components problem might be pretty misleading and the analyst might conclude that the real world consists of two actors. If we now consider that *i**-Agents or Positions are connected to ALBERT-Agents, a possible specification would be:

STOCKCLERK

ACTION COMPOSITION

ShippingProcess(item) $\leftrightarrow$ *Remove(item)<> MinusMirror(item)*

We immediately see that this representation leads to an increasing dependency towards the *i** model. We now need the *i** model to describe the roles that an agent plays. The ALBERT specification does not reflect any differentiation between the agent's roles. Inside the specification of an agent, statements or parts of statements have to be assigned to *i**-roles.

The reader should bear in mind that one can always refer to the node 'Actor' as a generic entity if one has doubts whether to model the node as an position, agent or role or if the specialisation of an actor is not important.

1.3.2 Agents and Societies in *i** and ALBERT

ALBERT has an explicit representation for modelling societies, aggregates that are composed of several different components. *i** does not deal with them explicitly (it has no explicit representation) but we might use the INSTANCE and PART mechanism that is connected to Agents and Positions in *i**. We have to pay attention, however, on the following precision regarding the meaning of an aggregate in the two models. *i** deals with intentions of an aggregate and ALBERT describes the behaviour of a society. Unlike ALBERT, where the behaviour of the society is completely described by the set of its components, an *i**-aggregate is taken to be intentional. It is not necessarily defined by its subparts with respect to its motivations.

We should also precise that when we define in ALBERT a class, all its corresponding instances have the same behaviour. This is not necessarily the case for *i**, where we can depict a special agent out of its class.

1.4 Identification of the behaviour

Until now, we only considered the actors, with their respective actor-decompositions, the dependencies that exist between them in the *i** model and the declaration of the ALBERT-Agents. The next step will lead to the complete description of the environment. This means, for *i**, that a 'complete' Strategic Rationale Model has to be elaborated and for ALBERT that the behaviour of the agents, representing the real world, has to be described.

1.4.1 Elaboration of the Strategic Rationale Model

The Strategic Dependency Model provided us with an initial understanding of the intentional relationships of the organisational environment. We will now try to get a deeper understanding of the processes and of the internal motivations that are behind the dependencies.

We will not simply elaborate a list of all the actions an entity performs as Jackson does. The Strategic Dependency Model already identifies the actors and their dependencies and responsibilities. As a result, a good starting point are the incoming and outgoing dependencies. Indeed, one cannot leave a dependency 'unanswered'. Someone depends (becomes able and at the same time becomes vulnerable) upon the actor for the achievement of a condition, the performance of a task or the deliverance of a resource. Theoretically, dependencies will be re-linked to some tasks, goals, softgoals or resources. If the dependency could not be linked to a task, goal, softgoal or resource, one has to pay attention if one is not in presence of an ability problem for the achievement of the dependency. The most common modelling step is however to connect the incoming dependency or the set of related dependencies to a task or a goal on the dependee's side. A task decomposition or a means-ends decomposition can then be developed in order to clarify furthermore the influences and impacts of the dependency. On the depender's side, one cannot say that there is an often reappearing 'configuration'. All the possibilities are imaginable.

Documentation in *i.**

As for the Strategic Dependency Model, one should also join to the Strategic Rationale Model a detailed description for every Actors, their possible decomposition, the dependencies between actors and internal to actors and the internal behaviour of actors.

1.4.2 Elaboration of the ALBERT Model

A first idea to model the behaviour of ALBERT-agents might be to start by the dependency links and the internal behaviour represented in the previous Strategic Rationale Model and then to describe the details in the ALBERT part. We do not automatically adopt the first approach for the following reasons:

- in the Strategic Rationale Model only strategic important relationships are represented and
- the internal (to actors) representation of behaviour is by no means complete and the representation in the Strategic Rationale Model can not conclude to a corresponding representation in the ALBERT Model. For instance, a resource does not necessarily lead to a state component in ALBERT.

Inasmuch as we cannot generalise the assertion that all dependencies induce a communication and vice versa, which could be represented in ALBERT as information/perception constraints, we will propose a different approach which first describes the internal behaviour of the agents and then only analyses the communication

constraints between the agents. The relation between the *i** concepts and ALBERT statements is only examined afterwards.

A imaginable 'meta-algorithm' or process for elaborating the ALBERT document might be:

- Define State Components and Data Types
  - Define Initial Valuation and Derived State Components
  - Define State Behaviour
- Define actions
  - Define Effect of Actions on State Components
- Define co-ordination
  - Internal: Action Composition, Action Duration, Precondition, Triggerings, (State Behaviour)
  - External: Action Perception/Information

State Components.

A possible life of an agent in ALBERT is described by a sequence of states. These states are structured by state components. They represent the internal knowledge of the agent about his environment. The chapter about ALBERT described already the possible types of state components. We will not reintroduce the different possibilities here but we want to say few words about some pragmatic aspect of state components .

State components, just as attributes for other object-oriented approaches, reflect a property, a feature an agent in the real world has. The most common state components for people are (for example): name, date of birth, eye colour,... Although the creation and the destruction of real world entities' attributes are hard to define, ALBERT state components are restricted to the lifetime of an agent. Their creation and destruction takes place with the creation and destruction of their respective agents' life. In general, a real world entity has an infinite set of properties or features but we have to select the features that are important for our understanding of the entities of the real world. If we take, for example, a person that gets married. He/She always had the state component 'Date-Of-Marriage' although he/she only thinks of it now that he/she starts playing the new role of husband/wife. If we are now only interested in the person as a pupil we can leave out this state component. A priest, however, who wants to keep a data base of the couples married by him, might be more interested in such a state component.

With respect to the last consideration about the importance of the domain, we can also say that the distinction whether a real world entity is an agent by itself or an attribute of an agent relies upon the part of the real world we want to model. A engine can be an attribute of a car for an insurance company and at the same time be an object/agent for the constructor of the car.

This difficulty of identifying state components and agents leads to the following approach:

1. Identify agents and do not yet bother about agents and state components. At first, every real world entity can be modelled as an agent.
2. Reject agents which are not necessary for our purpose, are in fact attributes/state components or are redundant with other agents.

3. Select and regroup state components.

The discussion above indicated the difficulty for giving hints for eliciting and modelling state components. One can refer to the same kind of documents to achieve knowledge about the state components that are important for the knowledge of the problem domain. Every noun that appears in these documents might be a possible agent or a state component of an agent. We will restrict ourselves by saying that, whatever state components one chooses, it is always helpful to add a detailed description, which can also be informal, to the declaration of the state components. This is helpful because during the modelling stage we necessarily choose words to represent the real world entities. These words may, however, have a different meaning for different people. It might be very useful to give an informal definition besides the usual declaration. This is also true for the declaration of data types.

Usually, state components can be constrained. The 'templates' ALBERT offers, can help a lot for eliciting constraints about the state components:

- The initial valuation of a state component, which defines the value of the state component at the first state of a possible life.
- The derived state components. These are 'artificially' created state components in order to improve the readability and the use of the specifications.
- The behaviour of states for the whole life-cycle.
- The relation of state components with actions: preconditions (necessary condition for the occurrence of an action) and triggerings (obligation for an action to occur)

Actions.

Since state components cannot change by themselves, ALBERT introduces actions to change these state components. In opposition to JSD [Jackson, 83], actions are not necessarily instantaneous. They can last over a period of time and are decomposable. What is common between the action notion in JSD and action notion in ALBERT is that actions are always performed by an agent and that the change of the state components by an action is instantaneous (event). We see meanwhile that the ALBERT solution is nearer to the real world concept of an action because it has a duration and it can be decomposed.

Besides the alteration of state components, actions have also another role in an ALBERT specification: the co-ordination of internal and external processes. This is partly indicated in the action composition template.

Just like for the state components, the ALBERT headings generate a set of possible constraints about actions:

- As already said the triggerings and the preconditions.
- The Action Composition establishes relationships between the occurrences of actions.
- The Action Duration template defines the exact duration, an upper limit or a lower limit.

- And finally the Effects of Action template that describes the modification of state components by actions.

Communication.

Agents can also communicate with each other agents by the means of state information/perception and action information/perception mechanisms. An agent can learn about the value of another actor's state component or the occurrence an external action and tell other actors about the occurrence of an internal action or the value of a state component.

1.5 Definition of the Objective

Until now, we have described an abstract model of the real world. This model is characterised by a problem; a situation that is not desired as it is at this moment. A problem might be a lack of control on a process existing in the real world or a new requirement needed by the real world. We now have to introduce goals or requirements that have to be met by the future system.

In the *i** model, we introduce a new role that assumes new supplementary objectives. This role is often based upon an existing role with additional conditions that consider the objective.

In ALBERT, we use the vocabulary introduced by the modelling of the real world entities to add constraints on top of the whole specification that will ensure that the problem will be resolved. These constraints are not assigned to a specific ALBERT-agent and we cannot speak of a 'real' ALBERT specification. These constraints usually consist in controlling the occurrence of actions, the state of agents or in the introduction of new processes.

These are typical functional requirements: the processes that exist in the real world have to be corrected in order to exclude the problem or new processes have to be added.

2. System Requirements

The previous step identified:

1. A problem domain or problem environment as some part of the real world. The problem domain was described by the ALBERT document for clarifying the behaviour of the different agents and by the *i** model to show the rationales that are behind this behaviour. This problem domain is however unsatisfactory (lack of information or lack of control) for the client.
2. A set of objectives/requirements which should be respected by a new environment, i.e. the solution environment.

These requirements are now assigned to a system that assumes the responsibility for 'solving the problem'. The system, in order to achieve the requirements, has to bring about some output which is additional to the behaviour of the environment. Output can be information necessary for the environment or control upon the environment. In addition, the system also has to be connected to the real world: this connection provides the system with the necessary input.

We basically deal in this step with functional requirements and the role played by *i** may seem lessened but we consider that the *i** model can support us in searching for functional alternatives and specifying the qualitative requirements.

2.1 The Omni-System

The Omni-System[8] is characterised by two properties which give it an overall power: the omniscience property and the omnipotence property.

- Omniscience. By the omniscience property, we mean the capability of the system to be aware of all the important information about the environment. Important information is information which should have an impact on the behaviour on the system and thus information that should influence the behaviour of the yet still unsatisfactory environment. This information can be the occurrence of a particular behaviour of an environmental agent or a particular state in the environment, for instance. If the system cannot find the information in the modelled environment, two alternatives are possible: (1) the information has be deduced/derived from the existing information or (2) we forgot to model some important part about the domain and we will have to go back to the first step.
- Omnipotence. The omnipotence property gives the system the power to have a direct influence on the behaviour of the problem environment. This influence is generally executed by the means of actions or information towards the environment.

Omni-System: enlarged, interacting and reduced.

The *enlarged* Omni-System is characterised by the actor 'Omni-System', the modified actors of the problem environment and their incoming and outgoing dependencies. The *interacting* Omni-System is defined by the intersection of the Problem Environment and the Enlarged Omni-System: the changed actors that still belong to the environment. We have to consider this interacting Omni-System because it is impossible to clearly separate the Problem Environment and the System. The *reduced* Omni-System is restricted to the actor 'Omni-System'; the exclusion of the Interacting Omni-System from the enlarged Omni-System.

[8] For this second step, we use the notions 'Omni-System' and 'System' indistinguishably.

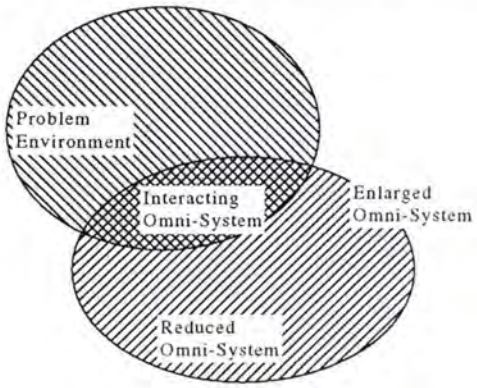


Figure 27: The Omni-System

We introduce this Omni-System to distinguish between the indicative and optative mood as Jackson and Zave [Jackson et al,96] characterise the difference between the statements describing the environment without the machine and the statements describing the desired environment. The introduction 'resolves' two problems:

- The implementation bias, which is the consequence of the use of states in specification languages. The internal behaviour of a system is often specified in terms of its internal states which leads to the implementation bias. Because the system is now characterised by the omniscience property, it does not have to maintain internal states about the environment.
- The necessary distinction between the indicative mood and the optative mood. Unlike the A-7 method [Jackson et al,96], ALBERT statements do not clearly distinct between the constraints identifying the system and the constraints identifying the problem environment. Consequently, we make the distinction by the expressive introduction of the Omni-System as a separated actor and its dependencies.

This introduction of the system, leads to the specification of:

- states/actions shared by the environment with the system
- conditions under which the system is altered
- states/actions controlled by the system and communicated to the environment
- restrictions under which the behaviour of the environment is altered.

Omni-System: An ALBERT-Agent?

For the *i** model, the situation is clear: we introduce a special actor 'System' that assumes the new responsibilities identified by the new objectives. Each objective imposed by the environment will lead to a dependency link between an environmental actor and the new system.

For the ALBERT specification, however, the situation is less obvious. Two situations are imaginable:

(alt 1) Explicit representation. We introduce a special agent (or society) in the ALBERT specification which is a correspondent to the *i** actor 'System'. In ALBERT, we identify then the problem environment agents and the 'System' agent which together compose the solution environment. The methodology would still be appropriate separately. Some agents describe the problem environment and some the system. This solution seems to be more structured and intuitive.

(alt 2) Implicit representation. We introduce no explicit system agent in ALBERT. The solution environment is then identified by the changed problem environment. It is the *i** model's role to differentiate between indicative and optative properties. The *i** actor 'System' would identify the constraints which had been added to or changed in the ancient problem environment. This alternative increases the dependency between the two models and especially the dependency of the ALBERT specification on the *i** model. Furthermore, the classical concept of an ALBERT agent would have to be revised. In the previous step, we considered ALBERT-agents as concrete resources having responsibilities for actions happening in the environment and communicating with other agents. Contrary to the preceding alternative, the agents would describe the problem environment and, at the same time, the system.

Decomposition.

One should however not conclude that the Omni-System has to be one single actor in *i** and one agent in ALBERT. It can also be an aggregate of different actors where each actor would be responsible for a smaller part of the overall function to provide. This is very useful for important functional requirements. This composition should although not be regarded as an obligatory implementation composition (which actually can be completely different). Furthermore, the constraints belonging to the enlarged Omni-System can include actors that belong to the problem environment.

For systems that offer a larger range of functional services, we adopt a 'decomposition' approach. First, we identify one system that meets all the imposed requirements. This system is then divided into subsystem, where each assumes an identified responsibility.

i-ALBERT Mapping.*

In accordance with the two alternatives, we presented previously, the mapping between the *i** entities and the ALBERT entities changes for this stage.

Firstly, if we represent the system explicitly in ALBERT, this system is a theoretical or 'virtual' system which has yet no counterpart in the real world. Consequently, the important argument that the mapping between the two models and the real world should be intuitive and natural does not exist for the system. We believe that it might be useful to relate every *i**-role to one ALBERT-agent. As a result, every ALBERT-agent reflects the behaviour of one role.

When our Omni-System is defined as a set of agents, we necessarily have to deal with the co-ordination between these agents. This can be done by constraints on top of the agents. These constraints have to be clearly stated and will be identified in the *i** model.

Secondly, if the system only represents the 'changed' environment, *i** actors, agents, positions or roles would identify some constraints in the environment.

2.2 Specifying the functionalities

In the following, we describe a possible approach for the elaboration of the system:

- we introduce the system,
- we identify the means by which the system has an impact upon the environment,
- we identify the necessary resources of the environment,
- we specify the relation between the outputs and the inputs.

This approach seems to be more adequate for the first alternative (the explicit representation) we gave. However this approach can also be used for the second alternative (the implicit representation) although we will not introduce the system.

2.2.1 Identification of the Omni-System

We connect the system to the environment and the system controls or adds new functionality to the environment. These functionalities are now introduced in the *i** and the ALBERT model.

In *i**, this leads to the creation of the actor system.

In ALBERT, we create an agent (or society) who assumes the responsibility for solving the problem. For the second alternative (alt 2), nothing has to be done since we do not represent the System explicitly.

2.2.2 Identification of the System-Output

Next, we introduce the means by which the Omni-System can add some functionality to the environment: the System-Output. This influence on the environment's behaviour can be basically achieved through the controlling actions or the supply of some information.

Identification of the system-output in *i.**

This is usually modelled in *i** by transforming the internal goals, resources or tasks to some dependencies, where the depender is usually an actor of the environment and the dependee is usually the system itself or a role played by the system.

We distinguish in the *i** part especially between task and resource dependencies on the Omni-System.

By a *task dependency* the depender wants the system to perform a special task. We use task dependencies for specially constraining dependencies and to indicate a perceivable shift of responsibility towards the system. The decision making is concentrated on the reduced Omni-System.

A *resource dependency* is not so constraining. The depender gains ability to use the resource but it is left upon his appreciation what he does with the resource. Therefore, if

the requirement is a control upon an existing process, we cannot say whether the objective is completely achieved by introducing the system: it may be that some organisational restructuring is needed (see our example in chapter 5, section 2.3.2, where the Efficient Order Processor receives a new precondition constraint). The decision making process (structure) on the reduced Omni-System's side is also not as big as for the task dependency.

Identification of the system-output in ALBERT.

The control of an existing process in the environment or the introduction of an additional functionality in ALBERT is basically done by the introduction of co-operation constraints in the system and environment specification. The system has first to be connected to the environment before changing the behaviour of the environment. It might be that special actions, state components or relating constraints have to be introduced in the environment because the environment was not designed to react on the system.

2.2.3 Identification of the System-Input

The introduction of the Omni-System will finally transform the problem environment into the solution environment. The identified problem will no longer exist and the environment will be satisfactory (with respect to that problem). But to be able to influence the environment, the system has to be informed about the environment. We have to identify this information.

Identification of the system-input in *i.**

Typically, we connect the system with the environment by some dependency links where the system is the depender of the relationship. These latter express the necessary resources, tasks or goals for the decision making process that takes place in the system. Softgoals usually express properties that have to be achieved by the environment.

Identification of the environment-media in ALBERT.

In ALBERT, we identify the state components and action occurrences in the environment that are necessary for the system. These components and actions are then made visible for the system.

2.2.4 Identification of the System's behaviour

Having now well defined the information about the environment and the control upon the environment, we can continue with describing the relation between the two components: the system's input and the system's output. This relation defines the behaviour of the enlarged Omni-System. We consider the enlarged Omni-System because we describe properties that hold thanks to the system.

Identification of the behaviour in *i.**

In *i**, we generally search for several roles assuming each a different kind of responsibility, i.e. a set of related incoming dependency links. These roles are then characterised by a behaviour. We model this typically by a task although the other concepts of *i** are also possible. When we modelled the behaviour by a task, we can continue by decomposing the task into tasks, goals, resources and softgoals. A goal lets

us choose between different alternatives for achieving the condition stated by the incoming dependencies.

Identification of the behaviour in ALBERT.

In ALBERT, we use the different headers for specifying the system's behaviour. To each role we assign an agent. This leads to a better identification of the role played by each agent but we might also have to express specific constraints about the co-ordination of the different agents. These constraints can be expressed on top of the different agents belonging to the system and identified by the *i** model.

2.3 Evaluation of the Alternatives

The problem that arose in the real world does not necessarily lead to one solution. Many functional solutions are conceivable. Although, we introduce here the first evaluation, we believe however that this evaluation, together with the elicitation of softgoals, is from now on a process that will continue during the whole elaboration of the specification. Especially with the 'implementation' of a concrete system-solution, new softgoals might appear and older ones might be refined.

For a first evaluation, there are two possibilities:

- If possible, we make an evaluation of the different alternatives and select one to continue with the next step. This evaluation is based upon general requirements which are not specific to one solution. Hence a first evaluation can be made from the softgoals that have to be respected by every sub-element of the decomposed task.
- We continue the process with a number of solutions and we evaluate the different functional alternatives after the solution has been designed. Indeed, the functionalities identified in this step are quite abstract and we might wait for a more concrete solution or implementation to have a better understanding of the alternatives and of the goals and softgoals they have to accomplish.

When we have defined the qualitative requirements, that the system has to accomplish, we have to get back to this step for the evaluation of these requirements and the selection from their possible alternatives. Furthermore, the evaluation might generate a better understanding about the requirements. Consequently the task of evaluating and identifying requirements and alternatives are strongly intertwined.

2.4 Identifying Softgoals Dependencies

We have described the functionality the system has to produce! That is already a great step towards the satisfaction of the client. But there is another factor that is, since the 80's-90's, becoming increasingly important: the non-functional requirements. This progress in importance is so dramatic that often the acceptance of a project depends upon these non-functional requirements. As a result, if a requirements engineering method wants to be successful, it has to include such constraints.

The difference between functional and non-functional requirements is generally hard to identify (see for instance the discussion in chapter 1) and often depends upon peoples' point of view. To give an intuitive understanding of non-functional requirements, we list some typical groups of non-functional requirements (properties that have to be respected by the functionality):

- Performance
- Cost Constraints
- Maintainability
- Accuracy
- Backup/Recovery
- Security
- Reliability

Besides the difference between these functional and non-functional requirements, another criteria is also used to make the difference between the type of requirements: quantitative and qualitative goals. In this perspective functional requirements refer to quantitative obligations and non-functional requirements to qualitative properties of the system. We believe that this is a more fruitful notion because they are related to the concept of goals and softgoals:

- goals which can be clearly defined,
- softgoals that are a priori not defined.

This distinction does not consider the type of language used for expressing the goals. Consequently, a condition can also be clearly stated in a non-formal language.

2.5 Refinement of Softgoals

With respect to the discussion above, we believe that, at first, requirements like accuracy or time constraints are modelled by softgoals. But these vague notions have to be elaborated and specified more precisely since we deal with a document that plays the role of a contract between two parties. When dealing with this precision, we identify two operations:

- the refinement of the softgoals into more elementary goals. These often turn out to be partially or totally composed of (hard-)goals. When the refinement finishes with a set of purely hardgoals the main softgoal can be transformed into a (hard-) goal that is definable.
- the 'operationalisation' of the goals. The hard goals can be partially or completely introduced as constraints in the ALBERT model.

Among these goals and softgoals, one can also identify goals/softgoals that are related to the general settings of the system and goals/softgoals that relate to the software:

- On the one hand, the first ones are meant for the requirements analyst who has to develop a system. They should be fulfilled by every possible system-solution and hence are identified for every system-solution. There are also goals/softgoals that are specific to one solution. These are usually developed during the elaboration of the system's 'implementation'.

- On the other hand, there are also goals/softgoals that are meant for the designer of the software. These are usually acquired during and after the implementation step and when the concrete system agents (hence also the software agents) are identified. They can also be derived from the user's requirements. Like for the previous group of goals, one can also talk about hard-goals and softgoals. For example, 'Use a specified ODBC driver' would be a hardgoal whether 'Use an intuitive User Interface' would be a softgoal.

A possible approach for refining the softgoals might be:

- Describe the softgoals in detail using an informal or a formal language. If their definition is clear-cut, they will have be transformed into goals. If they can be introduced in the ALBERT model they become delivrable/operational goals. Otherwise, they stay at the qualitative level and will be reduced furthermore.
- Refine the softgoals and goals into their components. Softgoals have to be explored to reduce the complexity of their meaning and to detail their impact on the specification. Goals that are yet not detailed enough to be introduced in the ALBERT specification have to be refined and operationalised.
- Introduce the deliverable goals in the ALBERT specification. The other goals and softgoals are kept for latter stages.

We can introduce the NFR framework proposed by L. Chung [Chung et al,94] that introduces also good 'recipes' for decomposing softgoals as accuracy, security, etc.

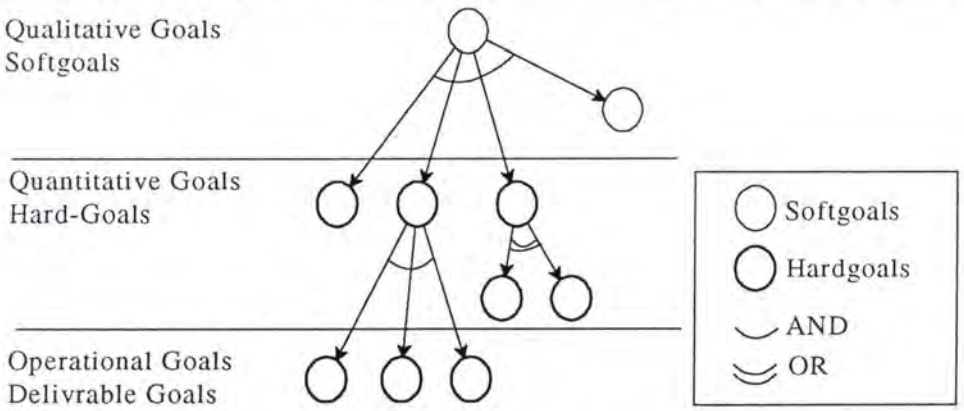


Figure 28: Refinement of Softgoals using the NFR Framework

3. System Specification

The system, we have specified, should now have all the necessary functional and non-functional requirements in order to eliminate the problem. But, there are two major problems with the present situation:

- The model of the system, we described, most often only works in theory. We have tried to solve a problem that occurred in the real world. Consequently, we made a model abstracting from all the irrelevant details of the real world. Now, we have to do the inverse step: take the abstract model of the system (our Omni-System) and map the (problem solving) functionality into concrete actors/agents. We consider this greatly as the 'implementation' phase of the requirements engineering process. The model has to be 'implemented' into the real world model. This leads usually for information system projects to the introduction of hardware and software agents. We can note that also purely human solutions can be considered. We will see this in the case study in chapter 5.
- The softgoals we identified in the previous step have to be revised so that they can be introduced in the requirements specification and assigned to several agents.

As already said, this last step can be regarded as the 'implementation' step. By this, we do not understand the design and coding of software but the elicitation and analysis of the system components. Indeed, our methodology does not merely limit its scope on the requirements of software. Problems arise in the real world including people, machines and software and the proposed solution has to interact with this problem environment. Consequently, we have to introduce the concrete media by which the communication between the system and the environment takes place.

In the following, we will describe a typical approach for designing the system. The system is altered from an abstract description of some functionalities into the specification of concrete agents.

3.1 System and Subsystems

In most cases, systems provide complex functionalities so that these latter are regrouped into subsystems. These subsystems can be implemented independently of other parts of the system and are characterised by common properties. Usually, these common properties are common responsibilities, functionalities, services or behaviours. For the moment, we do not consider concrete real world aspects. This will be the purpose of a latter stage.

In the 'System Requirements' step, we already introduced the notion of subsystems playing each different roles. This decomposition can be used but does not necessarily have to be coincident with the decomposition we elaborate here.

In *i**, we identify different positions and/or roles that have certain responsibilities. The positions and roles also have dependencies upon other position or roles.

In ALBERT, we specify agents or societies upon the same criteria. Each actor has a responsibility which leads to the specification of a behaviour. Furthermore, the agents are communicating by the means of the perception/information mechanism.

We can identify several types of subsystems which are derived from the retraction of the omniscience and omnipotence assumption and the functionalities described in the previous step. This natural division can be used to organise the subsystems into:

- an Application level derived from the functionalities.
- an Interface level derived from the retraction of the omnipotence and omniscience assumptions. Indeed, special actors and agents might be responsible for getting input information from the environment and others might be responsible for influencing the behaviour of the environment.
- a Mirror Level where the real world entities are mirrored and simulated. In general, an information system deals about entities of the real world. Hence, information has to be maintained. In a library system, for instance, books are getting borrowed and returned.

3.2 Subsystems and Processors

The agents and actors, representing the system's responsibilities, are now assigned to concrete agents because we want to co-operate with the real world. We refer to processors when we talk about the concrete entities that provide functional services like terminals, user interfaces, peripherals, CPUs, other hardware components and software components.

The processors are due to:

- the requirements expressed by the customers,
- the proposition of system components by the system developer.

The system has to communicate with the real world entities and we will retract the omniscience and omnipotence assumption. This retraction leads to the introduction of special agents (the processors) and to special statements refining the visibility and information constraints. If the processors are derived from the customers' requirements the system developer has to respect these constraints. If the customer has no predefined needs about the implementation the system developer has more freedom in choosing the necessary processors. He has, however, to respect the constraints (softgoals) expressed by the customer. Requirements like performance needs or cost issues may restraint the set of possible hardware and software components. In addition, they have to be included into their respective specification.

In *i**, the positions and roles are assigned to previously identified agents or new ones. Each agent playing a set of roles. For instance, the 'input-agent', responsible for processing the input of an information system, and the 'output-agent', responsible for processing the output to the environment, are now regrouped to one agent 'Terminal' who plays the two roles.

In ALBERT, several agents describing different roles or responsibilities may be regrouped into one agent or one agent may be split over several concrete agents.

Furthermore, we have to rethink the coordination of multiple agents, the communication between these agents and shared/unshared resources:

- Processes were spanned over several agents (roles) or were restraint to one agent. These processes have to be reconsidered because they are split over several agents or merged into one agent.
- The coordination of agents uses necessarily the information/perception mechanism. Consequently, these constraints also have to be analysed and updated.
- Resources that were shared between several agents also have to be assigned to one agent. Hence new constraints might also appear due to this need.

We deal here as in the first step, where we talk about the problem domain, with concrete entities of the solution environment. Consequently, the mapping between the real world entities, the *i** actors, agents, position or roles and ALBERT-agents should be intuitive. An ALBERT agent can be mapped to one *i**-agent (actor or position) and the *i** roles identify some constraints describing that role within several agents' 'template'.

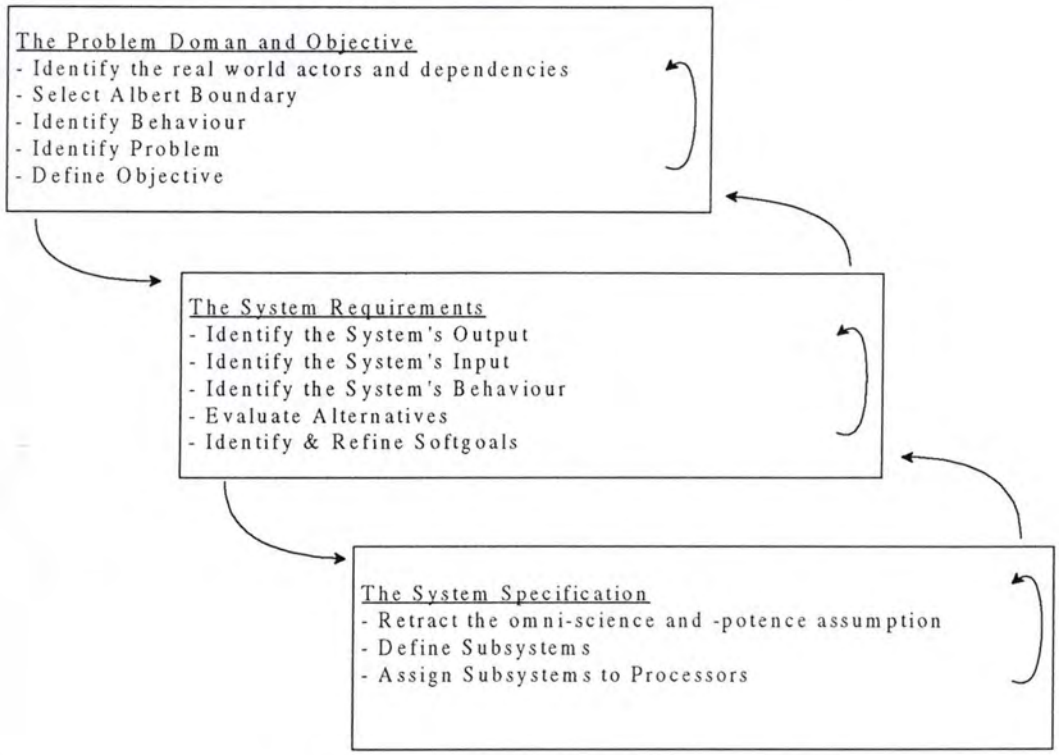
Conceptual Modelling

The components also have to maintain information about the environment since they have no direct access to this information. Consequently, the system simulates some part of the environment. We use the problem domain as a reference model which will guide us through the elaboration of the conceptual model.

The system developer will also have to decide under which form these conceptual models are maintained. Performance and cost constraints can have also a serious influence upon this decision. Upon these criteria he has to decide between a file storage system or a relational database system, for instance.

4. Summary of the Method

To summarise this chapter, we can draw the following graph which sums up the different tasks the system analyst/developer has to perform. These tasks are given in certain sequence but we should however note that a lot of these tasks are executed in parallel (the identification of the System's Input, Output and Behaviour for instance). In addition, some tasks are fated to backtrack. In this way, the evaluation of alternatives often has to be revised after a concrete solution has been proposed.


The Problem Doman and Objective
- Identify the real world actors and dependencies
- Select Albert Boundary
- Identify Behaviour
- Identify Problem
- Define Objective
The System Requirements
- Identify the System's Output
- Identify the System's Input
- Identify the System's Behaviour
- Evaluate Alternatives
- Identify & Refine Softgoals
The System Specification
- Retract the omni-science and -potence assumption
- Define Subsystems
- Assign Subsystems to Processors

CHAPTER 5
CASE STUDY: MAIL ORDER EXAMPLE

The previous chapter explained theoretically the basic steps of the approach, we choose for elaborating a requirements specification. This chapter now clarifies the different concepts by illustrating them on a case study: the mail order example which derived from a discussion with E.Yu.

The case study deals with the problem which could arise in every newly created mail order company. To simplify the model, only four major actors are considered: Customer, Office Clerk, Stock Clerk and Bank Clerk.

In the example, the Office Clerk processes orders submitted by a **Customer**. We assume that all transactions are paid for using credit cards. Thereby, in addition to the ordering of items to be purchased, a customer also submits its credit card information such that the Office Clerk could process the payment of the purchased items.

After having analysed and verified that the order is error free, the **Office Clerk** proceeds to process the payment. A debit request is submitted to the bank clerk for receiving the agreement from the bank clerk. The Office Clerk waits for the response from the bank clerk, which indicates either a confirmation or rejection of the debit requested, before making the invoice and ordering the transfer. Next, he transmits the invoice to the **Stock Clerk**, who has to ship the respective item. If the order is rejected, a corresponding procedure would be invoked (although we will not model it).

The **Bank Clerk**, upon receiving the debit request by the Office Clerk, is expected to present a response of confirmation or rejection for each transaction to the Office Clerk. When the transfer is ordered he has to perform the required transaction. The interaction between the office clerk and the bank clerk is more complicated and requires more time because the bank clerk belongs to another organisation.

This interaction between the bank clerk and the office clerk occurs for each order. But sometimes it turns out that orders are not deliverable because of insufficient stock levels. This costly interaction should thus be avoided in such situations.

As it results from the problem statement, the system to be installed will have to avoid this costly office clerk - bank clerk interaction when the stock is insufficient for handling the shipment.

1. Problem Domain Model and Objective

The goal of this section is to describe the problem domain, i.e. the interesting part of the real world, to identify the problems which lead to the conclusion that the real world is unsatisfactory and to identify objectives/requirements that have to be met by the future system so that the environment will become satisfactory.

At this stage, we clearly differentiate between the goal of the *i** model and the goal of the ALBERT specification.

- On the one hand, we are interested in a broader modelling of organisational issues (*i** purpose).
- On the other hand we concentrate on the communication with the future system and we think about the place and time where something has to be changed (ALBERT purpose).

1.1 Identification of the real world in *i**

Because we cannot seriously expect that the identification of the actors and dependencies of the real world entities of interest will be developed in one step, we will describe the exploration in two sub-steps. We choose a top down approach: from the identification of a bigger aggregate to the decomposition into several smaller entities. In addition, the identification of actors and dependencies is described in parallel and not in the exact sequence as described in the previous chapter.

1.1.1 Identification of the Actors and Dependencies (1st iteration)

For our example, we first identify the three main actors: the shareholders, the customer and the Mail Order Company itself (see Figure 29). We also identify other actors such as: the supplier for items for instance. We will however mark them as out of boundary.

The *Customer* depends upon the Mail Order Company to receive the desired item. This is expressed by the resource dependency *'Item'*. The Customer receives ability to use the item and becomes at the same time vulnerable upon the 'Mail Order Company' if the latter would run out of stock for instance.

Furthermore, he wishes that the incoming orders are processed efficiently in order to receive the item in a short delay. This is represented by a softgoal dependency *'Efficient[Processing]'*. At this early stage, we assume that the customer has not made up his mind about what he means under an 'efficient processing'.

The *'Confidential[Processing]'* softgoal dependency expresses the desire that the orders are processed confidentially with respect to privacy.

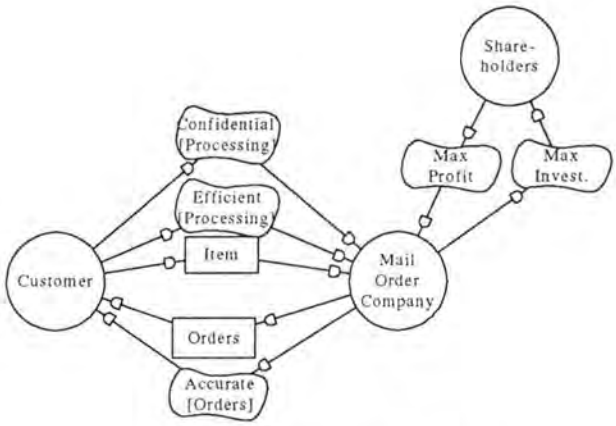


Figure 29: A first SD Model

The *Mail Order Company* depends on receiving orders from the clients because it is a private company and its profit can be defined by the margin between the price of the ordered items and the final cost of production for the items. This is indicated by the *'Orders'* resource dependency. The orders are the necessary input for the main activity of the Mail Order Company.

In order to stay more or less efficient and cost effective, the Mail Order Company also depends upon the fact that the orders are correctly filled. We represent this fact by the softgoal dependency *'Accurate[Orders]'*. Although its achievement can be clearly determined (either the order is correctly filled in and the item listed in the order is sold by the company or the order is false, i.e. the item is not sold by the company) its achievement also depends upon aspects that are more hard to define such as a clear handwriting for example. We can also note that this is a typical committed dependency because a control on the correctness has always to be done and the impact of false orders is considerable but not fatale for the company.

We identify, furthermore, the actor *'Shareholders'*. We choose this actor inasmuch as it is him who defines the most important objective the company has to accomplish: yield a maximum of profit. We model this dependency by a softgoal *'Max Profit'* because it is hard to define whether the condition is achieved or not. Although, we represent this dependency by a softgoal, it is one of the hardest and not easily debatable 'goals' an information project has to achieve. As our methodology is also thought to assist the client in the elaboration of early requirements and help in the search and selection of alternatives, we think that these goals are of importance as well. Every value activity can be sooner or later related to such an objective. These kinds of objectives will also be the ultimate boundary for the exploratory search for intentions and motivations. Everything that goes beyond these objectives would deal with psychological or religious aspects.

In return, the Mail Order Company depends upon the shareholders to invest large amount of money in the company.

1.1.2 Identification of the Actors and Dependencies (2nd iteration)

We now go on by decomposing the Mail Order Company. On the one hand, the customer is actually not interested in the individual who processes the orders. On the other hand, with the large amount of activities companies have to deal with, activities are often divided into sub-activities and these sub-tasks are then delegated to an agent or class of agents responsible for accomplishing the task. The Mail Order Company can be interpreted as an aggregate of different entities working together to accomplish a common goal. The discussion below refers to figure 30.

First, we identify the *'Office Clerk'*. It is under his responsibility to process the orders so that the resource dependency *'Item'* is accomplished. His way of processing will also have large influences on the softgoal *'Efficient Processing'* although we will see that it is not enough that only he fulfils the softgoal. The analysis of the dependencies, that the Office Clerk has, reveals two more actors: the Stock Clerk and the Bank Clerk.

The role of the *'Stock Clerk'* is to put the incoming items into the stock and to ship the sold items to the customers. We see here a typical case of task delegation: the Office Clerk could also carry out this task by himself but in order to rationalise the value activities the task has been delegated to the stock clerk who now assumes all responsibilities for the stock. Consequently, the Office Clerk becomes dependent upon the behaviour of the Stock Clerk and we model this by the goal dependency *'Ship [Item]'*. The Stock Clerk is still free to choose the adequate way for shipping the items.

A similar actor is the *'Bank Clerk'*, who belongs actually to another organism (we will model this latter). The Office Clerk depends upon the Bank Clerk to provide him with information about the Customer's account (modelled by a resource dependency *'Account Information'*) and to transfer the money from the Customer's account to the Mail Order Company's account (modelled by a goal dependency *'TransferMoney'*). We add to the resource dependency a softgoal dependency which qualifies the resource: *'Accurate [Info]'*.

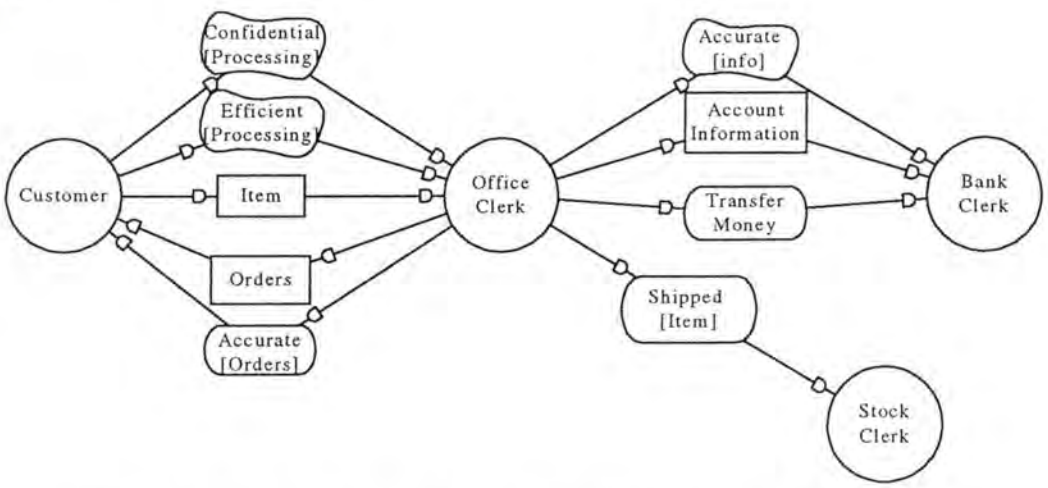


Figure 30: Identification of the Office-Clerk, StockClerk and Bank-Clerk

We now have already 'a lot of mess' between the different actors: some are internal and some are external to the society; some are individual and some are classes or societies of actors. It is time to clarify the different concepts.

1.1.3 'Specialisation' of the Actors

We will now introduce the different concepts *i** offers for restructuring actors. In addition, we will make the distinction between classes and instances.

First, in order to simplify the example and to focus on the elaboration of the steps, we assume that there are any number of customers, one Order Processing Clerk, one Bank Clerk and one Stock Clerk. Consequently, customers are represented as a class and the other ones as individuals. In *i**, the difference can only be seen in the textual declaration (in TELOS); there does not exist an explicit graphical representation of classes and instances.

Figure 31 identifies the decomposition/specialisation of the aggregate 'Mail Order Company' into its components: Office Clerk and Stock Clerk. This decomposition does not have to be complete; it only depicts the actors we are interested in. In our example, the Office Clerk and the Stock Clerk play both one role: '*Order Processor*' and '*Shipment Processor*'. This is a simple coincidence; it is completely possible that an agent plays a multitude of roles. This would be the case if, for instance, the Office Clerk would play the two roles: the one of receiving the orders and the one of shipping the items away.

We represented the Mail Order Company, the Office Clerk and the Stock Clerk as agents because we wanted to insist on the identity of these actors. The Mail Order Company was *the* company that ordered for the information project and the Stock Clerk and the Office Clerk are tangible, social actors; the ones that will be influenced by our new system. However, one could also decide to represent them as positions. They would hence have a more general meaning and refer more to abstract entities of the real world than to special actors.

The '*Bank*' is represented as a position because we do not refer a specific bank organism. Since then the Bank can be identified as any entity that offers a set of services. Furthermore, our '*Bank Clerk*' is only defined by a set of two roles (transferring money and informing about the credibility of accounts). The position 'Bank Clerk' assumes all dependencies and we gain hereby the liberty not to go too much into detail about the definition of roles, the Bank Clerk has to play.

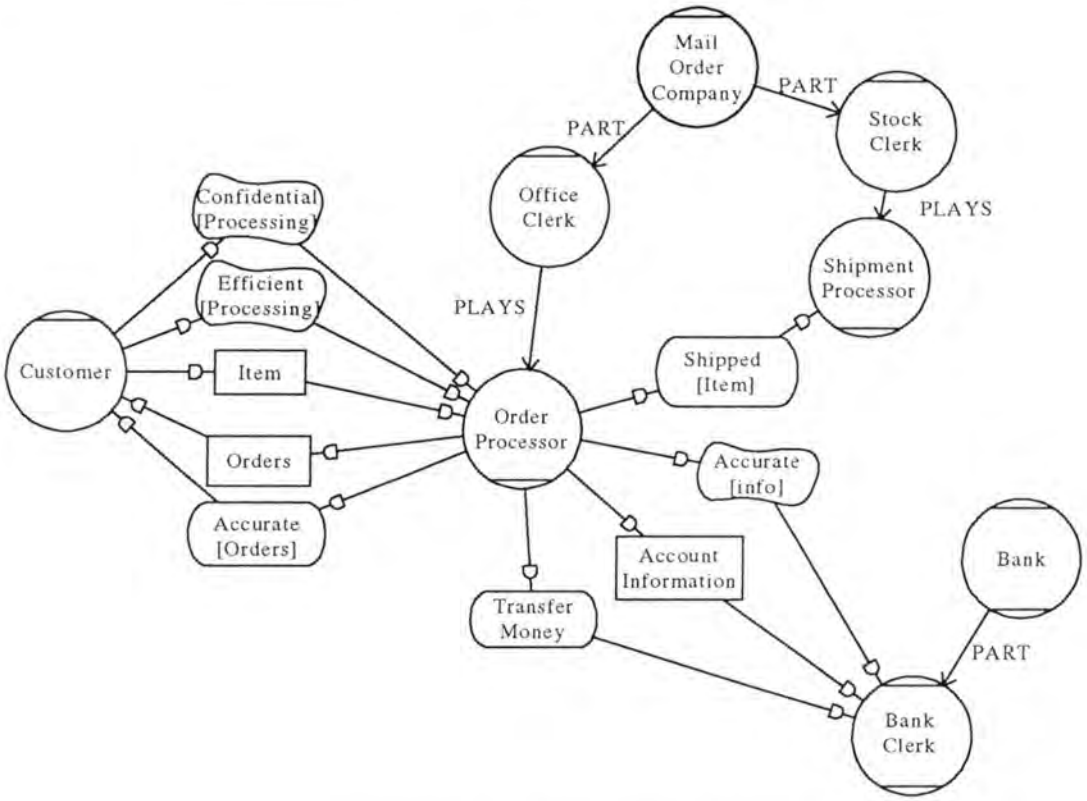


Figure 31: Agents, Positions and Roles

Restructuring of Dependencies.

In our example, most dependencies are ascribable to the roles. They do not depend upon the fact that a precise person or agent responds to the dependency and no important skills are needed.

In the previous chapter, we said that ideally a role reacts towards one dependency. Thus, one might think that we broke up with this 'rule'. But this is not the case. If we take, for instance, the Office Clerk with his role 'Order Processor'. We see that the incoming dependencies all deal with the activity of processing an order for an item and hence are related. The role 'Order Processor' has to react towards the main (resource) dependency 'Item' with a certain behaviour. The softgoal dependencies 'Efficient[Processing]' and 'Confidential[Processing]' are qualitative requirements. They will both have influences at the requirements engineering phase and the design phase. The other thing that they have in common is that they both are related to this behaviour that the role 'Order Processor' has to react with: this behaviour has to be efficient and confidential.

In *i**, a position can be defined as a set of behaviours and we believe that this is the right concept for modelling the 'Bank' and the 'Bank Clerk'. As a result, we can also assign many dependencies towards this position.

1.2 Identification of the ALBERT Model Boundary

We now identify the actors that we are going to introduce in the ALBERT specification. This is already a step that has a big influence on our set of possible alternatives. The actors we identify at this stage define the process that we assume to be unsatisfactory. Hence if we find out, at the following steps, that the description of the process is not broad or detailed enough, we have to go back to this step to add the necessary actors.

Although we have in the appendix, for the sake of completeness, described all the agents in ALBERT, this would not be necessary in reality. We identify the actors *'Customer'*, *'Bank'*, *'Bank Clerk'* as being out of the ALBERT model boundary. We are only interested in their behaviour with respect to the office clerk's and stock clerk's interaction. Indeed, the behaviour of the office clerk is initiated by the customer's request. Furthermore, the bank clerk has influences on the possible behaviour of the office clerk. We will only declare them with the actions or state components that are shared between these actors.

This step will lead to the following ALBERT declaration of the four agents Customer, Stock Clerk, Office Clerk and Bank Clerk:

CUSTOMER	OFFICE CLERK
STOCK CLERK	BANK CLERK

The two actors 'Office Clerk' and 'Stock Clerk' could be regrouped into the society 'Mail Order Company'. We obtain the following structure:

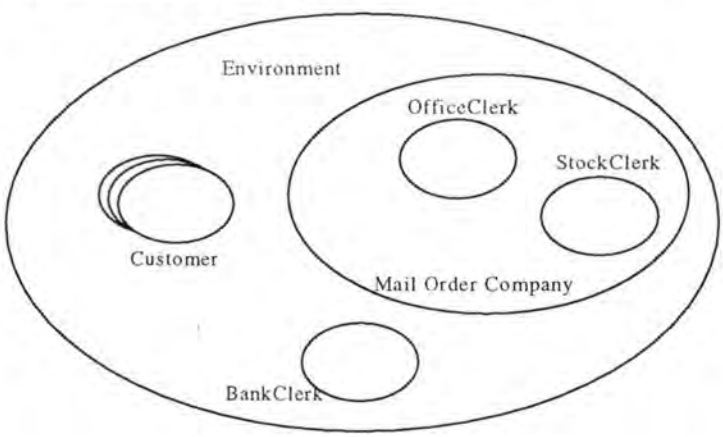


Figure 32: The Environment

1.3 Identification of the behaviour

The next step describes the behaviour in the *i** model and in the ALBERT model. When we talk in *i** about behaviour, we mean the internal, intentional behaviour that is behind

the Actors dependency links. The ALBERT model, on the other hand, tries to describe a process, that is unsatisfactory.

1.3.1 Elaboration of the Strategic Rationale Model

The *'Office Clerk'* responds to the main incoming dependency link *'Item'* by a task. In figure 33, we see that besides the dependencies between the *'Customer'* and the role *'Order Processor'* the task has also to meet the condition to produce a maximum of profit. This softgoal comes from the partially delegated Softgoal 'Make Maximum Profit'. The task 'Process[Order]' has as a sub-task *'Verify[Orders]'*. We decided to add this task because it constraints our processing, i.e. the Order Processor has to process the orders in certain way, and we do not know yet if this process also has to be changed. Before the orders can be processed, one has first to verify whether the listed item is sold. The 'Verify[Order]' task is connected to the 'Accurate[Orders]' dependency: the importance and the difficulty of this task depends upon how the orders have been filled in.

We now also see that the processing of an order is constrained by two conditions that have to be fulfilled for a successful processing. The first condition is that the account of the customer is in order, i.e. the account does not have to be in the red (represented by the task-goal decomposition *'Account OK'*. The verification of the condition depends upon the information the Order Processor receives from the Bank Clerk. We added a softgoal *'Accurate[Info]'* because it is important to receive the information about the respective customer's account that is the *most* recent. Another reason to represent this as a softgoal is because it will also have strong influences on design decisions: its achievement cannot be fully stated as a functional property.

The second condition the process has to bring about is that the money is transferred on the company's account: the goal *'Money Transferred'*. The main actors that are interested in this goal are of course the Mail Order Company and the Shareholders.

The two softgoals *'Less Errors'* and *'Fast Turnaround'* are derived from the incoming dependency 'Efficient Processing' and have to be achieved by each sub-element of the decomposed task 'ProcessOrder'.

We decided not to represent the responding behaviour on the 'Bank's side because we are not interested in its detailed behaviour and we assume that this Bank has the necessary processes to reply to the dependencies.

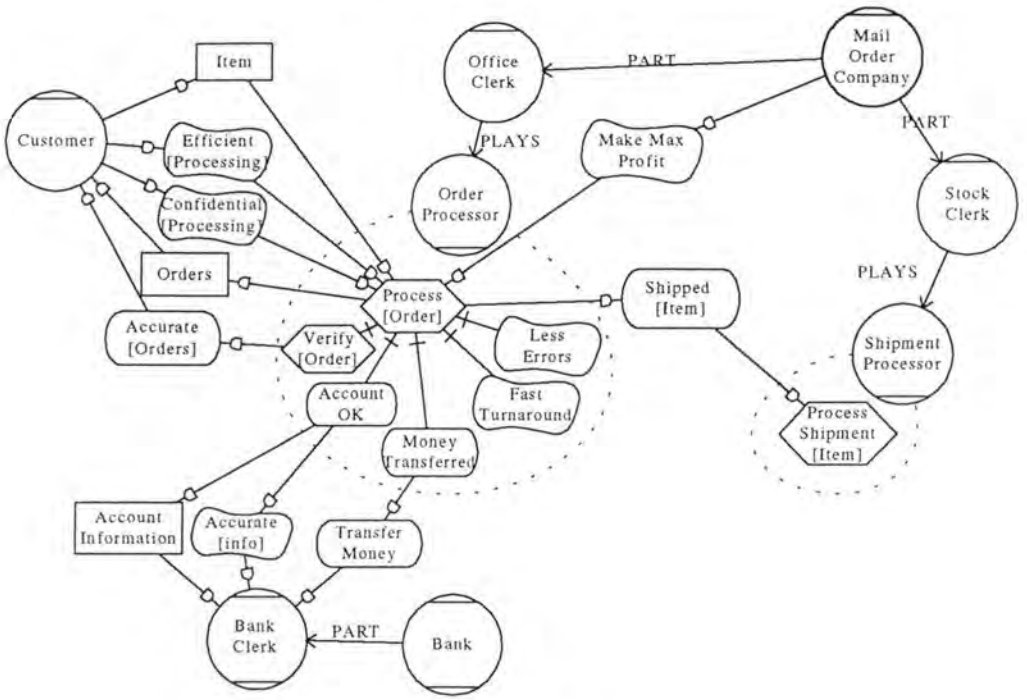


Figure 33: The SR Model

1.3.2 Elaboration of the ALBERT Model

In order to give an idea of the elaboration of ALBERT specification, we describe it as a sequential process, i.e. series of ordered actions that bring about a result (the specification). The approach we use here is only one possible way to proceed but not the only one. Furthermore, this is a very optimistic approach and we have to say that in reality this process is often much more disordered.

First, we identify a list of state components and data types that we think are necessary to describe the agents. We only concentrate on the two main agents the '*Stock Clerk*' and the '*Office Clerk*'. The whole specification can be found in the appendix.

The declaration data types is self-explanatory:

```
BASIC TYPES
    VISA
        | the credit card information the customer adds to the order
    ITEM
        | the delivered item
    ITEMTYPE
        | the type of the ordered item
CONSTRUCTED TYPES
    ADDRESS = CP[Name:STRING, Street:STRING, Locality: STRING]
        | the address of the customer
```

ORDER = CP[OrderId:INTEGER, Person: ADDRESS,Item: ITEMTYPE]
| the order of the customer

INVOICE = CP[OrderId: INTEGER, Person: ADDRESS, Item: ITEMTYPE, Sum: INTEGER]
| the invoice that the customer receives and which is produced by the office-clerk

State Components. The Office Clerk has two *state components*: *'Busy'* which describes whether he is occupied with doing something or free to process an incoming order and *'SoldItemTypes'* where all the items sold by the company are listed.

The Stock Clerk has as a state component the real stock: *'Stock'* which is modelled by a table of sets of items which are indexed by an item-type.

OFFICECLERK
DECLARATIONS
STATE COMPONENTS
Busy instance-of *BOOLEAN*
SoldItemTypes set-of *ITEMTYPE*
STOCKCLERK
DECLARATIONS
STATE COMPONENTS
Stock table-of *SET[ITEM]* indexed-by *ITEMTYPE*

Initial valuation and Derived Components. The next step consists in defining the *initial valuation* of the state components and the introduction of *derived components*. One should however note that state components do not obligatory have to be initiated. Also the use for derived components is only useful if it clarifies the structures of constraints or it represent an entity of the real world. We define the initial value of the Office Clerk's state component 'Busy' as being 'false' and the Stock Clerk's stock as being empty.

OFFICECLERK
BASIC CONSTRAINTS
INITIAL VALUATION
Busy = FALSE
STOCKCLERK
BASIC CONSTRAINTS
INITIAL VALUATION
Stock[_] = {}

Effects of Actions. State components cannot change by themselves and we introduce the *actions* of interest, which have an *effect* upon the state components.

For the Office Clerk, we assume that the change upon the 'SoldItemTypes' state component is of no significant importance for our modelling; we will not define any effect of action upon that state component. The state component 'Busy' is true during the processing of an order and false after the processing of an order has been accomplished.

For the Stock Clerk, the *'Add'*-action is not absolutely necessary for our purpose but for the sake of completeness we introduce it. One can add or remove items from the Stock Clerk's stock.

OFFICECLERK	
OPERATIONAL CONSTRAINTS	
EFFECTS OF ACTIONS	
Continue(_,_,_) :	*Busy := TRUE*
	[]
	Busy := FALSE

STOCKCLERK	
DECLARATIONS	
ACTIONS	
RemoveFromStock(ITEMTYPE,ITEM)	
AddToStock(ITEMTYPE,ITEM)	
OPERATIONAL CONSTRAINTS	
EFFECTS OF ACTIONS	
AddToStock(i,it) :	*[] Stock[i] := (it ∪ Stock[i])*
RemoveFromStock(i,it) :	*[] Stock[i] := (it \ Stock[i])*

Action Structure. Until now, the occurrences of actions are unstructured: occurrences can happen at every moment in time. The next step consists in *structuring the occurrences of actions*. This is done in our example mainly by the introduction of processes in the 'Action Decomposition' template although other templates can also be used[9]. Of course, all these actions have to be declared. Refer to the appendix for the declaration part.

Two actions, i.e. *'ProcessPayment'* and *'AlarmCustomer'*, are constrained by a state component: *SoldItemTypes*. Relating these preconditions to the action composition we obtain an 'if-structure'. If the item is sold by the company the Office Clerk proceeds with the process and does nothing otherwise.

One can also note the importance of the first line (under the Action Composition header) which enumerates the actions that can only occur inside its respective composition.

The cancelling of a process due to a rejection of the bank clerk has not been modelled.

[9] Action Duration, Precondition, Triggering and State Behaviour

OFFICECLERK

DECLARATIVE CONSTRAINTS

ACTION COMPOSITION

{ C.Order, Continue,AlarmCustomer, ProcessPayment, DebitRequest, BankClerk.AcceptOrder, TransferOrder, OrderShipment, BankClerk.RejectOrder }

ProcessOrder ↔ C.Order(o,vi) <> Continue(o,vi,C)

Continue(o,vi,C) ↔ AlarmCustomer(o,C) ⊕ ProcessPayment(o,vi)

ProcessPayment(o,vi) ↔
DebitRequest(am,vi) <> (
BankClerk.RejectOrder(am, vi) ⊕
(BankClerk.AcceptOrder(am, vi) <> TransferOrder(am,vi,account) <> OrderShipment(inv)))

OPERATIONAL CONSTRAINTS

PRECONDITIONS

AlarmCustomer(o,_) : Item(o) ∉ SoldItemTypes

ProcessPayment(o,_) : Item(o) ∈ SoldItemTypes

STOCKCLERK

DECLARATIVE CONSTRAINTS

ACTION COMPOSITION

{ OfficeClerk.OrderShipment, RemoveFromStock, Ship}

ProcessOrderShipment ↔
OfficeClerk.OrderShipment(inv) <> (
(RemoveFromStock(Item(inv), it) <> Ship(inv, it))
⊕ DAC)

OPERATIONAL CONSTRAINTS

PRECONDITION

RemoveFromStock(i,_) : Card (Stock[i]) > 0

Cooperation Constraints. We already defined external actions into our structure of the actions. In order to really perceive the occurrence of actions and the value of state components, we have to define the information/perception 'templates'. Nearly all action occurrences are always perceived in our example, except the 'Order' action by the customer which is only perceived by the Office Clerk when he is not occupied.

OFFICECLERK

COOPERATION CONSTRAINTS

ACTION PERCEPTION

K (C.Order(_,_) / TRUE)

K (BankClerk.AcceptOrder(_,_) / TRUE)

K (BankClerk.RejectOrder(_,_) / TRUE)

ACTION INFORMATION

K (OrderShipment(_).StockClerk / TRUE)

K (DebitRequest(_,_).BankClerk / TRUE)

K (TransferOrder(_,_,_).BankClerk / TRUE)
XK (AlarmCustomer(_,C^1).C^2 / C^1 = C^2)
STOCKCLERK
COOPERATION CONSTRAINTS
ACTION PERCEPTION
K (OfficeClerk.OrderShipment(_) / TRUE)
ACTION INFORMATION
K (Ship(_,_).Mail / TRUE)

1.4 Definition of the Objective

We now describe the problem informally in natural language: the order processing clerk makes out the invoice and orders the shipment even if the items are not in the stock. For the production of the invoice he has to interact with the Bank Clerk who belongs to another organisation. This leads to a lot of inefficiency (paperwork for items that finally cannot be shipped) and annoyance on the customer's side. Therefore, we consider the following objective:

Objective (in natural language):

> The Order Processing Clerk can only check out the customer's account, make the invoice, and order the shipment if the respective item is available (i.e. Card(Stock(Item)) > 0)

This objective is reformulated as a first order formula which stands above all agents and which has to be fulfilled by the set of agents we are going to introduce in the next step.

Formulation of the objective: (as an 'overall' precondition)

> *OfficeClerk.ProcessPayment(o,_) : Card(StockClerk.Stock(Item(o))) > 0*
>
> the order processing clerk can only proceed when the stock of the requested item is not empty

On the ***i**** side, we introduce a new role, which belongs now to the system: the one of an *'Efficient Order Processor'*(figure 34). This role is yet not assigned to a concrete agent. In opposition to the old 'Order Processor', the behaviour has been changed to include the new constraint, i.e. the objective above. The 'Efficient Order Processor' serves as an intermediate actor: on the one hand he still belongs to the (old) problem environment and on the other hand he also belongs to the (new) system.

This objective can be seen as a responsibility derived from the *'Efficient[Processing]'* softgoal (with the customer as the depender) and the *'Max Profit'* softgoal. We represented this by the positive contribution links in figure 34.

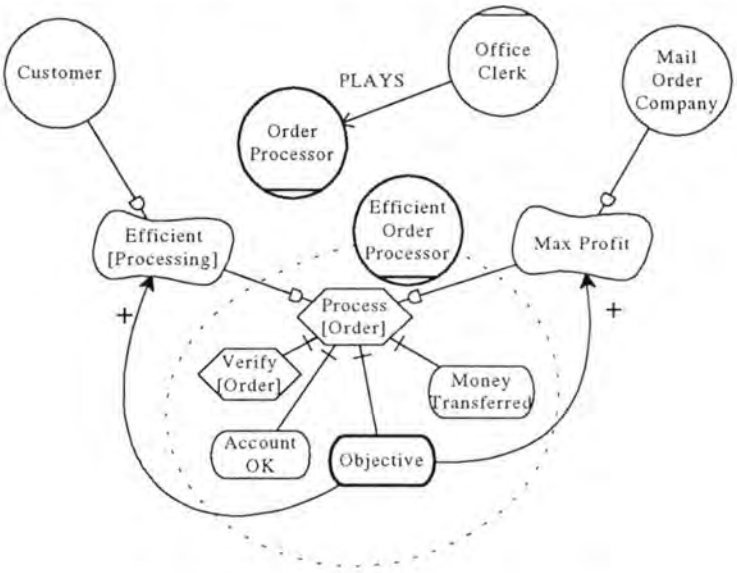


Figure 34: The new 'Objective-Role'

2. System Requirements

We are now going to introduce our Omni-System that considers the objective; the functional requirement specified by the client: orders can only be processed when the desired item is in the stock.

Furthermore all the non-functional requirements have to be expressed. The Strategic Rationale Model is enriched by softgoal dependencies towards the system.

As already said in the previous chapter, the Omni-System[10] is characterised by the two following properties:

- omniscience: we introduce for the system a perfect perception of the environment: the Stock Clerk and the Office Clerk.
- omnipotence: the system can have an impact on the behaviour of the problem environment: the Stock Clerk and the Office Clerk.

We first choose the means by which the environment is controlled in order to certify the satisfaction of its behaviour (System-Output) and then only we decide how the system makes this decision and what are the necessary resources (System-Input).

First, we create an actor 'System' in our Strategic Rationale Model. We do not define it as an agent, position or role because at this stage we do not have a precise idea about what the system might look like. In addition, we link the objective to the system by a goal dependency: the Efficient Order Processor becomes dependent upon the system to achieve the goal. This sub-objective represents the information we need to constrain the behaviour of the efficient order processor.

[10] In this section, we use the terms system and Omni-System indistinguishably.

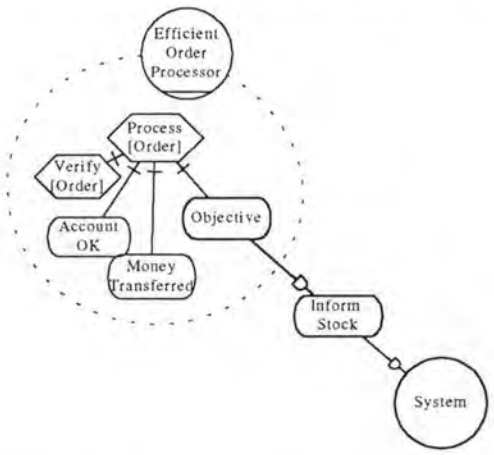


Figure 35: The introduction of the System into the SR Model

2.1 Identification of the System-Output

We next introduce the different media by which the system can have the necessary impact upon the problem environment.

2.1.1 Identification of the system-output in *i**

There are some possibilities to represent the influence upon the 'Efficient Order Processor's behaviour[11]:

1. We could for example let the 'Efficient Order Processor' depend upon a *resource* *'Stock Info'*. The 'Efficient Order Processor' gains hereby the ability to use the resource *'Stock Info'* but it is still able to decide itself what to do when the incoming information says that the stock is empty. Thus, we will have to constraint furthermore the 'Efficient Order Processor' if we want the problem to be solved. We duly note that although this might look like an ALBERT state perception, this does not have to be the case (consider for instance arguments of imported action).
2. Another possibility is to represent the dependency as a *task dependency*. We mean by this that the 'Efficient Order Processor' is interested in a particular way the goal of receiving the information about the stock should be achieved. We understand by 'particular way', the acceptance or the rejection of the order. In opposition to the previous solution, this dependency is more restricting because the 'System' gains hereby a control over the behaviour of the 'Efficient Order Processor' who simply does what the system commands. Further decomposed, this task dependency could result in two task dependencies *'Accept'* and *'Reject'*. These tasks would then control the behaviour of the Efficient Order Processor'.

11 Actually, these are two different functionalities needed from the 'reduced' system: (1) give an information so that the office clerk can make the necessary decision and (2) control the behaviour of the office clerk.

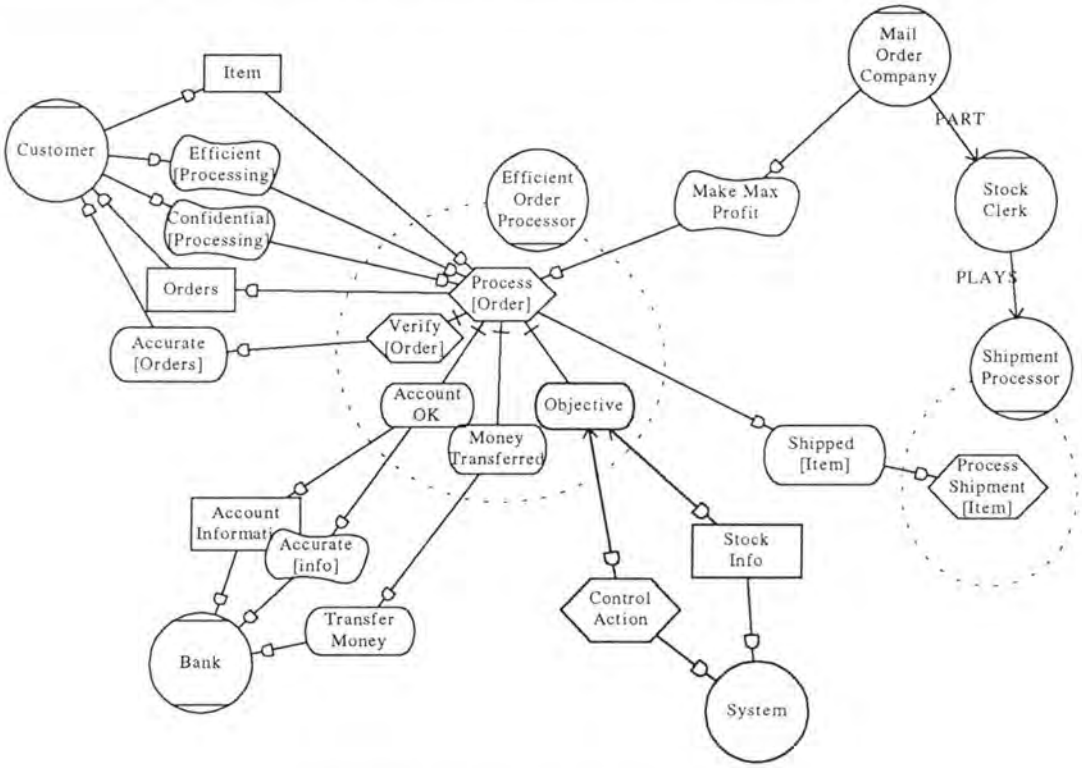


Figure 36: Two alternatives of the system-media

2.1.2 Identification of the system-output in ALBERT

For the ALBERT part, we also consider the two possibilities. We should insist once more that the link between the *i** and the ALBERT part is not based upon the fact that we use a state information/perception mechanism in the first solution and an action mechanism in the second possibility.

The difference is more characterised by the importance of the (reduced) Omni-System's responsibility. The distinction is also made upon the place where the decision is made.

1. State Perception/Information (and Precondition later on)

   The (reduced) Omni-System only provides the environment with some information about the stock ('*Inv*'). It is upon the environment to take the necessary measures to enforce the objective. Some part of the decision making process is put in the interacting system (see section 2.3.2 of this chapter).

SYSTEM
STATE INFORMATION *K (Inv[_]. Efficient_Order_Processor / TRUE)*
EFFICIENT_ORDER_PROCESSOR
STATE PERCEPTION *K (System.Inv[_] / TRUE)*

2. Action Perception/Information

   For the second solution, we choose two actions which will have an influence on the behaviour of the 'Efficient Order Processor'. These actions are made visible towards the Efficient Order Processor.

SYSTEM
ACTION INFORMATION *K (AcceptOrder(_).Efficient_Order_Processor / TRUE)* *K (RejectOrder(_).Efficient_Order_Processor / TRUE)*
EFFICIENT_ORDER_PROCESSOR
ACTION PERCEPTION *K (System.AcceptOrder(_) / TRUE)* *K (System.RejectOrder(_) / TRUE)*

2.2 Identification of the System-Input

The Omni-System somehow has to make decisions which are based upon knowledge about the environment. We are now going to introduce this necessary information to the system.

2.2.1 Identification of the system-input in *i**

The system has to provide the '*Efficient Order Processor*' with information about the real stock. Consequently, the information needed from the environment is the Stock Clerk's state component '*Stock*'. In ***i****, we represent this by a resource dependency on the shipment processor. We do not differentiate here between the two solutions: in both cases the system has to hold *that* information. The solution with the controlling actions needs one further information: the one about the moment at which the controlling actions should occur: represented by '*Stock Request*' in figure 37.

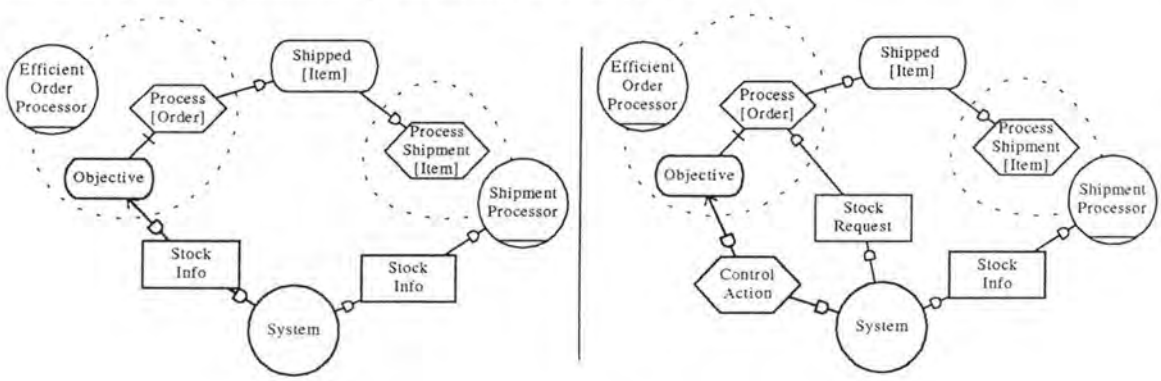


Figure 37: Identification of the Environment-Media

2.2.2 Identification of the system-input in ALBERT

1. State Perception/Information

   The System needs the information about the real stock. On the System's side, we have to introduce a state perception constraint and on the Stock Clerk's side a state information constraint. Because the System has the omniscience property, the formulas in the information/perception statements are always true.

SYSTEM
STATE PERCEPTION *K (StockClerk.Stock[_] / TRUE)*
STOCKCLERK
STATE INFORMATION *K (Stock[_].System / TRUE)*

2. Action Perception/Information

   Besides the information about the stock, we have to introduce the request by the office clerk for the occurrence of the action. The Efficient Order Processor requests the stock explicitly: Action Information *K(StockRequest(_).System / TRUE)* and the System always perceives the request: *K (Efficient_Order_Processor.StockRequest(_) / TRUE).*

SYSTEM
ACTION PERCEPTION *K (Efficient_Order_Processor.StockRequest(_) / TRUE)* STATE PERCEPTION *K (Stock[_].StockClerk / TRUE)*
STOCKCLERK
STATE INFORMATION *K (System.Stock[_] / TRUE)*
EFFICIENT_ORDER_PROCESSOR
ACTION INFORMATION *K (StockRequest(_).System / TRUE)*

2.3 Identification of the System's behaviour

We understand by System's behaviour the behaviour that spans over the whole Omni-System. So also the interacting Omni-System has to be considered (see section 2.3.2).

2.3.1 Identification of the behaviour in *i**

In both solutions, we only introduce a task to which the respective dependencies will be connected. We indicate by this representation that the system has a process for accomplishing the goal, task or resource, i.e. ability.

For our example (see figure 38), we restraint ourselves to the mere representation of this task *'ProcessInfo'*. A deeper understanding of the 'composition' of the process can be found in the ALBERT part.

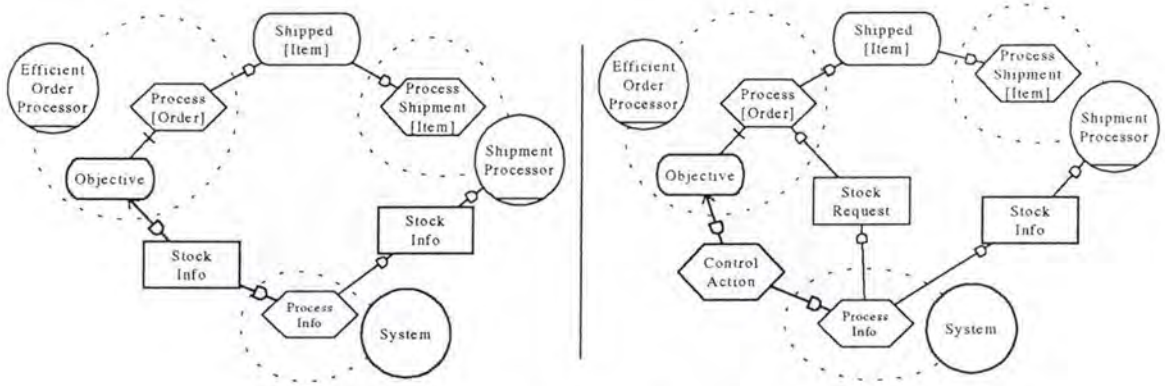


Figure 38: Identification of the behaviour

2.3.2 Identification of the behaviour in ALBERT

The system is now in the possession of the necessary output to control the environment and the necessary knowledge about the environment. We now define how the system uses the information to influence the behaviour of the environment.

1. State Perception/Information and Precondition

   The first solution aimed more in giving a 'passive' role to the reduced system: it only provides the Efficient Order Processor with some information about the stock. To 'resolve' the problem and hence to obtain the achievement of our objective (the process only continues if the stock is not empty), we have to add a precondition constraint upon the behaviour of the Efficient_Order_Processor:

   $$ProcessPayment(o,\_) : System.Inv[Item(o)] \neq \{\}.$$

   This constraint forbids the Efficient_Order_Processor to continue the process if the stock is empty.

   The reduced Omni-System only has to derive the stock from the real stock. Because ALBERT does not allow the use of external state components in the right-hand expression of derivation rules, we need to introduce a state behaviour constraint and an effect of action, responsible for changing the value of the stock.

EFFICIENT_ORDER_PROCESSOR
PRECONDITIONS *ProcessPayment(o,_) : System.Inv[Item(o)] ≠ {}*
SYSTEM
STATE BEHAVIOUR *[ ] Inv[i]: = Card(StockClerk.Stock[i])*

EFFECTS OF ACTIONS
Act: *[] Inv[i] = alpha*

2. Action Perception/Information, Precondition and Action Composition

   In this solution the responsibility of the (reduced) system is increased. Upon receiving a request by the *Efficient_Order_Processor* (*StockRequest(o)*), the system decides whether the Efficient_Order_Processor continues the processing. The decision whether to continue the process or not depends upon the state of the Stock Clerk's stock. For the sake of convenience, we will not model the cancelling process that follows a rejection by the system.

SYSTEM
ACTION COMPOSITION
ProcessControl(o) ↔ ***Efficient_Order_Processor.StockRequest(o)*** *< >*
(AcceptOrder(Item(o))⊕ RejectOrder(Item(o)))
PRECONDITION
AcceptOrder(i) : ¬ (Card(StockClerk.Stock[i]) ≤ 0))
RejectOrder(i) : (Card(StockClerk.Stock[i]) ≤ 0))
EFFICIENT_ORDER_PROCESSOR
ACTION COMPOSITION
ProcessOrder ↔ *C.Order(o,vi) <> Continue(o,vi,C)*
Continue(o,vi,C) ↔ *(AlarmCustomer(o,C) ⊕*
*(**StockRequest(o)** <> (*
*(**System.AcceptOrder(o)** <> ProcessPayment(o,vi))*
*⊕ **System.RejectOrder(o)**)))*

2.4 Alternative System Modelling

In this section we describe the alternative representation of the system in ALBERT. We do not introduce a special 'System' agent but instead we change the problem environment to 'generate' the solution environment. The system is identified by the difference of the solution environment and the problem environment.

The modelling of the system in ***i**** does not change: we still identify an actor 'System'. In the previous section, we identified two solutions: - the accept/order actions and - the perception of information and the precondition. We saw that the latter solution led to some awkward constraints. We will show that the alternative system approach can introduce some specification facilities.

We adopt the same approach as for the previous system modelling:

1. We introduce the necessary information, i.e. the knowledge about the stock, by the state perception constraint of the Office Clerk.

2. We identify the counterpart of this state perception constraint, i.e. the necessary state information constraint in the Stock Clerk's template.
3. We introduce the structure or behaviour of the system by the additional precondition for the action 'ProcessPayment'.

OFFICECLERK

DECLARATIVE CONSTRAINTS

ACTION COMPOSITION

ProcessOrder $\leftrightarrow$ *C.Order(o,vi)* <> *Continue(o,vi,C)*

Continue(o,vi,C) $\leftrightarrow$ *AlarmCustomer(o,C)* $\oplus$ *ProcessPayment(o,vi)*

OPERATIONAL CONSTRAINTS

PRECONDITIONS

AlarmCustomer(o,_) : Item(o) $\notin$ *SoldItemTypes*

ProcessPayment(o,_) : Item(o) $\in$ *SoldItemTypes* $\wedge$ ***StockClerk.Inv[Item(o)] > 0***

COOPERATION CONSTRAINTS

STATE PERCEPTION

K (StockClerk.Stock[ ] / TRUE)

STOCKCLERK

DECLARATIONS

STATE COMPONENTS

Stock $^{table\text{-}of}$ *SET[ITEM]* $^{indexed\text{-}by}$ *ITEMTYPE* $\rightarrow$ ***OfficeClerk***

COOPERATION CONSTRAINTS

STATE INFORMATION

K (Stock[ ].OfficeClerk/ TRUE)

Although we cannot generalise from this example, we already see that this alternative can facilitate the specification of the solution environment.

This kind of system modelling in ALBERT increases the importance of the linkage to the *i** model. The actor 'System identifies now the additional and altered constraints (represented by the 'bold' constraints).

2.5 Evaluation of the functional alternatives

Besides the functionalities the system has to provide, the client is interested in the non-functional or qualitative requirements. Indeed, these can have an important influence on the desired product.

First, we find softgoals underlying the processing of the orders[12]: '*Less Errors*' and '*Fast Turnaround*'. These softgoals are consequences of the softgoal dependency '*Efficient[Processing]*' and have to be ideally fulfilled by every element the task '*Process[Order]*' is composed of.

[12] We note that these softgoals existed already although we did not represent them explicitly.

The different alternatives *'StockInfo'* and *'ActionComposition'* are compared considering the two softgoals. This evaluation of alternatives is left to the judgement of the clients and another evaluation might be possible. Especially with the development of a concrete solution, the clients and the analysts get a better understanding of the softgoals and the evaluation might be revised.

In Figure 39, we observe that the 'Control Action' alternative improves the fast turnaround and lowers the rate of mistakes. We choose this latter solution because it corresponds more to our requirements.

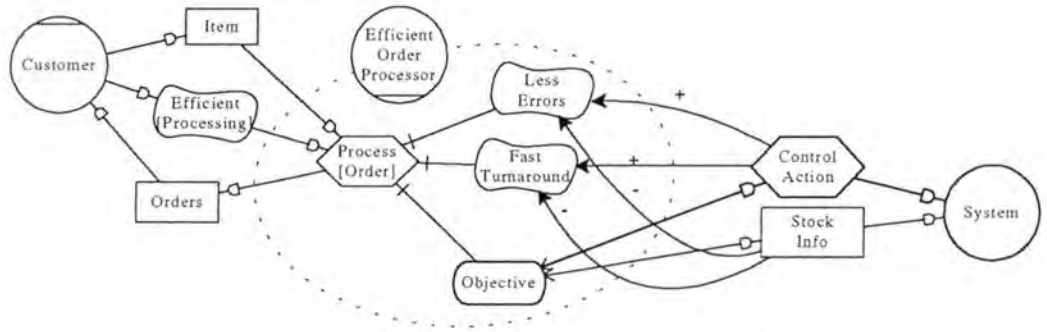


Figure 39: Evaluation of alternatives

2.6 Identification of Softgoals

We introduce the non-functional requirements or softgoals which are of importance for the user. In Figure 40, we identify three softgoals that should be considered by the system. For the moment, they remain softgoals because we do not yet want to explore their detailed meaning. We are more interested in their elicitation.

The two softgoals 'FastProcessing' and 'AccurateInfo' partially derive from the customer's 'EfficientProcessing'-softgoal. The softgoal 'LowCost' reflects the Company's view (increase the profit).

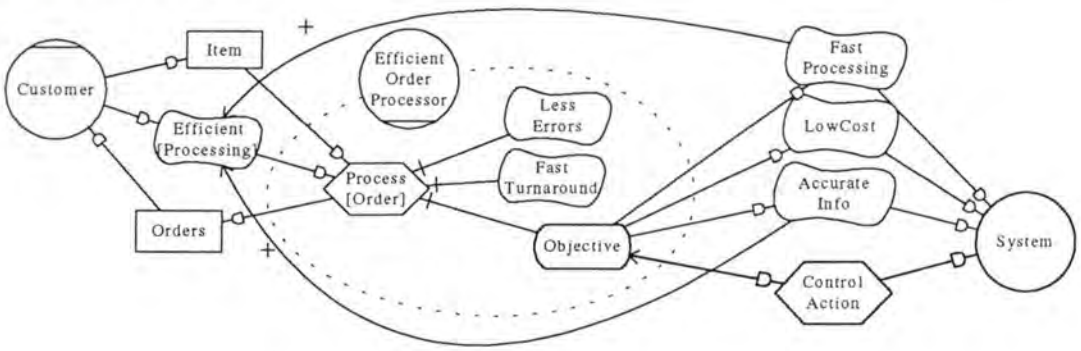


Figure 40: Introduction of Softgoal-Dependencies

Because a detailed understanding is of importance for the system development and the selection of the concrete implementation, we describe the softgoals more precisely in natural language:

- FastProcessing: The time between the stock request and answer by the system is less than a yet undefined time period.
- LowCost: The cost of development and maintenance should be less than a unknown amount.
- AccurateInfo: The information reflected by the acceptance or the rejection reply must be accurate with the information about the stock.

2.7 Refinement of Softgoals

In order to achieve a better understanding of the softgoals, we refine the previously identified softgoals. In Figure 41, we concentrate on the accuracy softgoal, which is decomposed into:

- A transfer protocol softgoal. The finally implemented communication between the different agents has to be errorless. This requirement remains a softgoal because we do not yet know what kind of protocol we will use. It will guide the system developer in choosing from several communication protocols.
- Knowing that the system cannot always request the availability of the items from the Stock Clerk, in accordance with the fast processing softgoal, the system has to maintain a mirror of the real stock and this mirror has to be updated as fast as possible by the Stock Clerk. The goals 'Mirror' and 'Update Mirror' will lead to the introduction of a state component 'Inv' and its modifying actions in ALBERT.
- The Office Clerk has to be informed by the system in a minimum of time. This leads to a motivation for keeping track of the alteration of the real stock. In addition, we describe a property which has to be fulfilled by the system:
    - $OfficeClerk.StockRequest(\_) \Rightarrow WithinF_{1''}(System.AcceptOrder(\_) \vee System.RejectOrder(\_))$

      ```
      if the Office Clerk request the availability of an item, the
      response has to occur within 1''
      ```

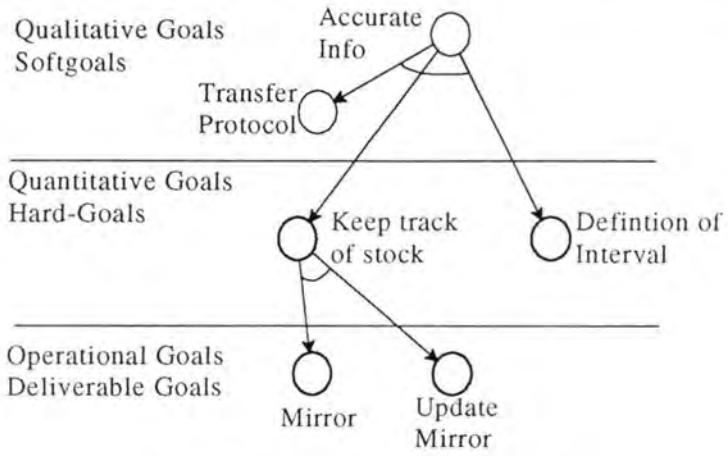


Figure 41: Refinement of Softgoals

3. System Specification

In our example, the Omni-System only played one role and we didn't represent this explicitly. Consequently, we can directly transform the actor 'System' into a role 'Stock Informant'. This role can now be played by a specific agent.

For the elaboration of the system specification, we consider two alternatives:

- In the first, we create an Information System, which has to control the behaviour of the Office Clerk.
- In the second, we assign the functionality specified at the previous step to the Stock Clerk, who now plays two roles: shipping the items away and replying to the stock-information requests.

3.1 System Analysis/Design

3.1.1 The Information System Solution

In this alternative, the role of replying to stock requests is played by an information system[13]. (see Figure 42)

[13] We didn't introduce special agents responsible for the communication (like terminals, user interfaces,...) between the two entities because of the simplicity of the case study.

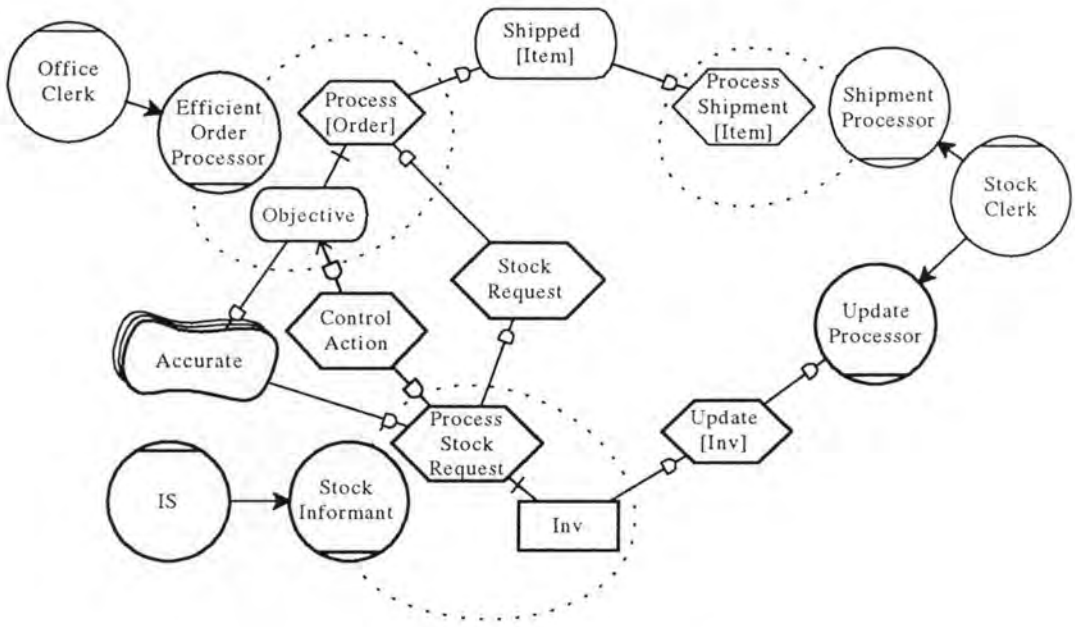


Figure 42: The IS Solution[14]

*i** Modelling

In Figure 42, we first create an agent '*IS*' who plays the role of providing the Office Clerk with information about the stock (represented by the role '*Stock Informant*') and influencing by this means the behaviour of the office clerk. This is done by the previously selected Accept-Reject mechanism modelled by the task dependency '*Control Action*'.

The information system has now to maintain its own stock information because of the '*Fast Processing*' softgoal. The task '*Process Stock Request*' and consequently the decision whether the item is available or not, now depends upon the information maintained by the information system about the stock (i.e. 'Inv', themirror of the real world stock). We introduced this concept also because of the retraction of the omniscience assumption. The information system has no longer a perfect visibility upon the real stock. Accordingly, we also have to think about the way the mirror of the stock is maintained. We introduce the task dependency '*Update Stock*' to refer to this need. The modelling of the 'Update Stock' dependency as a task dependency indicates that, since we deal with a concrete information system, the Stock Clerk has to operate in specified way. Every time he adds items to the stock or removes items from the stock he has to update the virtual stock.

The 'Update [Inv]' dependency will be linked to the newly created role '*Update Processor*' played by the Stock Clerk. We do not consider a more detailed description of the 'Update Processor' and we refer to the ALBERT part to get a better knowledge about his behaviour.

[14] The softgoal 'group' Accurate stands for the three softgoals dependencies identified at the previous step (step2)

Within this update-solution for the retraction of the perfect visibility assumption, we ignore possible unreliabilities in the communication between the Stock Clerk and the Information System, for instance transmission problems, etc.

One may also notice that the previously resource dependency *'Action Occurrence'* changed to a task dependency 'Action Occurrence'. We interpret this as a more constraining dependency: since we deal with a concrete information system, the information has to be introduced in a special way, using a user interface for instance.

ALBERT Modelling

After we have introduced the system in the ***i**** model we specify the information system in ALBERT. In this case study we identify only the agent *'Information System'* and ignore agents like user interfaces, sensors, etc.

IS

DECLARATIVE CONSTRAINTS

ACTION COMPOSITION

{ OfficeClerk.StockRequest, AcceptOrder, RejectOrder}

ProcessStockRequest(i) ↔ OfficeClerk.StockRequest(i) <>
(AcceptOrder(i) ⊕ RejectOrder(i))

OPERATIONAL CONSTRAINTS

PRECONDITIONS

*AcceptOrder(i) / **(Inv[i] - 1) ≥ 0***

*RejectOrder(i) / **(Inv[i] - 1) ≤ 0***

EFFECTS OF ACTIONS

AddToStock(i,) : *[]*
Inv[i] := Inv[i] + 1

RemoveFromStock(i,) : *[]*
Inv[i] := Inv[i] - 1

COOPERATION CONSTRAINTS

ACTION PERCEPTION

K (OfficeClerk.StockRequest(_,_) / TRUE)

K(StockClerk.RemoveFromStock(_,_)/ TRUE)

K(StockClerk.AddToStock(_,_)/ TRUE)

ACTION INFORMATION

K(AcceptOrder(_).OfficeClerk / TRUE)

K(RejectOrder(_).OfficeClerk / TRUE)

OFFICECLERK

DECLARATIVE CONSTRAINTS

ACTION COMPOSITION

ProcessOrder ↔ C.Order(o,vi) <> Continue(o,vi,C)

Continue(o,vi,C) ↔ (AlarmCustomer(o,C) ⊕ (
StockRequest(Item(o)) *<>*
*((**IS.AcceptOrder(i)*** *<> ProcessPayment(o,vi)) ⊕*

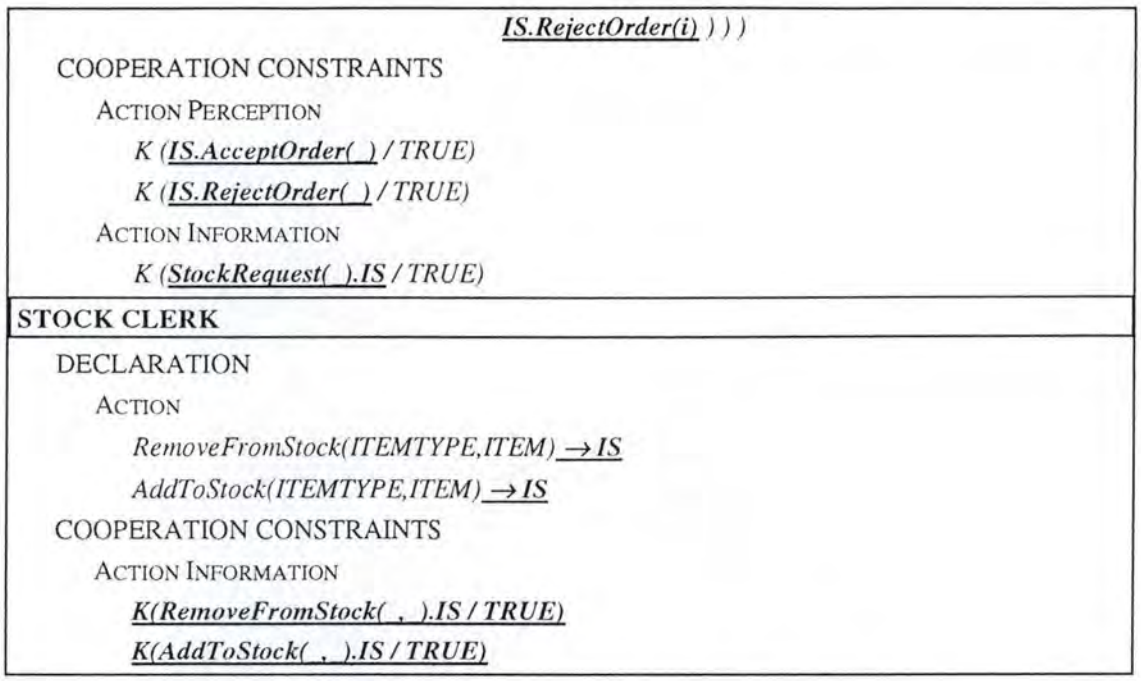


Figure 43: The ALBERT IS specification

Figure 43 shows some excerpts of the ALBERT specification. We first identify the agent '*IS*' representing the information system. The state component '*Inv*' represents the mirror of the real stock. It has the same specification as the real stock except that it doesn't deal with the real items but with integers representing the cardinality of each stock. The 'decision' of the system, represented by the two preconditions and the action composition in the IS-template, is now made upon the value of that virtual stock.

The *Office Clerk* after having verified the order now asks the information system for the availability of the item (see Action Decomposition and Action Information headings of the Office Clerk in figure 43). The payment processing continues when the information system has accepted the request.

The *Stock Clerk* now also updates the computerised stock when he adds or removes items. We modelled this by a visibility of the 'RemoveFromStock' and 'AddToStock' actions towards the information system and the effect of actions constraints modifying the state of the virtual stock 'Inv'.

3.1.2 The Stock Clerk Solution

i^* modelling

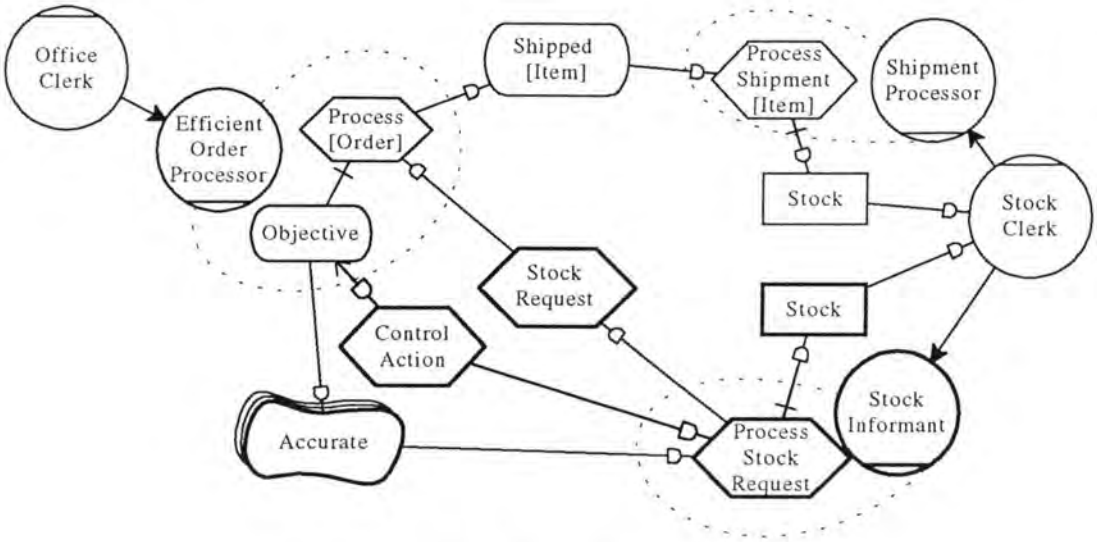


Figure 44: The Stock Clerk Solution

In the i^* model the role of informing about the availability for the item is now played by the agent '*Stock Clerk*'. We simply connect the previously identified role to this agent. The Stock Clerk does not have to maintain a mirror of the real stock because he has direct access on it. We modelled this by a resource dependency 'Stock' towards the Stock Clerk.

ALBERT modelling

In the ALBERT specification, the agent stock clerk plays two roles previously identified by the i^* model. This is modelled by the additional composed action '*ProcessStockRequest*': the Office Clerk now requests the Stock Clerk and he waits until he receives a response from the Stock Clerk.

STOCKCLERK

BASIC CONSTRAINTS

INITIAL VALUATION

Stock[_] = {}

DECLARATIVE CONSTRAINTS

ACTION COMPOSITION

ProcessStockRequest(i) $\leftrightarrow$ ***OfficeClerk.StockRequest(i)*** <>
(***AcceptOrder(i)*** $\oplus$ ***RejectOrder(i)***)

ProcessOrderShipment $\leftrightarrow$
OfficeClerk.OrderShipment(inv) <>(
(RemoveFromOnHold(Item(inv), it) <> *Ship(inv, it))*
$\oplus$ *DAC)*

OPERATIONAL CONSTRAINTS

PRECONDITION

AcceptOrder(i) : Card (Stock[i]) > 0

RejectOrder(i) : Card (Stock[i]) ≤ 0

RemoveFromStock(i,_) : Card (Stock[i]) > 0

EFFECTS OF ACTIONS

AddToStock(i,it) : *[] Stock[i] := (it ∪ Stock[i])*

RemoveFromStock(i,it) : *[] Stock[i] := (it \ Stock[i])*

COOPERATION CONSTRAINTS

ACTION PERCEPTION

K (OfficeClerk.OrderShipment(_) / TRUE)

K (OfficeClerk.StockRequest(_) /TRUE)

ACTION INFORMATION

K(AcceptOrder(_).OfficeClerk / TRUE)

K(RejectOrder(_).OfficeClerk / TRUE)

3.2 Revision of Softgoals

When the present solution was shown to the mail order company, it turned out that the accurate softgoal had been misunderstood. Indeed, the process we modelled did not consider the fact that items that were still in the stock but that were already reserved for another order cannot be counted as available items. The consequence is that there are still a few undeliverable orders for which a payment processing occurs. This leads to a revision of the '*Accurate Info*' softgoal which is represented in figure 45.

We added a new softgoal '*Keep track of OnHolds*', which introduces the state component '*OnHold*' in the specification and its respective modification actions.

The meaning of the state component 'Inv' is redefined: it reflects now only the items that are still in the stock and not reserved for a preceding order.

The state component reflects the items that are in the stock and reserved by a preceding order request.

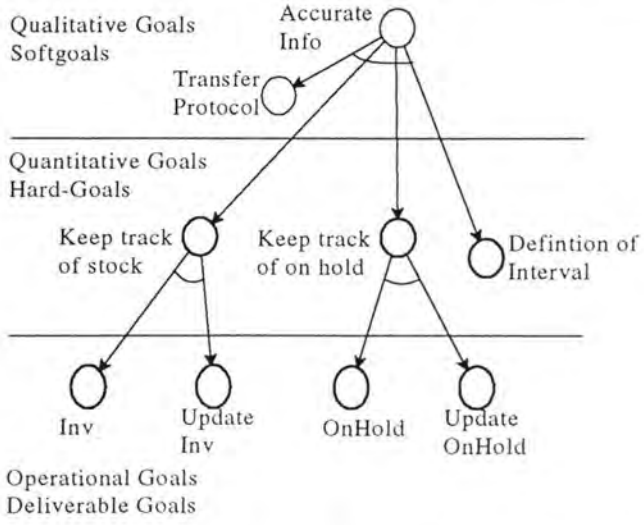


Figure 45: Revision of the 'Accurate Info' *Softgoal*

3.2.1 The IS Solution

*i** modelling

Additionally to the usual stock information the information system now also have to keep track of the items that were reserved but not yet shipped. Note that we modelled this by one resource 'Inv/OnHolds' for the sake of simplicity and the task 'ProcessStockRequest' might be decomposed into two resources. In addition, modelling the updating of the inventory and the 'OnHold' by two dependency links might be more adequate.

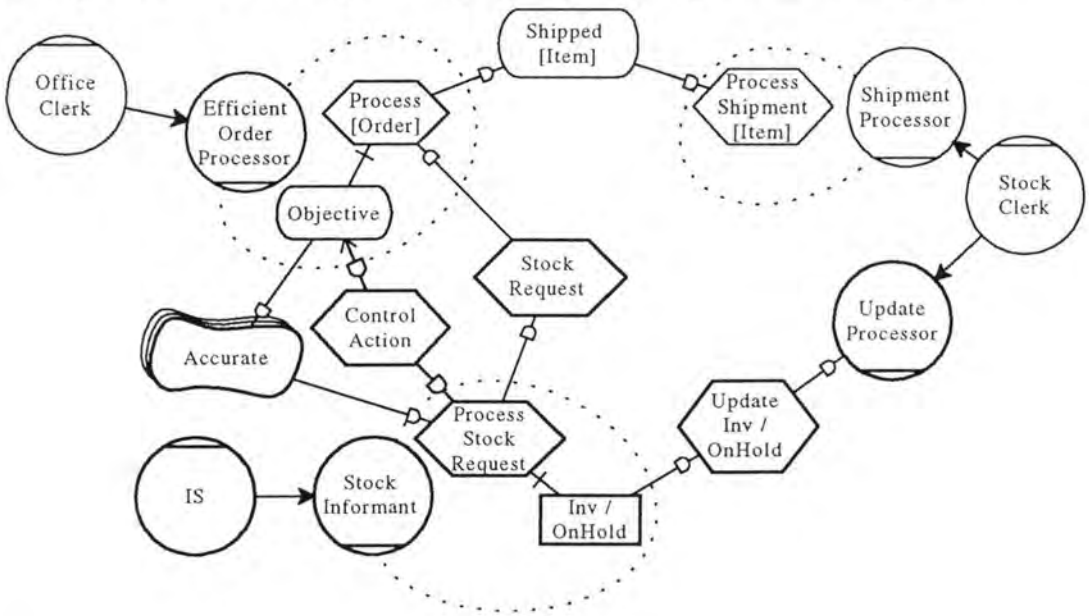


Figure 46: The improved IS Solution

ALBERT modelling.

In the ALBERT specification, we introduce a state component which reflects the history of the stock, i.e. the items already reserved for orders and not yet shipped away: *'OnHold'*. When an order is accepted by the Information System the item is immediately put on hold and consequently the items that are counted for the following request are the items still available. The decision whether the request is accepted or rejected is made upon the number of items in the stock and not reserved by another order, i.e. 'Inv'.

IS

DECLARATIONS

STATE COMPONENTS

Inv table-of *INTEGER* indexed-by *ITEMTYPE*

OnHold table-of *INTEGER* indexed-by *ITEMTYPE*

BASIC CONSTRAINTS

INITIAL VALUES

Inv[_] = 0

OnHold[_] = 0

DECLARATIVE CONSTRAINTS

ACTION COMPOSITION

{ OfficeClerk.StockRequest, AcceptOrder, RejectOrder}

ProcessStockRequest(i) $\leftrightarrow$ *OfficeClerk.StockRequest(i)* <> *(AcceptOrder(i)* $\oplus$ *RejectOrder(i))*

OPERATIONAL CONSTRAINTS

PRECONDITIONS

AcceptOrder(i) : ***(Inv[i] - 1) > 0***

RejectOrder(i) : ***(Inv[i] - 1)*** $\leq$ ***0***

EFFECTS OF ACTIONS

AcceptOrder(i) : []

Inv[i] := Inv[i] - 1

OnHold[i] := OnHold[i] + 1

3.2.2 The Stock Clerk Solution

The solution described previously was a typical 'computer' solution. This does not necessarily have to be the case. The alternative we describe in this section presents a manual solution for our specified problem. The Stock Clerk assumes the new role of answering to the request.

i Modelling*

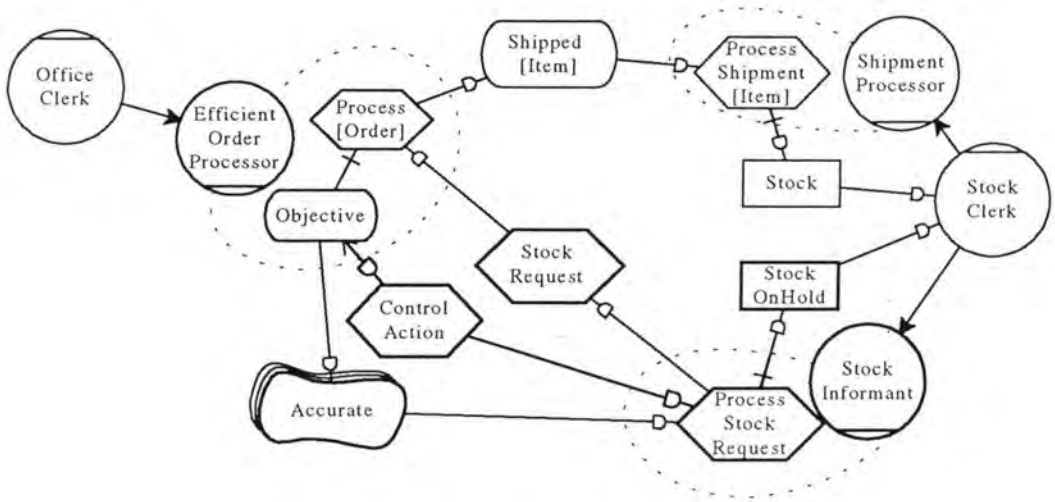


Figure 47: The Stock Clerk Solution

The stock informant now also has to maintain information about the items that are reserved for other orders ('Stock/OnHold').

ALBERT Modelling

In the ALBERT specification, the office clerk requests the stock clerk for the availability of the item that are not reserved. The Stock Clerk now puts the item in an intermediate stock when a request is being accepted.

STOCKCLERK

BASIC CONSTRAINTS

INITIAL VALUATION

Stock[_] = {}

OnHold[_] = {}

DECLARATIVE CONSTRAINTS

ACTION COMPOSITION

ProcessStockRequest(i) $\leftrightarrow$ ***OfficeClerk.StockRequest(i)*** <>
(***AcceptOrder(i)*** $\oplus$ ***RejectOrder(i)***)

ProcessOrderShipment $\leftrightarrow$
OfficeClerk.OrderShipment(inv) <>(
(RemoveFromOnHold(Item(inv), it) <> *Ship(inv, it))*
$\oplus$ *DAC)*

OPERATIONAL CONSTRAINTS

PRECONDITION

AcceptOrder(i) : Card (Stock[i]) > 0

RejectOrder(i) : Card (Stock[i]) ≤ 0

RemoveFromStock(i,_) : Card (Stock[i]) > 0

EFFECTS OF ACTIONS

AddToStock(i,it) : *[] Stock[i] := (it ∪ Stock[i])*

RemoveFromStock(i,it) : *[] Stock[i] := (it \ Stock[i])*

AcceptOrder(i): *[] OnHold[i] := (it ∪ Onhold[i])*

Stock[i] := (it \ Stock[i])

COOPERATION CONSTRAINTS

ACTION PERCEPTION

K (OfficeClerk.OrderShipment(_) / TRUE)

K (OfficeClerk.StockRequest(_) /TRUE)

ACTION INFORMATION

K(AcceptOrder(_).OfficeClerk / TRUE)

K(RejectOrder(_).OfficeClerk / TRUE)

CHAPTER 6
CONCLUSION

The main objective of the thesis was to propose a methodology for the elaboration of a requirements document. Although the word methodology in its first degree means the study of methods, we use it here in the way it is generally used in Software Engineering: 'the way of performing a task'. We used the words 'method' and 'methodology' hence indistinguishably. In general, every method/methodology is defined by a set of languages, a sequence of deliverables, a sequence of tasks and heuristics.

The two languages the methodology is based upon are *i** for representing the non-functional/organisational issues and ALBERT for the system specification.

We divided this process into three main steps:

- the description of the problem domain and the objectives, derived from the existing problems,
- the introduction of a system that solves the problems,
- the specification of the system's internals.

The thesis also showed how the system analyst/developer can deal with - a priori - non functional requirements and how they can be introduced during the requirements engineering process. Goals increase in importance when we relate them to their rationales and 'owners'. This was realised by elaborating the Strategic Dependency and Strategic Rationale model.

Instead of giving a highly detailed description of the approach, we only described some coarse-grained tasks that were loosely ordered. Indeed, we believe that it is more important that the user of a method also knows why he performs the tasks than performing them in an automatic way. Furthermore, too detailed methodologies often influence the way of thinking in a 'negative' way because they restrict the set of solutions. According to this view, we insisted more on the rationales of the different phases a requirements specification should go through. We also noted existing problems about the process (see for instance the interacting Omni-System).

Likewise, we showed different alternatives for representing the system in ALBERT. The implicit representation of the system in ALBERT (Chapter 4, section 2.1: alt 2) seems to be impossible if the method is applied only with the ALBERT framework. But the introduction of the *i** model inverses the situation. A detailed analysis of the two representations, identifying their pros and cons, could be an interesting future work.

In our description of the process, we also made an important assumption: that we are not in presence of an existing (legacy) system. In reality, although, this often appears to be false. In our case study, a solution for the problem might already exist; the manual solution for instance. Indeed, in everyday situations, systems often are created without an explicit elaboration of alternatives or specifications. What now if we detect a legacy system? Two alternatives might be imaginable: (1) we model the system as part of the problem domain and add the system on top of this environment or (2) we first try to retract the existing system that did not satisfy completely the environment and model the problem environment without the legacy system. The first solution seems to be more adequate from a conceptual point of view: the incompleteness of the system partially or completely caused the problem to be solved. Consequently, it belongs to the problem environment. The second solution, however, does not suppose the existing system as a constraint and might result in a better solution. Again, these possibilities should be analysed furthermore.

In comparison to other methods that exist since the 80's (see JSD [Jackson, 83] for instance), this methodology has not yet been certified by a lot of non-trivial case studies. We hope, however, that, because of its profound relation to JSD, it inherits some empirical aspects of the latter method.

Finally, in order to assist the system analyst/developer in their work a CASE tool should be elaborated for the parallel use of *i** and ALBERT. Several properties are conceivable:

- Tools are already in course of being elaborated for the individual use of the two models. The parallel modelling in *i** and in ALBERT can increase the complexity and the workload significantly. The tool should try to reduce these two aspects.
- Correspondingly, the tool should keep trace of two types of evolutions:
    - The 'space' evolution, that relates the *i** concepts to their counterpart in the ALBERT specification.
    - The 'time' evolution, that relates the different models in time.
- The tool should moreover assist the analyst/developer by highlighting inconsistencies or incompletenesses between the models or by proposing 'solutions' based upon previously gained domain knowledge.

BIBLIOGRAPHY

[Boehm,79]: B. Boehm, *Guidelines for Verifiying and Validating Software Requirements and Design Specifications,* Samet P.A.: IFIP 1979, North-Holland, Amsterdam. 711-719

[Booch,86]] G. Booch, *Object-Oriented Development,* IEEE Transactions on Software Engineering, vol. SE-12, no 2, February 1986

[Borgida et al,92] 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

[Chung et al,94]] L. Chung, B. Nixon and E. Yu, *Using quality requirements to systematically develop quality software,* Fourth International Conference on Software Quality, McLean. VA. USA, October 1994

[DuBois,95]] Ph. Du Bois, *The ALBERT II Language - On the Design and the Use of a Formal Specification Language for requirements Analysis,* PhD Thesis, Computer Science Department, University of Namur, 1995

[Dubois et al,95]] E. Dubois, Ph. Du Bois, J-M Zeippen, *A Formal Requirements Engineering Method for Real-Time, Concurrent, and Distributed Systems,* Computer Science Department, University of Namur, January 1995

[DuBois,97]] Ph. Du Bois, *The ALBERT II Reference Manual - Version 2.0,* Computer Science Department, University of Namur, March 1997

[Dardenne et al,93] A. Dardenne, A. van Lamsweerde, S. Fickas, *Goal-Directed Requirements Acquisition*, Science of Computer Programming 20, 1993, pp3-50

[Gause et al, 89] D. Gause and G. Weinberg, *Exploring Requirements: Quality Before Design,* Dorset House Publishing, New York, 1989

[IEEE,91] IEEE—610.12, *IEEE Standard Glossary of Software Terminology,* 1991

[Jackson,83] M. Jackson. *System Development*, Prentice-Hall, 1983

[Jackson et al,96] M. Jackson, Pamela Zave, *Four Dark Corners of Requirements Engineering*, February 1996

[MDL,97] E. Dubois, *Méthodologie de développement de logiciels: matières approfondies,* Computer Science Department, University of Namur, 1997

[Mintzberg et al,91] H. Mintzberg, A. Langley, P. Pitcher, E. Posada, J Saint-Macary *Opening Up Decision-Making: The View From The Black Stool*, September 1991

[Porter et al,85] M.E. Porter, V.E. Millar. *How Information Gives You Competitive Advantage.* Harvard Business Review, Vol. 63, No 4, July-August 1985, pp.149-160

[Pohl,92] Pohl, K. *The Three Dimensions of Requirements Engineering,* NATURE Report Series, Informatik V, RWTH-Aachen, URL : ftp://ftp.informatik.rwth-aachen.de/pub/reports/index.html (1997)

[Pohl,96] Pohl, K. *Requirements Engineering: An Overview* . NATURE Report Series, Informatik V, RWTH-Aachen, URL : ftp://ftp.informatik.rwth-aachen.de/pub/reports/index.html (1997)

[DeMarco,79] T. DeMarco, *Structured Analysis and System Specification,* Prentice Hall:Englewood Cliffs, 1979

[Scheer,93] A.L. Scheer. *A New Approach to Business Processes,* IBM Systems Journal. vol 32. No 1. 1993

[Wieringa et al,97] R. Wieringa, Eric Dubois, Sander Huyts, *Integrating Semi-Formal and Formal Requirements*, in "CAISE'97", Barcelona (Spain), LNCS 1250, 1997

[Wieringa,97] R. Wieringa, *Steps Towards a Method for the Formal Modelling of Dynamic Objects,* URL : http://www.cs.vu.nl/vakgroepen/infsys/roelw.publ.html February 1997

[Wieringa,97b] R. Wieringa, *Object-Oriented Analysis, Structured Analysis and Jackson System Development,* URL : http://www.cs.vu.nl/vakgroepen/infsys/roelw.publ.html, February 1997

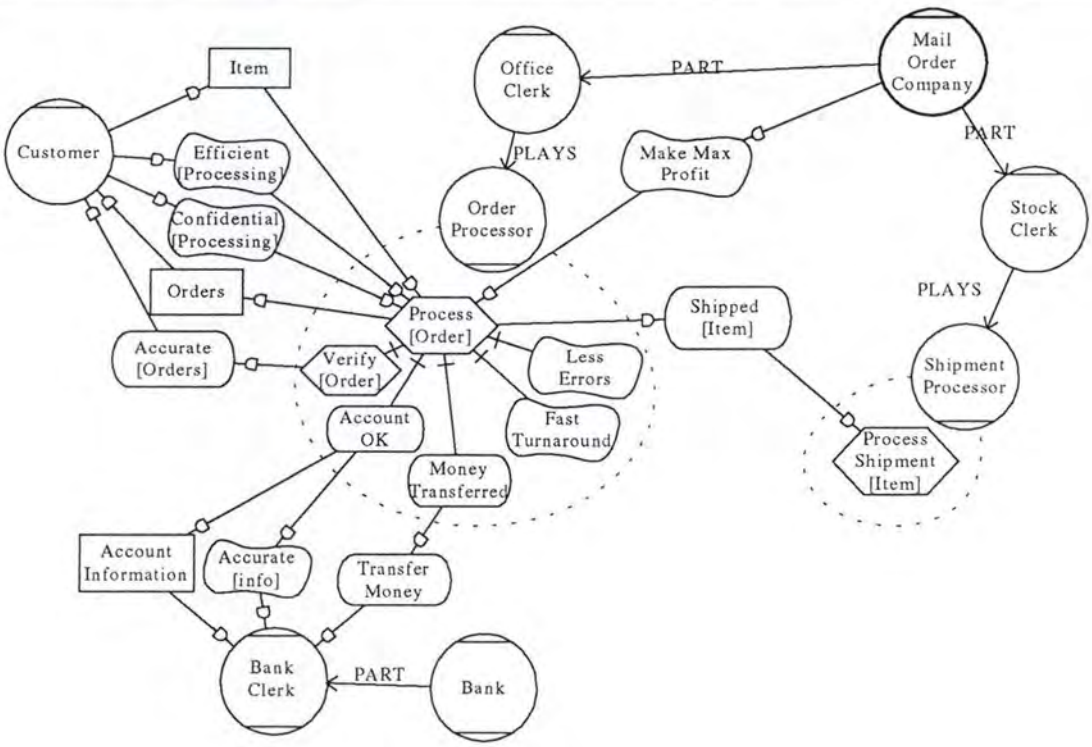
[Yu, 95] E. Yu. *Modelling Strategic Relationships For Process Reengineering.* PhD Thesis, Dept. Of Computer Science, University of Toronto, Ontario Canada,1995

APPENDIX

In the following, we show the different *i** models and ALBERT specifications at the different main steps of the methodology. For the sake of convenience, the state components and actions are not re-declared at every step. Only newly introduced state components and actions are declared. Furthermore we concentrated on the process where the system is introduced.

1. Problem domain and Objective

1.1 The *i** model



1.2 The ALBERT specification

BASIC TYPES

VISA

ITEM

ITEMTYPE

CONSTRUCTED TYPES

ADDRESS = CP[Name:STRING, Street:STRING, Locality: STRING]

ORDER = CP[OrderId:INTEGER, Person: ADDRESS,Item: ITEMTYPE]

INVOICE = CP[InvoiceId: INTEGER, Person: ADDRESS, Item: ITEMTYPE, Sum: INTEGER]

CUSTOMER

DECLARATIONS

STATE COMPONENTS

Possessed $\overset{set\text{-}of}{}$ *ITEM*

ACTIONS

Order(ORDER, VISA) $\rightarrow$ *OfficeClerk*

OPERATIONAL CONSTRAINTS

EFFECTS OF ACTIONS

Mail.Deliver(_,it) : *[]*

Possessed := Possessed $\cup$ *it*

COOPERATION CONSTRAINTS

ACTION INFORMATION

K (Order(_,_).OfficeClerk / TRUE)

ACTION PERCEPTION

K (Mail.Deliver(_,_) / TRUE)

OFFICECLERK

DECLARATIONS

STATE COMPONENTS

Busy $\overset{instance\text{-}of}{}$ *BOOLEAN*

SoldItemTypes $\overset{\text{set-of}}{\longrightarrow}$ *ITEMTYPE*

ACTIONS

ProcessOrder

ProcessPayment(ORDER, VISA)

DebitRequest(INTEGER,VISA) $\rightarrow$ *BankClerk*

TransferOrder(INTEGER,VISA,VISA) $\rightarrow$ *StockClerk*

MakeInvoice(ORDER,INVOICE)

OrderShipment(INVOICE)$\rightarrow$ *StockClerk*

AlarmCustomer(ORDER,CUSTOMER) $\rightarrow$ *Customer*

Continue(ORDER,VISA,CUSTOMER)

BASIC CONSTRAINTS

INITIAL VALUATION

Busy = FALSE

DECLARATIVE CONSTRAINTS

ACTION COMPOSITION

{ Continue, C.Order, AlarmCustomer, ProcessPayment, DebitRequest, BankClerk.AcceptOrder, TransferOrder, OrderShipment, BankClerk.RejectOrder }

ProcessOrder $\leftrightarrow$ *C.Order(o,vi) <> Continue(o,vi,C)*

Continue(o,vi,C) $\leftrightarrow$ *AlarmCustomer(o,C)* $\oplus$ *ProcessPayment(o,vi)*

ProcessPayment(o,vi) $\leftrightarrow$
DebitRequest(am,vi) <> (

BankClerk.RejectOrder(am, vi) $\oplus$

(BankClerk.AcceptOrder(am, vi) <> TransferOrder(am,vi,company_account) <>
OrderShipment(inv)))

OPERATIONAL CONSTRAINTS

PRECONDITIONS

AlarmCustomer(o,_) : Item(o) $\notin$ *SoldItemTypes*

ProcessPayment(o,_) : Item(o) $\in$ *SoldItemTypes*

EFFECTS OF ACTIONS

Continue(_,_,_) : *Busy := TRUE*

[]

Busy := FALSE

COOPERATION CONSTRAINTS

ACTION PERCEPTION

K (C.Order(_,_) / TRUE)

K (BankClerk.AcceptOrder(_,_) / TRUE)

K (BankClerk.RejectOrder(_,_) / TRUE)

ACTION INFORMATION

K (OrderShipment(_).StockClerk / TRUE)

K (DebitRequest(_,_).BankClerk / TRUE)

K (TransferOrder(_,_,_).BankClerk / TRUE)

XK (AlarmCustomer(_,C^1).C^2 / $C^1 = C^2$)

STOCKCLERK

DECLARATIONS

STATE COMPONENTS

Stock table-of *SET[ITEM]* indexed-by *ITEMTYPE*

ACTIONS

ProcessOrderShipment

Ship (INVOICE,ITEM) → *Mail*

RemoveFromStock(ITEMTYPE,ITEM)

AddToStock(ITEMTYPE,ITEM)

BASIC CONSTRAINTS

INITIAL VALUATION

Stock[_] = {}

DECLARATIVE CONSTRAINTS

ACTION COMPOSITION

{ OfficeClerk.OrderShipment, RemoveFromStock, Ship}

ProcessOrderShipment ↔

OfficeClerk.OrderShipment(inv) <> (

(RemoveFromStock(Item(inv), it) <> *Ship(inv, it))*

⊕ *DAC)*

OPERATIONAL CONSTRAINTS

PRECONDITION

RemoveFromStock(i,_) : Card (Stock[i]) > 0

EFFECTS OF ACTIONS

AddToStock(i,it) : *[] Stock[i] := (it ∪ Stock[i])*

RemoveFromStock(i,it) : *[] Stock[i] := (it \ Stock[i])*

COOPERATION CONSTRAINTS

ACTION PERCEPTION

K (OfficeClerk.OrderShipment(_) / TRUE)

ACTION INFORMATION

K (Ship(_,_).Mail / TRUE)

BANKCLERK

DECLARATIONS

STATE COMPONENTS

Account table-of *INTEGER* indexed-by *VISA*

ACTIONS

ProcessDebitRequest

AcceptOrder(INTEGER,VISA) → OfficeClerk

RejectOrder(INTEGER,VISA) → OfficeClerk

Transfer

Debit(INTEGER, VISA)

Credit(INTEGER, VISA)

BASIC CONSTRAINTS

INITIAL VALUES

Account[_] = 0

DECLARATIVE CONSTRAINTS

STATE BEHAVIOR

Account[_] ≥ -50.000

ACTION COMPOSITION

{ OfficeClerk.DebitRequest, AcceptOrder, RejectOrder, OfficeClerk.TransferOrder, Debit, Credit}

ProcessDebitRequest ↔ OfficeClerk.DebitRequest(am,vi) <>

(AcceptOrder(am, vi) ⊕ RejectOrder(am,vi))

Transfer ↔ OfficeClerk.TransferOrder(am,vi,company_account) <> Debit(am, vi) <>
Credit(am, company_account)

OPERATIONAL CONSTRAINTS

PRECONDITIONS

AcceptOrder(am,vi) : (Account[vi] - am) > -50.000

RejectOrder(am,vi) : (Account[vi] - am) ≤ -50.000

EFFECTS OF ACTIONS

Debit(am,ac):

[]

Account[ac] := Account[ac] - am

Credit(am,ac):

[]

Account[ac] := Account[ac] + am

COOPERATION CONSTRAINTS

ACTION PERCEPTION

K (OfficeClerk.DebitRequest(_,_) / TRUE)

K (OfficeClerk.TransferOrder(_,_,_) / TRUE)

ACTION INFORMATION

K (AcceptOrder(_,_).OfficeClerk / TRUE)

K (RejectOrder(_,_).OfficeClerk / TRUE)

MAIL

DECLARATIONS

ACTIONS

Deliver(INVOICE,ITEM) → Customer

TransportProcess

DECLARATIVE CONSTRAINTS

ACTION COMPOSITION

TransportProcess ↔ StockClerk.Ship(inv, it) <> Deliver(inv,it)

COOPERATION CONSTRAINTS

ACTION PERCEPTION

K (StockClerk.Ship(_,_).Mail / TRUE)

ACTION INFORMATION

K (Deliver (_,_).C / TRUE)

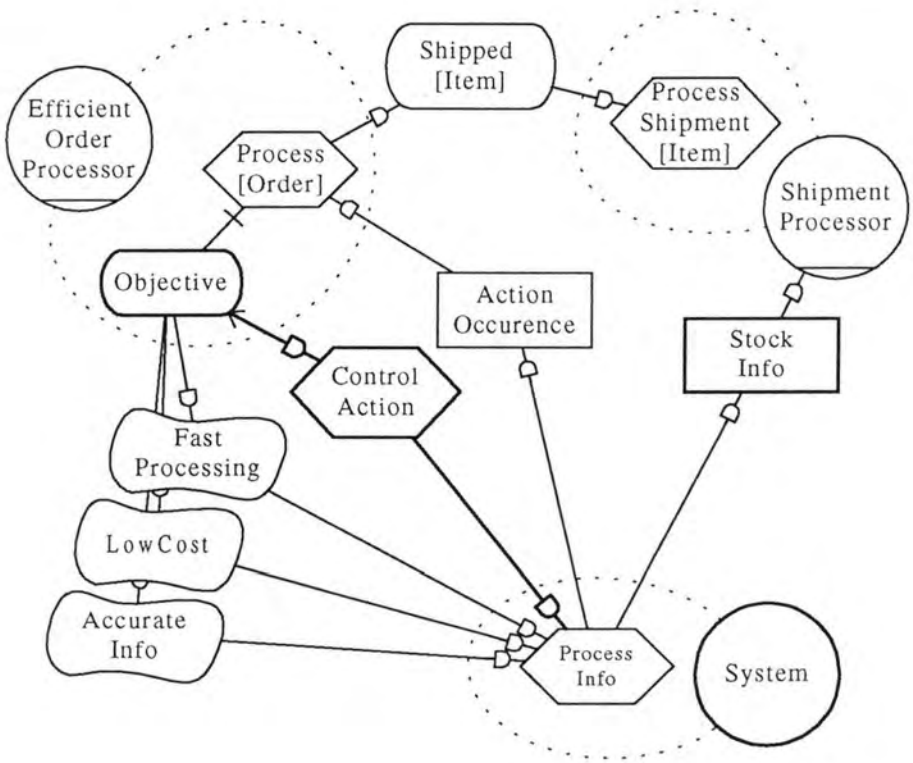
Problem: the order processing clerk makes out the invoice and orders the shipment even if the items are not in the stock. This leads to a lot of inefficiency (paperwork for items that finally cannot be shipped) and annoyance on the customer's side. Therefore we consider the following objective...

Objective (in 'natural' language):
The Order Processing Clerk can only check out the customer's account, make the invoice, and order the shipment
if the respective item is available (i.e. Card(Stock(Item)) > 0)

Formulation of the objective:
F(ProcessPayment(o,_) / Card(StockClerk.Stock(Item(o))) > 0):
the order processing clerk can only proceed when the stock of the requested item is not empty (otherwise the processing is cancelled)

2. System Requirement

2.1 The *i** model



2.2 The ALBERT specification

SYSTEM

DECLARATIONS of the new states and actions...

ACTIONS

ProcessStockRequest

AcceptOrder(ORDER) → *Efficient_Order_Processor*

RejectOrder(ORDER) → *Efficient_Order_Processor*

ACTION COMPOSITION

{ ***AcceptOrder***, ***RejectOrder***, ***Efficient_Order_Processor.StockRequest*** *}*

ProcessStockRequest ↔ ***Efficient_Order_Processor.StockRequest(o)*** <>
(***AcceptOrder(Item(o))*** ⊕ ***RejectOrder(Item(o))***)

PRECONDITION

AcceptOrder(i) : ¬ *(Card(StockClerk.Stock[i]) ≤ 0))*

RejectOrder(i) : (Card(StockClerk.Stock[i]) ≤ 0))

STATE PERCEPTION

K (StockClerk.Stock[_]/ TRUE)

```
the system always knows the state of the stock
```

ACTION PERCEPTION

K (Efficient_Order_Processor.StockRequest(_) / TRUE)

ACTION INFORMATION

K (AcceptOrder(_).Efficient_Order_Processor / TRUE)

K (RejectOrder(_).Efficient_Order_Processor / TRUE)

EFFICIENT_ORDER_PROCESSOR

DECLARATIONS of the new states and actions...

ACTIONS

StockRequest(ORDER) → System

BASIC CONSTRAINTS

INITIAL VALUATION

Busy = FALSE

DECLARATIVE CONSTRAINTS

ACTION COMPOSITION

{ C.Order, Continue, AlarmCustomer, ProcessPayment, DebitRequest, BankClerk.AcceptOrder, TransferOrder, OrderShipment.StockClerk, BankClerk.RejectOrder, Continue, ***StockRequest, System.AcceptOrder, System.RejectOrder*** *}*

```
Upon receiving an order from the customer, the Order Processing Clerk starts
with processing an order. i.e. he first analyses the order, requests the
availability of the item and then processes the payment or cancels the
process.
```

ProcessOrder ↔ *C.Order(o,vi)* <> *Continue(o,vi,C)*

Continue(o,vi,C) ↔ *(AlarmCustomer(o,C)* ⊕ *(* ***StockRequest(o)*** <> *(*

*(**System.AcceptOrder(o)** <> ProcessPayment(o,vi))*

*⊕ **System.RejectOrder(o)**)))*

ProcessPayment(o,vi) ↔
DebitRequest(am,vi) <>(

BankClerk.RejectOrder (am,vi)

⊕ (BankClerk.AcceptOrder (am,vi) <> TransferOrder(am,vi,company_account) <>
OrderShipment(inv)))

OPERATIONAL CONSTRAINTS

PRECONDITIONS

AlarmCustomer(o,_) : Item(o) ∉ SoldItemTypes

StockRequest (o) : Item(o) ∈ SoldItemTypes

EFFECTS OF ACTIONS

Continue(_,_,_) : *Busy := TRUE*
[]
Busy := FALSE

COOPERATION CONSTRAINTS

ACTION PERCEPTION

K (C.Order(_,_) / TRUE)

K (System.AcceptOrder(_) / TRUE)

K (System.RejectOrder(_) / TRUE)

K (BankClerk.AcceptOrder(_,_) / TRUE)

K (BankClerk.RejectOrder(_,_) / TRUE)

ACTION INFORMATION

K (StockRequest(_).System / TRUE)

K (DebitRequest(_,_).BankClerk / TRUE)

K (TransferOrder(_,_,_).BankClerk / TRUE)

K (OrderShipment(_).StockClerk / TRUE)

XK (AlarmCustomer(_,C^1).C^2 / $C^1 = C^2$)

STOCKCLERK

DECLARATIONS

STATE COMPONENTS

Stock table-of *SET[ITEM]* indexed-by *ITEMTYPE* → ***System***

ACTIONS

ProcessOrderShipment

RemoveFromStock(ITEMTYPE, ITEM)

AddToStock(ITEMTYPE, ITEM)

Ship (INVOICE,ITEM) → *Mail*

BASIC CONSTRAINTS

INITIAL VALUATION

Stock[_] = {}

DECLARATIVE CONSTRAINTS

ACTION COMPOSITION

{ Efficient_Order_Processor.OrderShipment, RemoveFromStock, Ship}

ProcessOrderShipment ↔

Efficient_Order_Processor.OrderShipment(inv) <> (
(RemoveFromStock(Item(inv), it) <> *Ship(inv, it))* ⊕ *DAC)*

OPERATIONAL CONSTRAINTS

PRECONDITION

RemoveFromStock(i,_) : Card (Stock[i]) > 0

EFFECTS OF ACTIONS

AddToStock(i,it) : *[] Stock[i] := (it ∪ Stock[i])*

RemoveFromStock(i,it) : *[] Stock[i] := (it \ Stock[i])*

COOPERATION CONSTRAINTS

ACTION PERCEPTION

K (Efficient_Order_Processor.OrderShipment(_) / TRUE)

STATE PERCEPTION

K (Stock[_].System/ TRUE)

ACTION INFORMATION

K (Ship(_,_).Mail / TRUE)

BANKCLERK

nothing has changed for this agent

MAIL

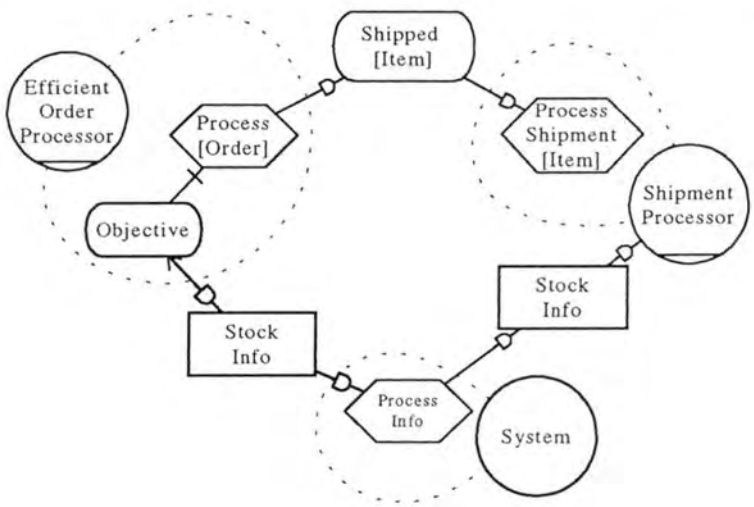
nothing has changed for this agent

CUSTOMER

nothing has changed for this agent

2.3 System Requirement (resource solution)

The system only shows the interesting information about the stock and the Efficient_Order_Processor decides what to do when the stock runs out items. This is represented by the Precondition in the Office-Clerk's template.



SYSTEM

DECLARATION

STATE COMPONENTS

Inv table-of *INTEGER* indexed-by *ITEMTYPE* → *Efficient_Order_Processor*

ACTIONS

**Act*

DECLARATIVE CONSTRAINTS

STATE BEHAVIOUR

[ ] Inv[i] = Card(StockClerk.Stock[i])

OPERATIONAL CONSTRAINTS

EFFECTS OF ACTIONS

Act: *[] Inv[i] := alpha*

COOPERATION CONSTRAINTS

STATE PERCEPTION

K (StockClerk.Stock[_]/ TRUE)

STATE INFORMATION

K (Inv[_].Efficient_Order_Processor / TRUE)

EFFICIENT_ORDER_PROCESSOR

BASIC CONSTRAINTS

INITIAL VALUATION

Busy = FALSE

DECLARATIVE CONSTRAINTS

ACTION COMPOSITION

{ C.Order,Continue, AlarmCustomer, ProcessPayment, DebitRequest, BankClerk.AcceptOrder, OrderShipment, BankClerk.RejectOrder, TransferOrder}

ProcessOrder $\leftrightarrow$ *C.Order(o,vi)* <> *Continue(o,vi,C)*

Continue(o,vi,C) $\leftrightarrow$ *AlarmCustomer(o,C)* $\oplus$ *ProcessPayment(o,vi)*

ProcessPayment(o,vi) $\leftrightarrow$
DebitRequest(am,vi) <> (
BankClerk.RejectOrder (am,vi)
$\oplus$ *(BankClerk.AcceptOrder (am,vi)* <> *TransferOrder(am,vi,company_account)* <> *OrderShipment(inv)))*

OPERATIONAL CONSTRAINTS

PRECONDITIONS

AlarmCustomer(o,_) : Item(o) $\notin$ *SoldItemTypes*

ProcessPayment(o,_) : Item(o) $\in$ *SoldItemTypes* $\wedge$ ***System.Inv[Item(o)] > 0***

EFFECTS OF ACTIONS

Continue(_,_,_) : *Busy := TRUE*
[]
Busy := FALSE

COOPERATION CONSTRAINTS

ACTION PERCEPTION

K (C.Order(_,_) / TRUE)

K (BankClerk.AcceptOrder(_,_) / TRUE)

K (BankClerk.RejectOrder(_,_) / TRUE)

STATE PERCEPTION

K (System.Inv[_] / TRUE)

ACTION INFORMATION

K (DebitRequest(_,_).BankClerk / TRUE)

K (TransferOrder(_,_,_).BankClerk / TRUE)

K (OrderShipment(_).StockClerk / TRUE)

$XK\ (AlarmCustomer(\_,C^1).C^2 / C^1 = C^2\ )$

STOCKCLERK

DECLARATIONS

STATE COMPONENTS

Stock $\overset{table\text{-}of}{}$ *SET[ITEM]* $\overset{indexed\text{-}by}{}$ *ITEMTYPE* $\rightarrow$ ***System***

ACTIONS

ProcessOrderShipment

RemoveFromStock(ITEMTYPE, ITEM)

AddToStock(ITEMTYPE, ITEM)

Ship (INVOICE,ITEM) $\rightarrow$ *Mail*

BASIC CONSTRAINTS

INITIAL VALUATION

Stock[_] = {}

DECLARATIVE CONSTRAINTS

ACTION COMPOSITION

{ Efficient_Order_Processor.OrderShipment, RemoveFromStock, Ship}

ProcessOrderShipment $\leftrightarrow$

Efficient_Order_Processor.OrderShipment(inv) <> (

(RemoveFromStock(Item(inv), it) <> Ship(inv, it))

⊕ DAC)

OPERATIONAL CONSTRAINTS

PRECONDITION

RemoveFromStock(i,_) : Card (Stock[i]) > 0

EFFECTS OF ACTIONS

AddToStock(i,it) : *[] Stock[i] := (it ∪ Stock[i])*

RemoveFromStock(i,it) : *[] Stock[i] := (it \ Stock[i])*

COOPERATION CONSTRAINTS

ACTION PERCEPTION

K (Efficient_Order_Processor.OrderShipment(_) / TRUE)

STATE INFORMATION

K (Stock[_].System/ TRUE)

ACTION INFORMATION

K (Ship(_,_).Mail / TRUE)

BANKCLERK

nothing has changed for this agent

MAIL

nothing has changed for this agent

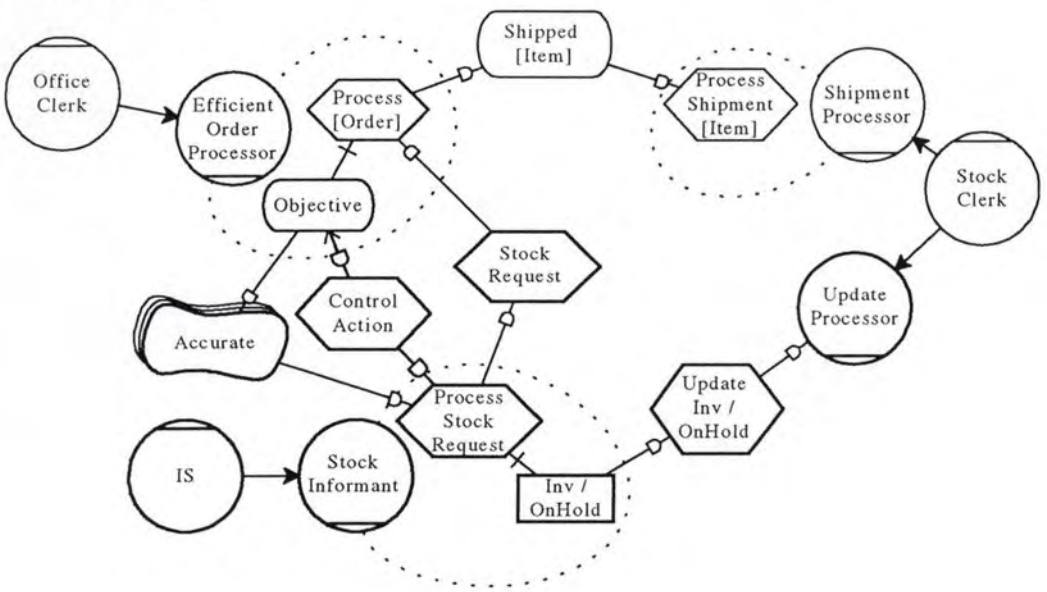
CUSTOMER

nothing has changed for this agent

3. System Specification

3.1 The IS solution

3.1.1 The *i** model



3.1.2 The ALBERT specification

IS

DECLARATIONS

STATE COMPONENTS

Inv table-of *INTEGER* indexed-by *ITEMTYPE*

OnHold table-of *INTEGER* indexed-by *ITEMTYPE*

ACTIONS

ProcessStockRequest(ITEMTYPE)

AcceptOrder(ITEMTYPE) → *OfficeClerk*

RejectOrder(ITEMTYPE) → *OfficeClerk*

BASIC CONSTRAINTS

INITIAL VALUES

Inv[_] = 0

OnHold[_] = 0

DECLARATIVE CONSTRAINTS

ACTION COMPOSITION

{ OfficeClerk.StockRequest, AcceptOrder, RejectOrder}

ProcessStockRequest(i) ↔ OfficeClerk.StockRequest(i) <>
(AcceptOrder(i) ⊕
RejectOrder(i))

OPERATIONAL CONSTRAINTS

PRECONDITIONS

AcceptOrder(i) : ***(Inv[i] - 1) > 0***

RejectOrder(i) : ***(Inv[i] - 1) ≤ 0***

EFFECTS OF ACTIONS

AcceptOrder(i) : []

Inv[i] := Inv[i] - 1

OnHold[i] := OnHold[i] + 1

OfficeClerk.NotifyCancel(i) : []

OnHold[i] := OnHold[i] - 1

Inv[i] := Inv[i] + 1

StockClerk.RemoveFromStock(i,_): []

OnHold[i] := OnHold[i] - 1

StockClerk.AddToStock(i,_): []

Inv[i] := Inv[i] - 1

COOPERATION CONSTRAINTS

ACTION PERCEPTION

K (OfficeClerk.StockRequest(_,_) / TRUE)

K(OfficeClerk.NotifyCancel(_) / TRUE)

K(StockClerk.RemoveFromStock(_,_)/ TRUE)

K(StockClerk.AddToStock(_,_)/ TRUE)

ACTION INFORMATION

K(AcceptOrder(_).OfficeClerk / TRUE)

K(RejectOrder(_).OfficeClerk / TRUE)

OFFICECLERK

DECLARATIONS of the new states and actions...

ACTIONS

StockRequest(ITEMTYPE) → *IS*

```
NotifyCancel(i).IS: the cancelling of a process for the item 'i' is notified
towards the information system ACCURACY INTEREST
```

NotifyCancel*(ITEMTYPE)* → *IS*

BASIC CONSTRAINTS

INITIAL VALUATION

Busy = FALSE

DECLARATIVE CONSTRAINTS

ACTION COMPOSITION

{ C.Order, Continue, AlarmCustomer, ***StockRequest****,* ***IS.AcceptOrder****, ProcessPayment,* ***IS.RejectOrder****, DebitRequest, BankClerk.RejectOrder,* ***NotifyCancel****, BankClerk.AcceptOrder, TransferOrder, OrderShipment}*

ProcessOrder ↔ *C.Order(o,vi)* <> *Continue(o,vi,C)*

Continue(o,vi,C) ↔ *(AlarmCustomer(o,C)* ⊕ *(* ***StockRequest(Item(o))*** <> *(*

*(****IS.AcceptOrder(i)*** <> *ProcessPayment(o,vi))*

⊕ ***IS.RejectOrder(i)*** *))*

ProcessPayment(o,vi) ↔

DebitRequest(am,vi) <> *(*

(BankClerk.RejectOrder (am,vi) <> ***NotifyCancel(Item(o))****)* ⊕

(BankClerk.AcceptOrder (am,vi) <> *TransferOrder(am,vi,company_account)* <>

OrderShipment(inv)))

OPERATIONAL CONSTRAINTS

PRECONDITIONS

AlarmCustomer(o,_) : Item(o) ∉ SoldItemTypes

StockRequest.Sys(o) : Item(o) ∈ SoldItemTypes

EFFECTS OF ACTIONS

Continue(_,_,_) : *Busy := TRUE*

[]

Busy := FALSE

COOPERATION CONSTRAINTS

ACTION PERCEPTION

K (C.Order(_,_) / TRUE)

*K (**IS.AcceptOrder(_)** / TRUE)*

*K (**IS.RejectOrder(_)** / TRUE)*

K (BankClerk.AcceptOrder(_,_) / TRUE)

K (BankClerk.RejectOrder(_,_) / TRUE)

ACTION INFORMATION

*K (**StockRequest(_).IS** / TRUE)*

K (DebitRequest(_,_).BankClerk / TRUE)

K (TransferOrder(_,_,_).BankClerk / TRUE)

K (OrderShipment(_).StockClerk / TRUE)

*K (**NotifyCancel(_).IS** / TRUE)*

XK (AlarmCustomer(_,C^1).C^2 / $C^1 = C^2$)

STOCK CLERK

DECLARATIONS NEW...

ACTIONS

RemoveFromStock(ITEMTYPE,ITEM) → IS

AddToStock(ITEMTYPE,ITEM) → IS

BASIC CONSTRAINTS

INITIAL VALUATION

Stock[_] = {}

DECLARATIVE CONSTRAINTS

ACTION COMPOSITION

{OfficeClerk.OrderShipment, RemoveFromStock, Ship}

ProcessOrderShipment $\leftrightarrow$ *OfficeClerk.OrderShipment(inv)* < > (

(RemoveFromStock(Item(inv), it) <> *Ship(inv, it))*

$\oplus$ *DAC)*

OPERATIONAL CONSTRAINTS

PRECONDITION

RemoveFromStock(i,_) : Card (Stock[i]) > 0

EFFECTS OF ACTIONS

AddToStock(i,it) : *[] Stock[i] := (it* $\cup$ *Stock[i])*

RemoveFromStock(i,it) : *[] Stock[i] := (it* $\setminus$ *Stock[i])*

COOPERATION CONSTRAINTS

ACTION PERCEPTION

K (OfficeClerk.OrderShipment(_) / TRUE)

ACTION INFORMATION

K(RemoveFromStock(,).IS / TRUE)

K(AddToStock(,).IS / TRUE)

BANKCLERK

nothing has changed for this agent

MAIL

nothing has changed for this agent

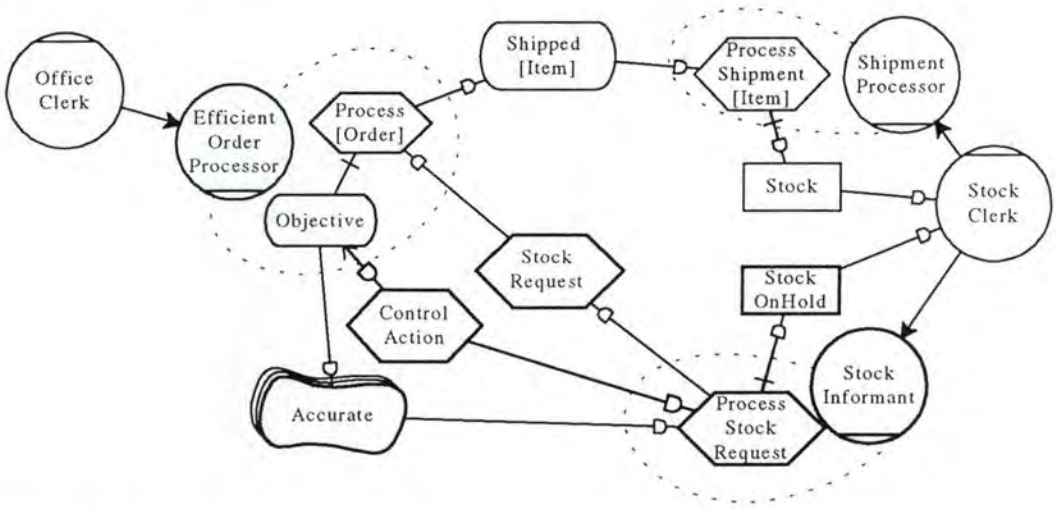
CUSTOMER

nothing has changed for this agent

3.2 The Stock Clerk Solution

The Stock clerk assumes the role of informing the office clerk about the stock.

3.2.1 The *i** model



3.2.2 The ALBERT specification

STOCKCLERK

BASIC CONSTRAINTS

INITIAL VALUATION

Stock[_] = {}

OnHold[_] ={}

DECLARATIVE CONSTRAINTS

ACTION COMPOSITION

*{ **OfficeClerk.StockRequest**, **AcceptOrder**, **RejectOrder**, OfficeClerk.OrderShipment, RemoveFromOnHold, Ship}*

*ProcessStockRequest(i) ↔ **OfficeClerk.StockRequest(i)** <>*
*(**AcceptOrder(i)** ⊕ **RejectOrder(i)**)*

ProcessOrderShipment ↔
OfficeClerk.OrderShipment(inv) <>(
(RemoveFromOnHold(Item(inv), it) <> Ship(inv, it))
⊕ DAC)

OPERATIONAL CONSTRAINTS

PRECONDITION

AcceptOrder(i) : Card (Stock[i]) > 0

RejectOrder(i) : Card (Stock[i]) ≤ 0

RemoveFromOnHold(i,_) : Card (OnHold[i]) > 0

EFFECTS OF ACTIONS

AddToStock(i,it) : *[] Stock[i] := (it ∪ Stock[i])*

RemoveFromOnHold (i,it) : *[] OnHold[i] := (it \ OnHold[i])*

AcceptOrder(i): *[] OnHold[i] := (it ∪ Onhold[i])*

Stock[i] := (it \ Stock[i])

COOPERATION CONSTRAINTS

ACTION PERCEPTION

K (OfficeClerk.OrderShipment(_) / TRUE)

K (OfficeClerk.StockRequest(_) /TRUE)

ACTION INFORMATION

K(AcceptOrder(_).OfficeClerk / TRUE)

K(RejectOrder(_).OfficeClerk / TRUE)

OFFICECLERK

BASIC CONSTRAINTS

INITIAL VALUATION

Busy = FALSE

DECLARATIVE CONSTRAINTS

ACTION COMPOSITION

*{..., **StockRequest(Item(o)), StockClerk.AcceptOrder(Item(o)), StockClerk.RejectOrder(Item(o))}***

ProcessOrder ↔ C.Order(o,vi) <> Continue(o,vi,C)

*Continue(o,vi,C) ↔ (AlarmCustomer(o,C) ⊕ (**StockRequest(Item(o))** <>*

*((**StockClerk.AcceptOrder(Item(o))** <> ProcessPayment(o,vi))*

*⊕ **StockClerk.RejectOrder(Item(o))**)))*

ProcessPayment(o,vi) ↔

DebitRequest(am,vi) <>(

BankClerk.RejectOrder (am,vi) ⊕

(BankClerk.AcceptOrder (am,vi) <> TransferOrder(am,vi,company_account) <>

OrderShipment(inv)))

OPERATIONAL CONSTRAINTS

PRECONDITIONS

AlarmCustomer(o,_) : Item(o) ∉ SoldItemTypes

StockRequest (o) : Item(o) ∈ SoldItemTypes

EFFECTS OF ACTIONS

Continue(_,_,_) : *Busy := TRUE*

[]

Busy := FALSE

COOPERATION CONSTRAINTS

ACTION PERCEPTION

K (C.Order(_,_) / TRUE)

K (StockClerk.AcceptOrder(_) / TRUE)

K (StockClerk.RejectOrder(_) / TRUE)

K (BankClerk.AcceptOrder(_,_) / TRUE)

K (BankClerk.RejectOrder(_,_) / TRUE)

ACTION INFORMATION

K (StockRequest(_).StockClerk / TRUE)

K (DebitRequest(_,_).BankClerk / TRUE)

K (TransferOrder(_,_,_).BankClerk / TRUE)

K (OrderShipment(_).StockClerk / TRUE)