MASTER IN BUSINESS ENGINEERING PROFESSIONAL FOCUS IN DATA SCIENCE

Requirements Engineering

Description and recommandations of use for KAOS, iStar, Tropos, Z Notation, BPMN, and UML

Amand, Alexandre

Award date:
2021

Awarding institution:
University of Namur

Requirements Engineering : Description and recommandations of use for KAOS, iStar, Tropos, Z Notation, BPMN, and UML

Alexandre AMAND

Directeur : Prof. S. FAULKNER

Mémoire présenté
en vue de l'obtention du titre de
Master 120 en ingénieur de gestion, à finalité spécialisée
en data science

ANNEE ACADEMIQUE 2019-2020

Table des matières

1. Introduction

1.1. Context

Information systems are taking a more and more important place in our life. From applications that we are using every day to connected objects that are becoming more and more the norm (GPS, connected watches, vocal assistants, etc.), information are generated always faster in large numbers. In an era when some terms like big data, business intelligence or 5G became frequent in our vocabulary, we have to take care about all the systems (from the simplest to the most complex) that surround us. Information systems need to constantly response to their performance, otherwise they become immediately has-been.

All these technologies and informatic tools that make part of our daily life need different development phases in their life cycle: from idea to conception/implementation, by passing through requirements elicitation. Requirements elicitation is a crucial phase. Its objective is to bound the scope of the system, to understand it, to explain it and to imagine its future. It is important to think as well as the future user as the developer during elicitation. There exists several techniques and mechanisms to capture requirements. It is essential to consider this phase that consists to create a list of all the requirements we will deal with. It is often easier when these requirements are modelled.

It is now commonplace to be able to systemize things, that touch to informatics or no. A simple example to prove it is the construction area. Do you know someone who is building a house without making plan?

But what is concretely an information system? An information system can be defined as a set of resources (staff, software, processes, data, material, informatic and telecommunication equipment, ...) allowing the collect, stockage, structuration, modelling, management, handling, analyse, transportation, exchange, and dissemination of information (texts, pictures, songs, videos, …) in an organization. Nowadays it is of paramount importance to distinguish information system and computer system. An information system may be considered as an “automatable” view of organization’s businesses and as a functional view of informatics. The information system is thus independent of the technical implementation. (Universalis, 2020)

1.2. Requirements Engineering

First, it is interesting to mention what is concretely a requirement. IEEE (Institute of Electrical and Electronics Engineers) gave a threefold definition in 1998. A requirement is: (IEEE, 1998)

1) “a condition or capability needed by a user (person or system) to solve a problem or achieve a goal.”
2) “a condition or capability, which has to be provided by a system or part of system, to fulfil a contract, a standard, a specification or any other formal documents.”
3) “a documented representation of a condition or capability.”

Getting a good requirements documentation is therefore very important. A good requirement must include some quality criteria like unambiguity, understandability, completeness, consistency, verifiability, traceability, relevancy, and feasibility.

The process of Requirements Engineering can be divided in four main phases (Marcelino-Jesus, Sarraipa, Agostinho, & Jardim-Gonçalves, 2014) :

- **Elicitation**: It is the first phase of Requirements Engineering and corresponds to the act to get all the relevant requirements of a system.
- **Analysis**: This second phase is obviously related to the first one. It consists in analysing the elicited requirements to clarify, organize, characterize, and document them before obtaining the specifications of the system.
- **Specification**: It consists in grouping requirements in a suitable form. It means that this phase must provide a readable and understandable requirements document, even for people who was not involved in the first phase.
- **Validation**: This last phase of Requirements Engineering is a phase of review and validation, in order to clarify and make consistent and complete the requirements. Potential faults will be detected here.

A prevalent issue in Requirements Engineering (RE) is the acquisition of the requirements. According to Eric Yu, "Requirements Engineering serves the crucial interface between the world of the application domain and the world of computing technology" (YU, Social Modeling and i*, 2009). The Requirements Engineering analysis is the most critical step in software lifecycle. It is very important not to make mistakes to avoid large effects on many outcomes. To formalize them, a first step is to know them. But the elicitation ("the process of seeking, uncovering, acquiring, and elaborating requirements" (Zowghi & Coulin, 2005)) and the acquisition are very difficult. Clients are not always able to express themselves well, to formulate their problems well or just do not know sufficiently things they are dealing with. Tough acquisition/elicitation is not an easy step, some techniques have been put in place to facilitate it (Lamsweerde, Dardenne, Delcourt, & Dubisy, 1991). The first one is the requirements specification. It is a technique where "*we are learning*". It is a cooperative learning process between clients and analysts. They contribute together to build the architecture of a future model, both with their own expertise and experience. It consists in describing in a rich language all the various concepts (objectives, agents, responsibilities, …). It must be formal enough for the elicitation but not too much coded for the analysts. The goal is to precisely describe the environment and the components to get knowledge about the application domain. Another technique is the formal specification of the requirements where "*we are structuring*". Here the composite system is automated, and we rework the specifications, we formalize them in a formal language. With this method we will get knowledge about the sophisticated formalism. (Dardenne, Lamsweerde, & Fickas, April 1993)

Another view of Requirements Engineering consists in dividing all the process in two main phases: the early-phase and the late-phase.

Early-phase of Requirements

The first one explains the *why* and the *how* questions. It is very important to show some interests for this phase because it is a complex issue to well define the requirements, the constraints, the environments, the domain, etc. It allows to have a powerful general idea of the system and to better understand it and the domain in which the system takes place. The early-phase will meet organizational goals, different possible alternatives, various stakeholders, etc.

Late-phase of Requirements

The second phase is focus on the question: *What the system should do?* This phase aims to provide a requirements document, destined to the developers, in order to well specify and understand the system to implement. Most of the existing requirements techniques are modelled for the later phase of Requirements Engineering. Fortunately, some modelling frameworks and languages (e.g. i*) were afterwards developed for the earlier phase which is as important. (YU, Towards Modelling and Reasoning Support for Early-Phase Requirements Engineering, 1997)

Another important distinction in RE is about the types of requirements, that can be either functional or non-functional. (QRA, 2020)

Functional Requirements

Functional requirements of a system explain the **what**. They define basically the behaviour of the system. They are what the system does and must not do, and how it answers to inputs. Functional requirements include if/then behaviours, computation, data management and business processes.

Functional requirements correspond to features of a system that allow it to work as intended. Therefore, if functional requirements are not satisfied, then the system will not work. They are mostly focus on user requirements.

Non-functional Requirements

Non-functional requirements explain the **how**. It means that their goals are to explain how the system will work, how it does what it is intended (i.e. functional requirements). Non-functional requirements do not affect the functionality of the system. Even if they are not met, the system will do what it is intended to do. It is nevertheless not a reason to make them less important. Non-functional requirements define the system behaviour as well that features which affect the user experience. The way they are defined and executed specify the system's ease of use and performance. Non-functional requirements correspond to the product properties and are more focus on user expectations.

For example, let us take a website. Where a functional requirement would be *to load a web* page, a non-functional requirement would be *the time to load this page*. If it takes 1 second or 10 seconds, the functional requirement will always be satisfied. However, the customer satisfaction and the performance of the system depend on the non-functional requirement.

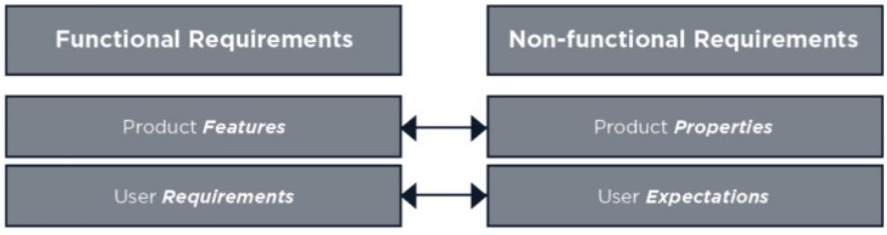


Figure 1: Functional vs. Non-functional Requirements (QRA, 2020)

But why it is important to differentiate the requirements? It is mainly due to the impact that they can have on a system or a project. Functional requirements are the best way for the client to express its

needs to the staff. It allows to keep the development team in the right scope and direction. The functional requirements, listed in a clear document, are then very important to ensure that the final product respects well the target. Providing a bad scope is a big problem and causes other ones. It leads to extra time, that costs some money and consumes more resources. If time or money are not available, the client will get a product with some quality defaults. If the client is in trouble after knowing the cost of its needs and wants, he can obviously ask to reduce the scope. It is here, during the scope reduction that non-functional requirements interact. Excessive non-functional requirements can lead to cost increasing, while insufficient non-functional requirements lead only to poor user experiences. Understanding the differences between functional and non-functional requirements will be an advantage for the client and for the developer. They will be able to better understand the system in its depth and better define the scope and the cost estimation, to lead to better customer satisfaction.

It is important to differentiate a language, a technique, and a framework in Requirements Engineering. A **language** explains how to express the model/system. A **technique** shows how to construct a model, how to do something (with steps, guidelines, etc.). A **framework** is more complete because it consists in a language and a methodology to use it. In a framework there are rules, steps to construct a model, etc.

1.3. Modeling Languages

Since we are still in an information revolution that carries on growing in every domain, we have to cope with a multitude of information, data, and knowledge (YU, Social Modeling and i*, 2009). The term of BigData is taking more and more importance and we are completely in transition to the web 4.0, an intelligent web which demands still more devices. And when we are talking about devices, we are talking about data. New technologies and new services that are emerging for few years requires a right understanding of the requirements (from software and clients). Requirements Engineering is very broad and multi-disciplinary. It is then very difficult to have uniforms notions to model a system and understand it. Therefore, modelling languages are multiples. There exist different kinds of languages, approaches and ontologies. Ontologies can be static or dynamic but also social or intentional for example (YU, Social Modeling and i*, 2009). Some languages are useful for the early-phase of Requirements Engineering while others are useful for the late-phase. Approaches can be based on a specific concept like the goal or be focused on the whole process. Social modeling was imagined because a lot of models for software and information systems were often based on relationships (for example entity-relationships, class diagrams, etc) with behavioural properties like abilities, capacities, behaviour, etc. Such aspects are integrated in the i* modeling language (see section 3) (YU, Social Modeling and i*, 2009).

1.3.1. GORE: Goal-Oriented Requirements Engineering

With the information revolution and the growing multitude of modeling languages, the number of appearances of the goal notions also grew. This approach of Requirements Engineering places the concepts of *Goals* in the centre of the model(s). “Goal-oriented Requirements Engineering is concerned with the use of goals for eliciting, elaborating, structuring, specifying, analysing, negotiating, documenting and modifying requirements” (Lamsweerde A. v., Goal-oriented requirements engineering: A guided tour, 2001). In response to the growing complexity of the systems, this approach has been developed. The objective is to have a better understanding of the (high-level) requirements and the whole process thanks to the goal concept. Because traditional

modelling and analysis methods do not allow certain configurations of systems for certain functionalities, it is a real issue that GORE is trying to solve. GORE puts the attention both to the activities that precede a goal, and the goals themselves (goal elicitation, goal refinement, goal analysis, etc.) (Lapouchnian, 2005). A goal can thus characterized an activity or a real objective of the system. It plays either a central role or is supporting the system. (Yu & Mylopoulos, 1998)

There exist lots of different definitions of a goal because it depends on the language. For example, in KAOS, van Lamsweerde defines it as an objective that the system should achieve thanks to a cooperative balance between agents of the system and its environment (Lamsweerde A. v., Requirements engineering in the Year 00 : A research perspective, 2000). In addition to KAOS, GORE approach also contains iStar and Tropos language. The goal-oriented modelling languages are explained and developed with further explanations in section 2 to 4.

Using a goal modelling approach to model a system and its environment includes some benefits. Lapouchnian cites some of the most important ones in his paper (Lapouchnian, 2005) :

- The perspective used by GORE are larger than for the typical methods. Goals are prescriptive assertions, domain properties and the environment are explicitly captured during the requirements elaboration process, and goals justify the reason that make them operational. It allows to easily support the early phase of requirements analysis.
- Thanks to the use of goals, the specifications are more complete if they can be achieved by respecting the domain and the system properties.
- In addition to completeness, the use of goals also brings some pertinence in the model if the goal is well achieved.
- A hierarchy in the activities and a differentiation between high-level and low-level goals can be made. It is the traceability feature of goal modelling. It also helps when we are dealing with complex requirements to structure them.
- Both quantitative and qualitative analysis are possible because a goal model can capture the variability of a domain thanks to alternative goals and goals refinement.
- Goal modelling allows a good communication with the customers thanks to the use of goals. Decision makers are more involved because they need to make some choices about the goals and the goals refinement.
- Since goals are more stable than other low-level modelling concepts in Requirements Engineering, it allows to better understand the requirements. Generally, a goal is more stable higher the level it is.

Traditional methodologies only consider requirements as the processes and data. They do not try to understand why the system work by a certain way, there are any *why* and *how* about the system. Goal-oriented methodologies will understand the system and put the emphasize on the how and why questions. Goals allows to clarify requirements that are not always very clear when the clients or the stakeholders communicate them to the analysts, overall, in the case of nun-functional requirements. Moreover, goals provide a useful way to deal with intern conflicts because 2 goals can interfere together. It is important to differentiate here the satisfaction (i.e. the action to satisfy a need) and the satisficing (i.e. the decision-making strategy that aims for a satisfactory or adequate result, rather than the optimal solution). (Yu & Mylopoulos, 1998)

1.3.2. Agent-Oriented Requirements Engineering

For several years now, software and software engineering have changed of state in the mind of developers. They need now to be based on open architectures to continuously contain changing and evolving dimensions to accommodate new system components and requirements. Software must also be operational on different platforms, considering then new constraints and users' needs. Therefore, software needs to be more robust, autonomous, and able to serve end-users with the minimum of overhead and interference as possible. All these new requirements generate a turn to new concepts, tools and support techniques for the software and Requirements Engineering. Agent-oriented was imagined to response to these issues and gained quickly in popularity in contrast to traditional methods (e.g. structured and object-oriented techniques). Agent-oriented modelling allows an open and evolving design of the system that takes into account new agents and their services. Moreover, agent-based approach allows agents to deal with unforeseen situations because the software design includes some concepts like goals linked to a planning of the capability to meet them. (Giorgini, Kolp, Mylopoulos, & Pistore, 2004)

Such an approach is used in i* and then derived and used in Tropos.

Now that Requirements Engineering is introduced, six modelling languages and frameworks will be described. Section 2,3 and 4 will describe GORE with KAOS, iStar and Tropos. A more formal language called Z is described in section 5. Section 6 covers BPMN which represents dynamic systems. The last model, UML, is described in section 7. Finally, section 8 will provide a conclusion and comparative table, aiming to bring some recommendations of use to business analysts.

2.KAOS

2.1. Description

KAOS means **K**nowledge **A**cquisition in aut**O**mated **S**pecification (Dardenne, Lamsweerde, & Fickas, April 1993) but *van Lamsweerde at al.* also interpret the acronym as **K**eep **A**ll **O**bjectives **S**atisfied (Lamsweerde & Letier, From Object Orientation to Goal Orientation : A Paradigm Shift for Requirements Engineering, 2003). « KAOS, is a goal-oriented software requirement capturing approach in Requirements Engineering. It is a specific Goal modeling method; another is i*. It allows for requirements to be calculated from goal diagrams. » (Lapouchnian, 2005) It is born from an idea and a cooperation of several professors from University of Louvain (Belgium) and University of Oregon (USA) in 1990. It is become a university reference to capture software requirements since its creation. A lot of requirements software are more based on the specifications of the solution instead of the problem itself. The current issue is then that most of the existing specifications languages are oriented to the functional requirements (i.e. what the software is expected to do) and let the non-functional requirements (i.e. operational costs, responsibilities, interaction with an external environment, reliability, integrity, flexibility,…) besides (Mylopoulos, Chung, & Nixon, 1992). It is the main idea behind the KAOS methodology which has 2 forces, that is the reuse of domain knowledge and the application of machine learning technology.

KAOS is a Requirements Engineering framework (i.e. a language and a methodology) that involves both analysts and the client. It contains a graphical part because KAOS relies on the construction of a requirements model. It is this model that is the contract between analysts and the client. As a goal-oriented language, KAOS describes goals (high-level and subgoals) in order to structure them.

2.2. Approach

The KAOS approach is composed by 3 main components: a conceptual model, acquisition strategies and an acquisition assistant (called GRAIL). KAOS has 3 levels of modelling from domain-independent abstractions to specific case of domain-level concepts. It is the meta-level, the domain level and the instance level.

2.2.1. Conceptual Model

The conceptual model delivers some abstractions in terms of requirement models to be acquired, it is the meta model. The meta model is a conceptual graph where nodes are representing abstractions and where edges are representing the structuring links of the model (*see figure 1*). The KAOS conceptual meta-model considers both functional and non-functional requirements.

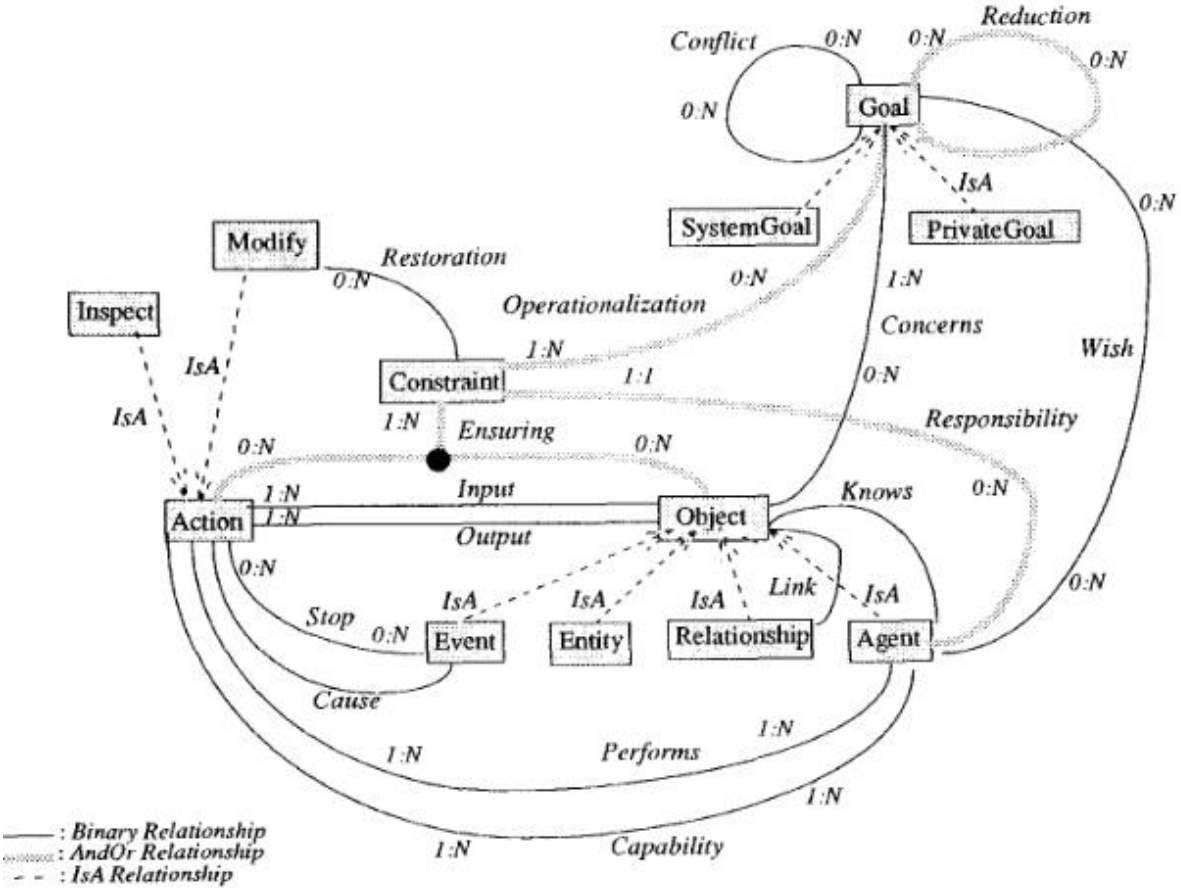


Figure 2 : a portion of the KAOS conceptual meta-model (Dardenne, Lamsweerde, & Fickas, April 1993)

The KAOS approach based on this conceptual meta-model considers 3 levels of requirements acquisition as shown in figure 3. The first one is the **meta-level**. That is the largest level and refers to domain-independent abstractions. In this level, we can find the meta-concepts (i.e. agents, actions, relationships, …), the meta-relationships (i.e. Is a, performs, input, …), the meta-attributes (for example the cardinality) and the meta-constraints. The intermediate level is called the **domain-level** and describes the specific concepts according the domain application of the system. Here a concept is an instance of a meta-concept. The more specific level is the **instance-level**. It corresponds to the particular instances of the domain-level. The representation of the conceptual meta-model shows well how KAOS allows to go from generic to specific requirements and information through the acquisition strategy.

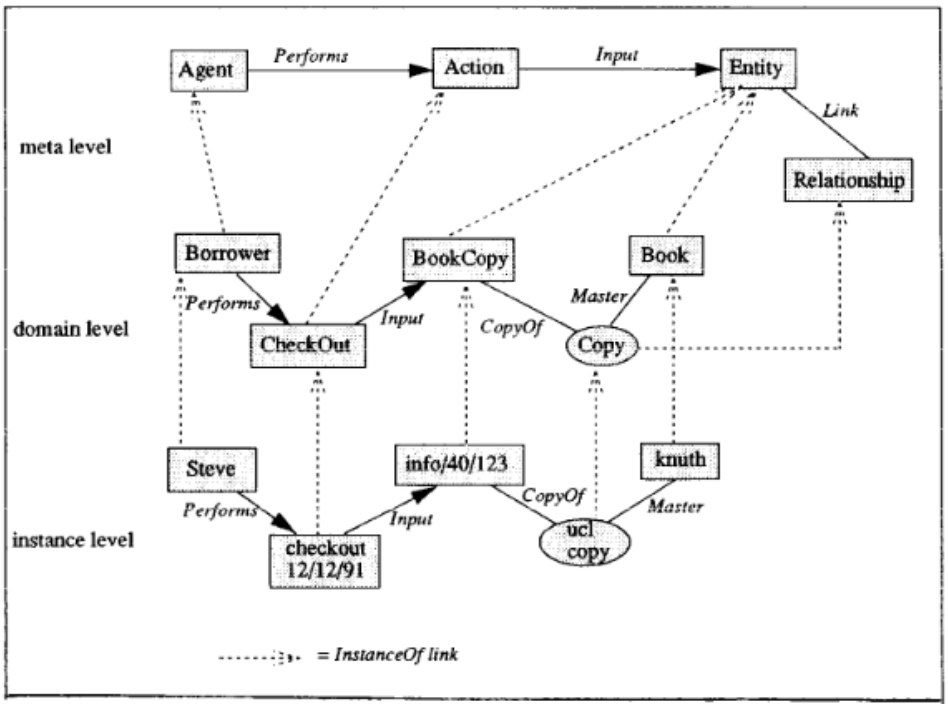


Figure 3 : the meta, domain and instance level (Dardenne, Lamsweerde, & Fickas, April 1993)

2.2.2. Acquisition Strategies

The KAOS methodology contains a set of strategies to elaborate the different opportunities of requirements models. The role of the strategies is to well describe the different steps to acquire the components of the requirements models as specific instances of the meta model and to detail how to go to the meta model. It is important to notice that each step in a strategy is itself composed from finer different steps. Dardenne at al. (Dardenne, Lamsweerde, & Fickas, April 1993) provides an acquisition strategy specific to the KAOS meta model. This strategy is a learning-by-instruction one to put the emphasis on the acquisition step. The principle is to browse the meta model backwards from the goal nodes through the adjacent links. The strategy provides a path for which at each node the corresponding meta-level knowledge is used to guide instance acquisition. That allows to give more importance to the high-level goals. Instead of starting from the low-level subprocesses as it is usually the case, we start from the system level and the organizational objectives from which will progressively arise lower-level descriptions.

This strategy suggested by *Dardenne at al.* gives a lot of importance to the goals, such as (Lamsweerde, Dardenne, Delcourt, & Dubisy, 1991):

- Goals incorporate the requirements components
- Goals justify why there are requirement components (not necessarily understandable to clients)
- Goals may be used to represent specific responsibilities of agents from the system
- Goals may explain and define why such an agent should perform such an action, according to the system criteria
- Goals provide information in order to detect and solve some conflicts that come from divergent and multiple viewpoints among human agents (Robinson, 1989)

Like the goals, the meta-model itself plays an important role in the acquisition strategy discussed by *Dardenne at al.* (Dardenne, Lamsweerde, & Fickas, April 1993).

- The instances of the concepts (meta-relationships, meta-concepts, meta-constraints, etc.) gives the requirements components of the system.
- Those same concepts and meta-concepts determine the general structure of the acquisition language.
- Also, from the concepts and meta-concepts, the system inherits features that are defined only once.
- The meta-model explains how to conduct the acquisition process. For example, by starting from goals. It is important to remember that better we know about the environment, better we can guide the acquisition process.
- The meta-model is very useful when basic tools do not know a lot about the application domain by providing a base for the system application domain (Lamsweerde, Delcourt, Delor, Schayes, & Champagne, 1988)

To briefly summarize how works the meta model, it is necessary to start from the most general aspect, the objects (i.e. something which can be involved in requirements). All the entities, relationships, events, or agents are objects. (see in figure 3)

"An entity is an autonomous object; its instances may exist independently from other object instances." For example, it is a borrower, a book, a bookcopy or a library.

"A relationship is a subordinate object; the existence of its instances depends upon the existence of the corresponding object instances linked by the relationship." For example, it is Input, Performs, CopyOf, etc.

"An event is an instantaneous object; that is, its instances exist at the instance level in one state only." An example of an event could be a ReminderIssued when a certain borrower gives a book later back.

"An action is a mathematical relation over objects. Action applications define state transitions." They include precondition, triggercondition and postcondition.

"An agent is an object which is a processor form some actions; agents thus control state transitions."

All these features allow the KAOS meta-model to be very powerful because it can structure both complex objects and actions into requirements components. It can also support repetitive scenarios thanks to the different concepts or handle conflicts. Finally, the mapping meta-relationship allows to easily link some objects or actions. (Dardenne, Lamsweerde, & Fickas, April 1993)

2.2.3. Acquisition Assistant (GRAIL)

The third and last component of the KAOS approach is the acquisition assistant. It consists in an automated support to help to follow a specific acquisition strategy. The assistant is divided in 2 different bases that are structured according to the components of the meta model: a requirements data base and a requirements knowledge data base. The second one is composed by a domain-level knowledge (i.e. the concepts and the requirements) and a meta level knowledge (i.e. the properties

(for example, a constraint that can be temporarily violated and that has to be handled by some appropriated actions) and the ways to conduct the strategy).

The acquisition assistant had been imagined responding to 4 main functions. It suggests issues to focus on and/or to come back on. It proposes requirements components to be included in the model thanks to the use of learning techniques. It provides explanations and summaries to support the global model. Finally, it supports multiple user viewpoints and their collaboration. Its knowledge based is organized by product-level knowledge (i.e. the general concepts), process-level knowledge (i.e. acquisition operators that can be applied for making progress in the acquisition state space) and rational-level knowledge (i.e. tactics and strategies that govern the application of acquisition operators). (Lamsweerde, Dardenne, Delcourt, & Dubisy, 1991)

GRAIL is a tool that can automate the production of the KAOS models (see section 2.4.). The GRAIL environment provides adequate support for goal-driven elaboration. It combines a graphical view, a textual view, an abstract syntax view, and an object base view of specification. The current version of GRAIL includes textual editor, a graphical editor, a hypertext navigator, a Latex report generator and an object base. The textual editor supports requirements acquisition and incremental both at declaration and assertion level. The graphical editor complements the detailed textual view with a high-level graphical view of the specification. The graphical view of the specification is easier and quicker to understand thanks to a powerful and automatic layout. The hypertext navigator allows specifiers to browse the entire specification in hypertext mode. The object base uses queries to make checks on the requirements specification. (Darimont, Delor, Massonet, & Lamsweerde, 1998)

2.3. A Goal Directed Acquisition Strategy

A main preoccupation of a system is that all the requiremets and instances have to always satisfy meta-constraints (like cardinality or other meta-constraints previously specified). As explained in section 2.2., the acquisition process progressively capture requirements by following the structure established in the meta-model. The acquisition processes are then drived by different strategies and domain models. A strategy define the specific way to acquire requirements and each step is itself divided in finer steps. An acquisition session consists in traversing the meta-model to acquire corresponding instances by using acquisition operators. The order of the traverse is determined by the employed strategy (Lamsweerde, Dardenne, Delcourt, & Dubisy, 1991). Domain models and requirements are described and defined in the same acquisition language. They are also organized like IsA inheritance hierarchies and supported by the acquisition assistant. The acquisition strategy suggested by *Dardenne at al.* is a goal-oriented one, and is composed by **7 steps** (Dardenne, Lamsweerde, & Fickas, April 1993). While the following steps are ordered, they may overlap and the strategy may be retroactive.

1) Acquisition of Goal structure and identification of Concerned Objects

This 1st step which makes interact the analysts with the client(s) is decomposed in 3 main parts. The first one is to identify SystemGoals and their features (patterns, categories, …) and afterwards to associate them to the parent goal(s) from which they arise. The objective is to also formalize subgoals according to what it has been specified before. It consists in reduce the goals.

Once the reduction is done (i.e. goals are operational), we must identify the concerned objects and define their features. Pay attention that the reduction must minimize the number of agents that will proceed the goals/actions, the costs, and the number of potential conflicts.

Then, the last sub step is to identify the possible conflicts between SystemGoals (i.e. conflict meta-relationship). Conflicts have to be prioritized and handled by priority.

2) Preliminary identification of potential Agents and their capabilities

This step includes both the identification of the agents in the system and the identification of the actions they are able to do (i.e. capability meta-relationship between a concept and an agent). It is important to always satisfy the constraints. For each action, we will list the human agents able to perform this action, and the possible automated agent available. It is important to well operationalize and instantiate the goals to achieve this step.

3) Operationalization of Goals to Constraints

The leaf goals and goals from step 1 will here be transformed into system objectives in terms of concepts. The reduction is used to transpose the constraints. The objective is to get as few as possible actions to handle and restore conflicts. An example of a constraint should be the limit period to borrow a book in a library.

4) Refinement of Objects and Actions

New objects and actions which arise from step 3 can induce some new constraints. It is also important to complete what was already identified (description, features, …).

5) Derivation of strengthened Actions and Objects to Ensure Constraints

Actions and objects do not necessarily guarantee that constraints will be met, though it is essential for the system. Strengthened actions are put on precondition, postcondition, invariants and sometimes on trigger conditions. It allows to ensure that the constraints are satisfied, while minimizing restrictions on actions.

6) Identification of alternative Responsibilities

This 6th step consists in linking agents and constraints by responsibilities according to their capabilities, description, features, etc. All potential responsibility links are identified and evaluated according to a cost value, motivation value and reliability value.

7) Assignment of Actions to responsible Agents

The last step of the strategy consists in assigning the actions to the agents, based on *performs* links elaborated in step 2 and 4 and responsibility link from step 6. A mistake to not produce is to assign powerful agents to an action which would prevent satisfying other constraints. The assignment starts from actions/constraints of high priority goals and must maximize reliability of the system while minimizing costs of performance. It is also important to avoid overloading agents. An agent is responsible of an action only if he is capable of it.

2.4. KAOS Models

KAOS methodology is composed of 4 types of models, that are inter-correlated (see figure 5). Modeling components are specified as shown in figure 4.

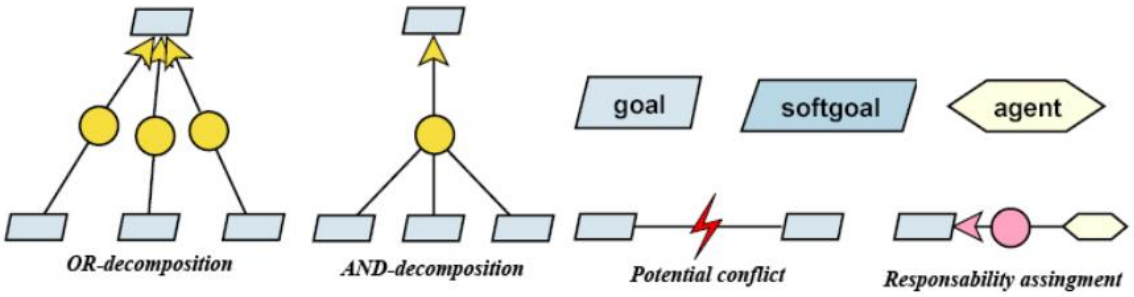


Figure 4: Modeling constructs of the KAOS methodology (Teruel, Navarro, opez-Jaquero, Montero, & Gonzalez, 2012)

2.4.1. Goal Model

The goal model represented goals and assign them to agents. It is the set of interrelated goal diagrams that have been put together for solving a particular problem. The top goal represents the strategic objective of the system and each goal is refined as a collection of subgoals. Achieving all subgoals consists in achieving the parent goal. In this modeling method, conflict relationship, and-relationship and or-relationship are modellable.

2.4.2. Responsibility Model

The KAOS Responsibility Model is the set of responsibility diagrams derived from the Goal model. A responsibility diagram describes for each agent the requirements and expectations that he is responsible for, or that have been assigned to him.

2.4.3. Object Model

The KAOS Object Model is used to define and document the concepts of the application domain already identified to satisfy the known requirements. It makes it easier for analysts to create and maintain a glossary section. When some new objects are identified while traversing the Goal Model, the analyst defines them in an object diagram and links them to the existing concepts. They can work on the glossary progressively and simultaneously during goals and requirements definition and acquisition. The glossary is built by browsing the Object Model and listing all the concepts that are inside. The KAOS Object Model can be compared to an UML Model in which KAOS entities are UML classes and KAOS associations are UML binary associations.

2.4.4. Operation Model

The operation Model sums up all the behaviours that agents need to have to fulfil their requirements. Behaviours are expressed in terms of operations performed by agents. In KAOS, the Operation Model is connected to the Goal Model. The analysts justify operations by the goals they "operationalize". A requirement ca be operationalized by some objects, some agent behaviours, or a combination of both.

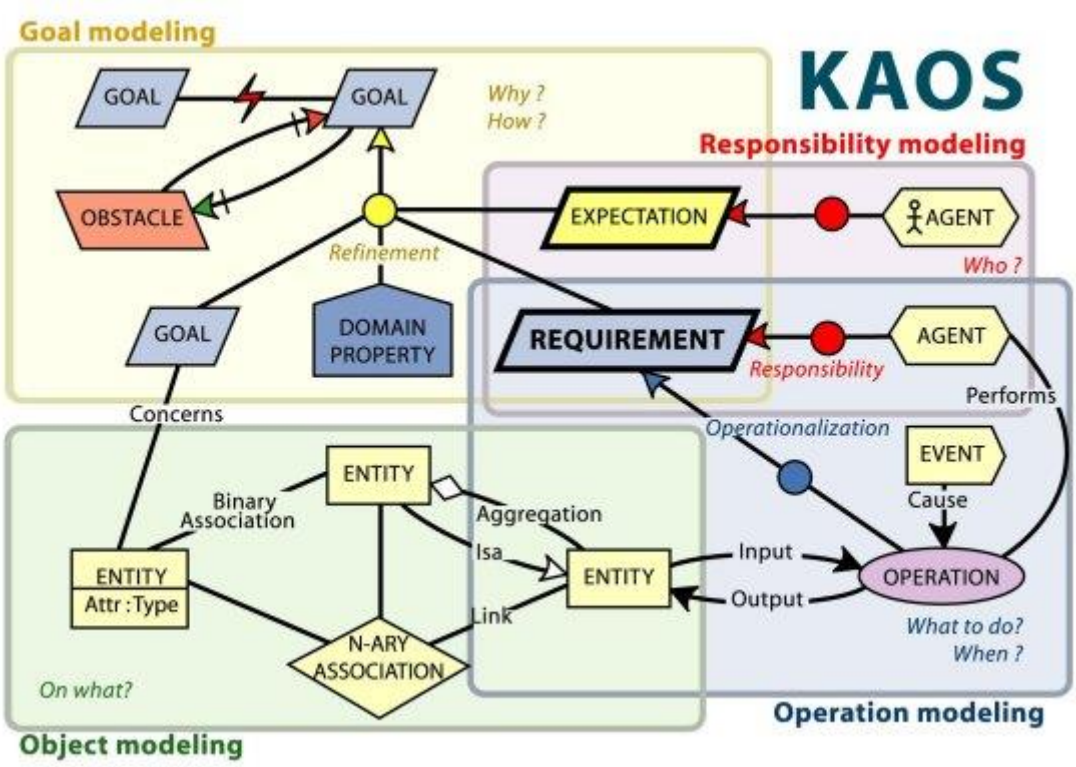


Figure 5: KAOS meta-model and 4 submodels (RESPECT-IT, 2007)

2.5. Benefits and critics

The use of the KAOS methodology has some benefits and motivations. (RESPECT-IT, 2007)

- *Completeness*: Requirements documents elaborated with KAOS tend to be more complete. The KAOS completeness can be explained by the fact that the models let no space for wishful thinking, no space for requirement for which we do not know who is responsible for, no space for unjustified operations and no space for operations for which we ignore who will execute what and when.
- *Traceability*: A major benefit of KAOS resides in the fact that it provides continuum between the problem description and the expected solution description. As the system requires frequent changes to follow market trends, it would be very interesting to keep the initial requirements document up to date with respect to the current state of development. In KAOS, the requirements document is derived from a KAOS model, so it is possible to modify the KAOS model and generate a consistent requirements document.
- *No ambiguity*: On the one hand, the completeness contributes to less ambiguity in requirement documents because we know who is responsible for which actions and who performs what. On the other hand, the Object model contains all the information useful to produce the requirements document glossary. The glossary validation forces the stakeholders to agree on the domain and application relevant concepts even if they generally have different background.
- *Conflict management*: Goals can be used to detect and manage conflicts among requirements.

The KAOS methodology has also limitations. The main limitation is the time to develop a system according to the KAOS methodology. A KAOS requirements study is justified as soon as the project man power is more than 100-man days. For medium and big-size projects a cost reduction of 30% can be expected. For other projects, the probability to get a positive return-on-invest is very low because the construction of a KAOS requirements models consumes lot of times. (RESPECT-IT, 2007)

2.6. Study Case

The following model is derived from a study case about Expert Committee, initially built in I* models (Oliveira, Cysneiros, Leite, Figueiredo, & Lucena, 2006).

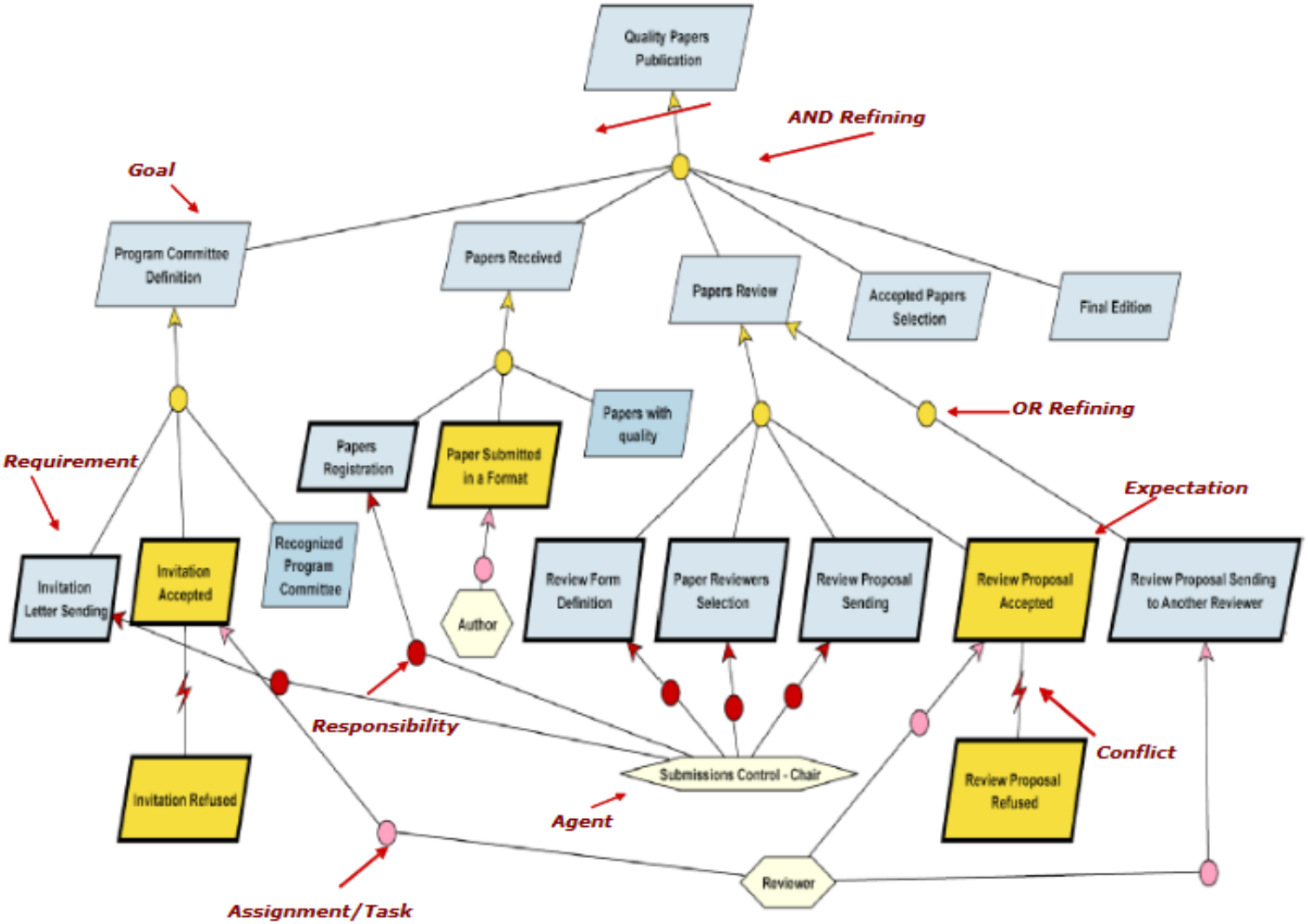


Figure 6: The Expert Committee (EC) study case in KAOS (Werneck, Oliveira, & Leite, 2009)

3. I Star (i*)

3.1. Description

iStar was developed by Eric Yu (from University of Toronto) in 1995 and means **I**ntentional **S**trategic **A**ctor **R**elationship modeling. I STAR is also used as **i***. Although his author mentioned it as a framework, it is not really the case. i* is more a modeling language than a framework because Yu do not explain a certain method to use it. i* will express something, a system. It will look to it, identify stakeholders and agents, and will represent all with his own concepts. The objective of i* is to be a support in the reflection of the environment of an organization and the system. i* is an attempt to integrate some aspects from social modeling and reasoning in Requirements Engineering and its different methods and techniques. i* has the difficult task to well combine both agents' interests (goals, beliefs, abilities, commitments) and system's interests. It will then adopt a social modeling ontology for its main modeling concepts by taking actor as the central modeling construct. (YU, Social Modeling and i*, 2009)

3.2. Approach

First, I* is very useful and important for the early phase approach of Requirements Engineering because the language answers to the *why* and *how* the system takes place in an organization. Eric Yu developed the i* framework to support modelling and reasoning about organizational contexts and their information systems. Several numbers of levels of analysis for early-phase of Requirements Engineering, in terms of ability, workability, viability and believability are possible thanks to i* models. Tough KAOS framework also uses the notions of goals and agents, it provides a good methodology for requirements acquisition but makes any difference between early and late phase of Requirements Engineering, in contrast to i*. (YU, Towards Modelling and Reasoning Support for Early-Phase Requirements Engineering, 1997). The general approach of the i* language is both goal and agent-oriented.

It is a goal-oriented modeling language because the concept of goal takes a central place in i*. Organizational goals and stakeholders' interests are very important, and a particular focus is put on it. In i*, each actor and agent (human, software, hardware) act according its own interests and objectives.
It is also an agent-oriented modeling language due to the high importance of the agents and the relationships in the language. i* is a language that represents all the relationships between the actors present in the system. This concept is quite simple: An actor/agent (human or not, is the *depender*) needs that other actors (human or not, *the dependee*) to achieve a task (the goal of the *depender).* Such relationships can involve *opportunities* or *vulnerabilities* for the agents. An opportunity happens when the depender can achieve a goal thanks to the dependee while he could not do it alone. A vulnerability happens when the dependee do not what he is expected to do. Actor is the central concept in i*. (YU, Towards Modelling and Reasoning Support for Early-Phase Requirements Engineering, 1997) The i* modeling language is used to get better understanding about organizational relationships and the rationales hidden behind them (Eric YU, 1995).

A major concept of the i* modeling language is the independency between autonomous and intentional actors who have behavioural properties. They have a certain freedom, but it is very

important that actors satisfy the requirements of the system and the organization. i* is built on 4 main premises (YU, Social Modeling and i*, 2009):

- Actor autonomy: In i*, actors can be either human, hardware, software or whatever and are capable of independent actions. Engineers need to deal with this issue (not fully controllable behaviour) when they are modeling a system or an organization.
- Intentionality: The actors do not behave randomly. They logically want to serve their own goals and interests. Therefore i* takes the intention into account to express why a particular actor is taking a particular decision or is doing a certain action, by respecting the autonomy premise.
- Sociality: As usual, social phenomena is a complex concept. A model is limited to represent it. i* is focus on one aspect of the sociality, which is the dependency between actors. “Actors depend on each other for goals to be achieved, tasks to be performed, and resources to be furnished”. It is the SD model that tries to answer to these questions.
- Rationality: Actions are related to attributed goals and motivations. The SR model focus then on these intentional elements and aims to justify why actors act like it is modelled.

3.3. I Star Modeling

i* is composed by 2 models: The Strategic Dependency (SD) model and the Strategic Rationale (SR) model. The concepts use in each model are almost the same. (Eric YU, 1995)

3.3.1. Strategic Dependency (SD) model

This first model is focused to answer to “Who depends on who?”. It consists then in an external intentional model. The SD model represents the different relationships between the actors in an organizational context and in terms of goals. The model consists of a set of nodes and links where each node represents an actor and each link between 2 actors means that an actor depends on the other to achieve a goal. It allows to understand how the system we are modeling is embedded in his organizational environment. The analysts can make cooperate some systems when they notice that an actor intervenes in several systems. i* allows to express each operational configuration thanks to the SD model; the different alternatives being explored in the SR model.

The concepts of depender and dependee explained in section 3.2. take sense in the SD model. A third concept must be explained. It is called the *dependum* and is an intentional element (i.e. a resource a task, a goal or a softgoal) around which the dependency is centred. The different types of dependencies are seen in section 3.3.3.

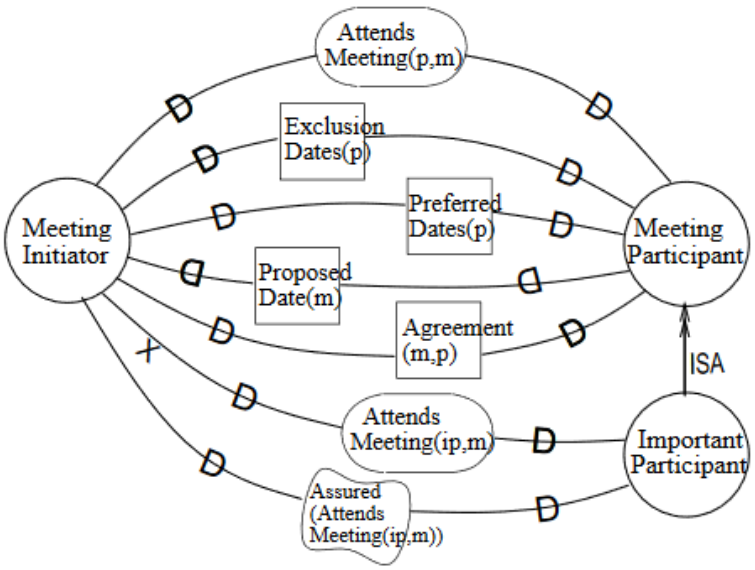


Figure 7: Example of SD model for meeting scheduling (YU, Towards Modelling and Reasoning Support for Early-Phase Requirements Engineering, 1997)

3.3.2. Strategic Rationale (SR) model

In contrast to the SD model, the SR model is an internal intentional model because it puts the attention to the intern activities and tasks acting by each actor. Each actor has some intentional properties (goals, beliefs, abilities, commitments) (YU, Towards Modelling and Reasoning Support for Early-Phase Requirements Engineering, 1997). They also have strategic interests that motivate them to achieve their tasks and to interact with the other actors. The SR model can then answer to 2 main questions: Why actors do what they do? And why an actor is linked to another one? This model is an answer to the limitation of the first one. Indeed, the SD model allowed only limited analysis due to its strong assumption about actor autonomy, and the fact that it is only focus on the external relationships between actors, staying mute on internal aspects of the systems. (YU, Social Modeling and i*, 2009)

Like the SD model, the SR one is also composed by a set of nodes and links where each node represents a dependum (i.e. goal, task, resource or softgoal). The SR model contains only 2 different links compared to the SD model: the means-ends links and tasks-decomposition links (see section 3.3.3.)

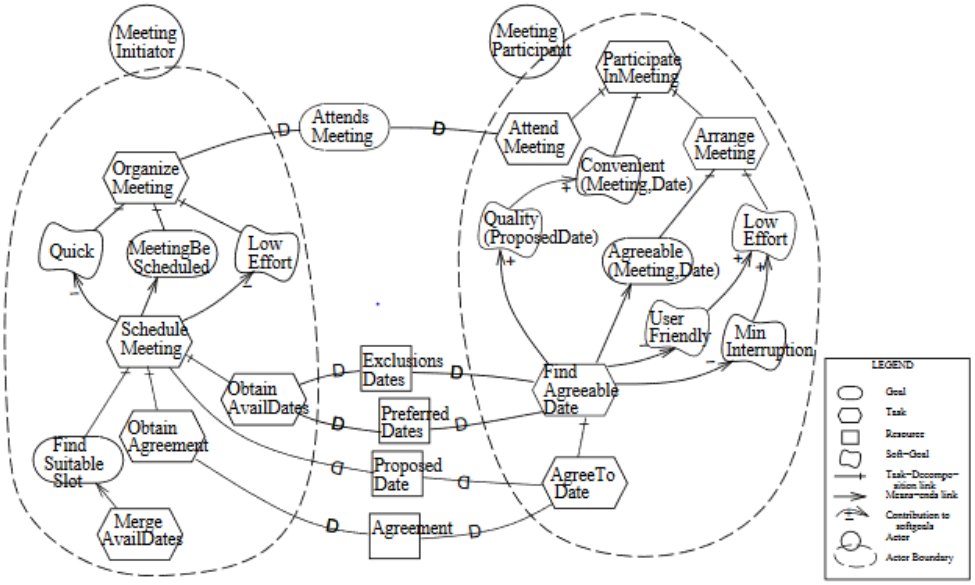


Figure 8: Example of SR model for meeting scheduling (YU, Towards Modelling and Reasoning Support for Early-Phase Requirements Engineering, 1997)

3.3.3. Modeling concepts

As i* is a modeling language, it contains his own concepts/constructs to express a system.

1) Overview of i modeling constructs*

In Social Modeling and i*, Eric Yu exposed the modeling concepts specific to the i* language (see Figure 9). The constructs are more precisely explained in the next sections. (YU, Social Modeling and i*, 2009)

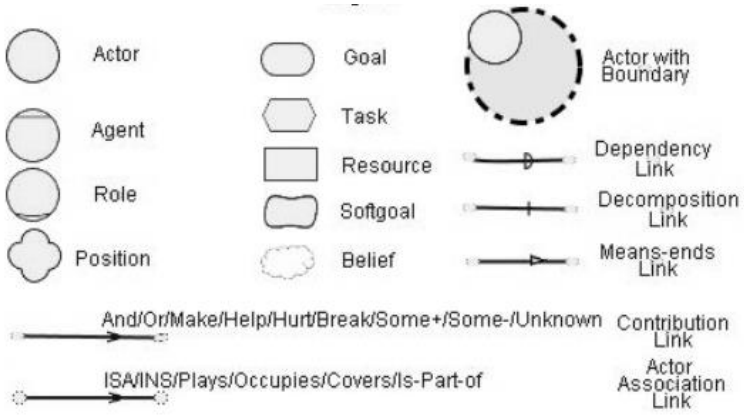


Figure 9: Overview of i modeling constructs (YU, Social Modeling and i*, 2009)*

2) Actors

Franch et al. give an appropriate definition for the actors in i* and define it as “active, autonomous entities that aim at achieving their goals by exercising their know-how, in collaboration with other actors. They may be human (e.g. a person, a role played by a person), organizational (e.g. a company, a department, an agency) or technological (e.g. a software agent, cloud system, some device)”. (Xavier Franch, 2016)

Actors can be generic (i.e. without any specialization) but they can also be represented as an agent, acting a role, or taking a position (See figure 10). According to the types of actor we want to model, the representation is different. An agent is an actor with concrete and physical properties (for example Eric Yu). A role is an abstract characterization of the behaviour of a social actor (For example a researcher). Finally, a position is an intermediate abstraction that can be used between a role and an agent (Fabiano Dalpiaz, 2016). Actors are modelled as modeling abstraction and it is the analysts who decide how to model them, their scope, their identity. Analysts decide if they represent people in a work group individually or directly as an entire group of actors. Boundaries and intentions of actors are represented in i* through the relationships they have with other actors (YU, Social Modeling and i*, 2009). It is very important to notice that actors have different degree of freedoms and may violates some constraints or commitments (Eric YU, 1995).

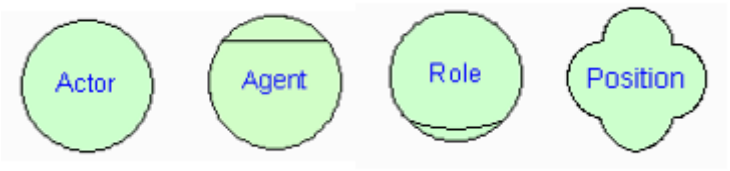


*Figure 10: Types of actors in i**

3) Links

Links are the modeling constructs to join different kinds of actors. There exist 3 types of actor links: is-a link, is-part-of link and plays link (See figure 11). The is-a links actors of the same type. One actor/role specializes another actor/role who has the same type (for example a bachelor student is a student). The is-part-of links also actors who have the same type. The source and the target actors must be of the same type (for example, the Faculty of Law is part of University of Namur, it is 2 agents). The last actor link is the plays one. It links an agent to a role (for example Eric Yu plays the role of a researcher).

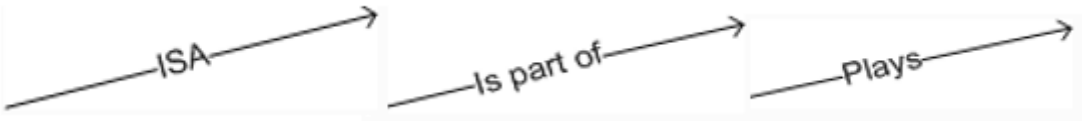


*Figure 11: Types of actor links in i**

As mentioned in section 3.3.2., the SR model uses 2 additional types of links (see figure 12). The means-ends links explains why an actor do a specific task. The means contribute to the achievement of the end, usually the mean is a task. There is an OR relationship when considering many means. It can be either relationships between goal and task, resource and task, task and task, softgoal and task, softgoal and softgoal or goal and goal. The other extra links are the task-decomposition ones. They represent a description of intentional elements that provide a routine for a task. It consists in the decomposition of a task into different intentional element. When the task is decomposed into more than one element, the relationship to consider is an AND relationship because high tasks are achieved only if subtasks ae fulfilled. (YU, Towards Modelling and Reasoning Support for Early-Phase Requirements Engineering, 1997)

Figure 12: Additional types of links for SR model

Another type of links is also used in the SR model. It is the contribution-links and determine the contribution to the accomplishment of a softgoal only. It can be positive or negative but also an AND (a parent softgoal is satisfied only if all the internal elements are fulfilled) or an OR (a parent softgoal is satisfied only if one of the internal elements is fulfilled) relationships. It can also be a help (a contribution for the realization of an internal element, but not sufficient to achieve it in entire), a hurt (partially negative but not sufficient to deny the softgoal) or a break (partially negative and sufficient to deny the softgoal) relationship. Contribution-links allows to have more information about the softgoals and how and why they are achieved or not.

4) Dependencies

In i*, we distinguish 4 types of dependency based on different dependum (see figure 13). Therefore, each dependum leads to a type of dependency. The dependency aims to show the degree of freedoms allowed in the relationship. (Eric YU, 1995)

The first one is the **goal dependency**. The depender depends on the dependee to achieve a certain goal (dependum) that would modify the state of the world but the dependum is vulnerable as the dependee may fail to achieve it.

The second one is the **task dependency**. The depender depends on the dependee to execute an activity, a task. This kind of dependency should specify how the task get done and not why. Since the dependee may fail to fulfil the task, the depender is vulnerable.

The third one is the **resource dependency**. The depender depends on the dependee for the availability of an entity which can be physical or informational. This dependency gives the depender the ability to use the resource. The depender becomes vulnerable as the resource could turn out to be unavailable. We have to notice that a resource dependency is different from a resource flow which does not mandatory imply a resource dependency.

The last one is the **softgoal dependency**. A softgoal is a goal that is not clearly defined. Usually, a softgoal will be clarified while trying to reach it by finding alternatives. Alternatives are found by the dependee whilst the decision is made by the depender. The depender depends on the dependee to meet some tasks in order to reach a softgoal. In contrast to a goal dependency, softgoal dependency conditions are elaborated during the task. Here the dependum aims to represents a quality (e.g. fast, cheap, reliable, secure, etc.). Softgoals are then like goals but with additional quality criteria for the achievement of the goal. Such a concept is very useful to model concepts from the social world that requires contextual interpretation. (YU, Social Modeling and i*, 2009)

The SD model also considers the different degree of a dependency. More the depender is vulnerable, stronger is the dependency. From the dependee point of view, it would mean that he must do a greater effort to provide the dependum. The 3 different degrees are open (market with **O**), committed (no mark) and critical (marked with **X**). Since the depender depends on someone else (the dependee), he cannot be sure that his own goals will be achieved. Everything depends on the

freedom of the dependee to do or not what he is expected to do and how it is specified. (YU, Social Modeling and i*, 2009)

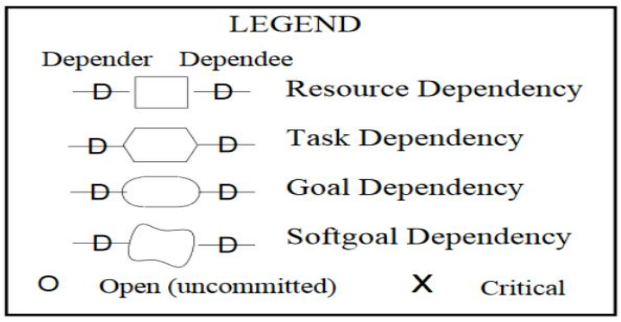


Figure 13: Types of dependencies in i (YU, Towards Modelling and Reasoning Support for Early-Phase Requirements Engineering, 1997)*

3.4. Application areas of i*

i* was developed to be useful in several areas. The main domain in which i* is used is obviously the Requirements Engineering. Lots of efforts has been introduced to develop both requirements models and language to capture precise specifications about what the system should do. It is very important to well define requirements and specifications in adequation with the system's environment. There exist a lot of complex social environments, for which social modeling can provide a new way to analyse, impossible with a static or dynamic traditional ontology. i* modeling language is then become very useful to capture and get the requirements of diverse domains as air traffic control, agriculture, healthcare, e-government, submarine systems, telecommunications, industrial enterprise, and business processes (see below). All these domains involve the cooperation of different actors together, with each their own intentions and interests. The use of i* allow to better integrate the stakeholders in the system and to better understand it with higher level requirements about actors and interactions. Intentional and social aspects of i* can be complemented with other modeling frameworks or languages to be even more pertinent during the requirements acquisition. (YU, Social Modeling and i*, 2009)

Software development is another domain in which the use of i* is very useful because it also consists in a complex social activity that involve a large number of persons to work on the project (with different roles, different specializations, different skills). Software development is often quite difficult to organize and manage due to its high variability. Social modeling allows to integrate some aspects to deal with cooperation and conflicts between actors. i* is here used to highlight the human social aspect of software processes. Different subsets of software development such as data warehousing, database design, or business intelligence can require the use of i*. (YU, Social Modeling and i*, 2009)

Social modeling and i* can also match with enterprise engineering. Since static and dynamic modeling ontologies only specify what happens, without explaining why it is happening, social models are needed. Therefore i* can be useful to link business processes to business goals taking into consideration goals from all the stakeholders. The SR model can support both reasoning about the process and the social configurations it involves. i* can be used as a strong base to build a reliable method for business processes and process reengineering. (YU, Social Modeling and i*, 2009)

Security, privacy, and trust are other aspects that i* can take into account. They represent software features that are becoming more and more primordial for a powerful software. Nevertheless, only few security models adopt a social perspective while it includes a lot of different agents. The concepts of (high-level) softgoals allow to model security, privacy, and trust in i*. More specific aspects as information confidentiality, integrity and availability can be modelled as refined goals. In addition to the security aspects, i* can also represent goals and strategies of the attackers, that can be analysed during the requirements acquisition and design. Risk modeling can also use social modeling like i* to build models and risks analysis. (YU, Social Modeling and i*, 2009)

3.5. Benefits and critics

iStar was developed in order to answer to the complexity of systems that the modern society generates. i* allows to integrate in its models new concepts to answer to the limitations of the traditional modeling languages.

First, i* incorporates complementary and competitive interests of the actors. The modeling language elaborated by Eric Yu takes into account all the interests and objectives together. Agents act strategically, dealing with opportunities and vulnerabilities. Moreover, the concept of softgoal introduced in i* allows to differentiate quality levels and make clearer the understanding of the system. Another benefit of the use of i* compared to the use of a traditional language is the incorporation of the intentionality. In traditional modeling languages it is the role of the analysts to monitor that, according to what they consider important for the system. While in i*, the agents' intentions are hidden by the SR model through the SD model.

Obviously i* includes some limitations:

- As i* is a social modeling, a big importance and interest are placed on the social concepts, i.e. the actors. The actor autonomy premise implies that actors are reasoning from their own perspective. Therefore, to model a system in i*, it would need as many SR models as there are actors in the system because each SR model represents a specific actor viewpoint. The modeling can then become very complex and large.
- There does not exist temporal concepts in i*. The models are static while business processes are always temporal, dynamic. Dynamic and static aspects are not integrated in i* social and intentional ontologies. When a business process is modelled in i*, only its social and intentional aspects are represented. From the same point of view, evolution and change dimensions are very limited in i*.
- Although Eric Yu developed i* as a framework, there is a lack of formal definitions of the concepts of i*. It is the reason for which we have considered it as a modeling language and not a real framework. However, i* can be used as a framework if we used the rules of i* but with a modeling language a little bit different from the concepts proposed by Eric Yu. For example, the concepts of actor, role and position are distinguished in i*, but their meanings are not well specified for interpretation.

It is important to notice that i* represents only one possible variation and approach of social modeling. There exist other techniques, frameworks or languages to model and analyse social aspects while i* is one of the most reliable. (YU, Social Modeling and i*, 2009)

3.6. Study Case

The recurrent study case of the thesis is obviously built in 2 models: a Strategic Dependency (SD) model that represents the interactions between different actors of the Expert Committee case, and a Strategic Rationale (SR) model that represents internal intentions of each actor. Pay attention that here only the SR model between *Reviewer* and *Chair* is exposed. (see figure 13 and 14)

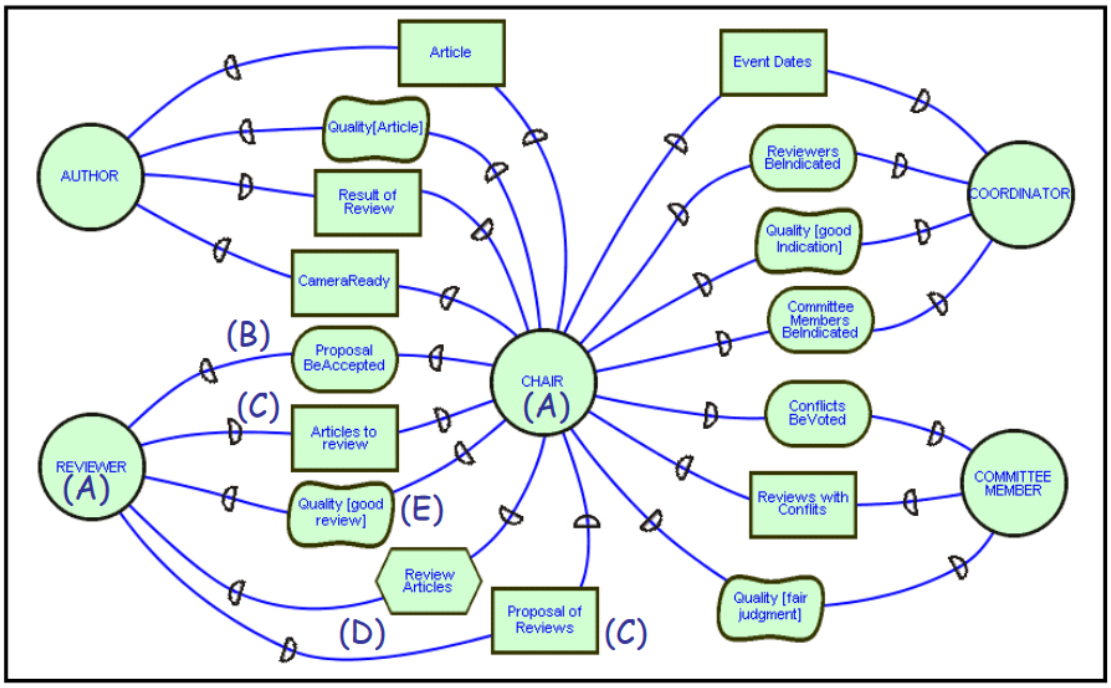


Figure 14: SD model of Expert Committee case (Oliveira, Cysneiros, Leite, Figueiredo, & Lucena, 2006)

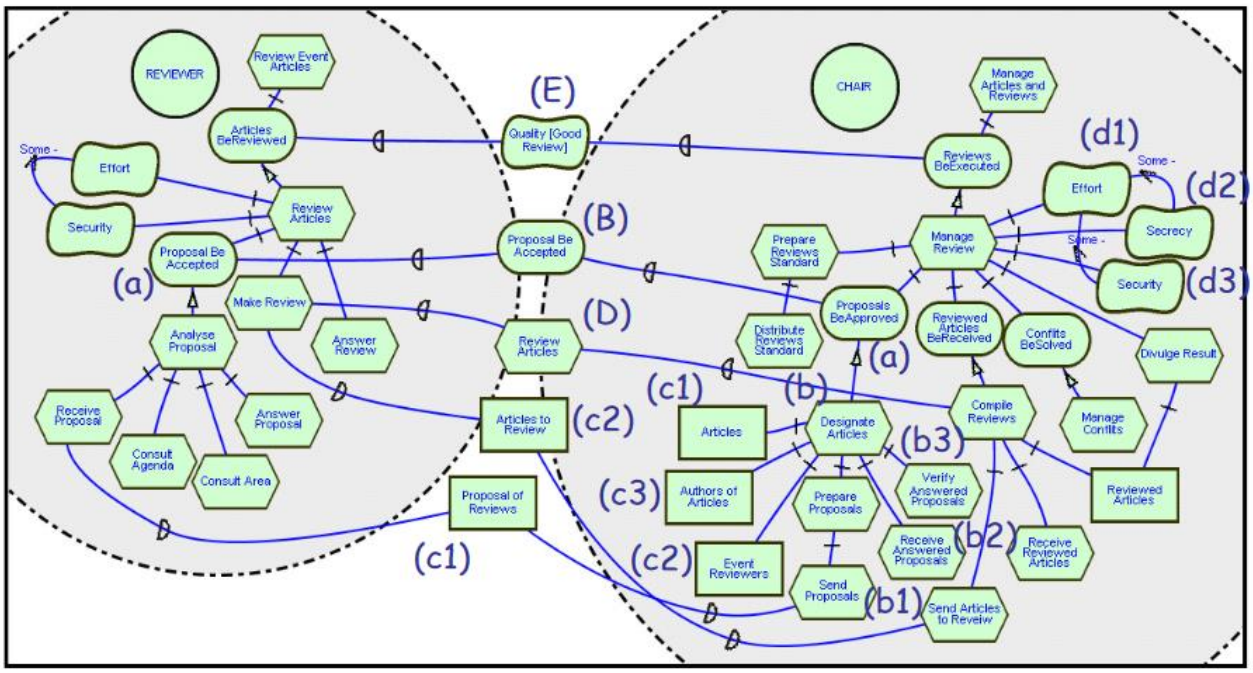


Figure 15: SR model of Expert Committee case between Reviewer and Chair (Oliveira, Cysneiros, Leite, Figueiredo, & Lucena, 2006)

4. Tropos

4.1. Description

Tropos is an agent-oriented methodology used to help in software development and was developed by John Mylopoulos (*et al.*) from University of Toronto (Canada) in the early 2000's. The origin of the name Tropos comes from a Greek word, which means "way of doing things" or which means "turn" or "change". Tropos was developed to response to the existing semantic gap between a system and its environment. Indeed, current information systems need to better match their operational organizational environment, but traditional methodologies are not very focused on the organizational dimensions but more on programming concepts. Therefore, Tropos aims to reduce as more as possible this gap with the proposed methodology (Castro, Kolp, & Mylopoulos, 2001). Tropos is a methodology that contains 4 main phases (see section 4.3.). Tropos is a methodology that finds its most important contribution in the Requirements Engineering. The methodology is based on the i* framework (of Eric Yu) and adopts different concepts like actor, goal and dependency and uses them to model early and late requirements. Although Tropos is an agent-oriented methodology, it also suggests some goal-oriented techniques for capturing and analysing the requirements of the system and the business but also risk, security, and location requirements. (The Tropos Crowd, s.d.) Tropos is a methodology that guides software development from early requirements to the detailed design of the software. It uses i* to model and reason about system requirements and different choices about system configuration choices. In addition to the basis from i* modeling language, Tropos also counts a formal language called Formal Tropos that aims to add constraints, invariants, conditions (pre, post) to capture more requirements and information about the context. (Lapouchnian, 2005)

4.2. Approach

As mentioned just above, Tropos is an agent-oriented software engineering methodology. It aims to cover entirely the software development process. Two main ideas are derived from Tropos (Giorgini, Kolp, Mylopoulos, & Pistore, 2004). The first one is the agent dimensions that is present throughout the software development (from the early phase of Requirements Engineering to the implementation), also considering intentional aspects like goals and plans of the agents. The second main idea in Tropos is the fact that the methodology (like i*) also considers the early phase of Requirements Engineering with a lot of attention in order to get a better understanding of the environment and the possible interactions between human actors and the software. While late-requirements allow to capture the what and the how for the system-to-be, early ones allows to get a reason, a why the system is developed. Therefore, system dependencies will support a refined analysis and a better handling of the functional and non-functional requirements. (Giorgini, Kolp, Mylopoulos, & Pistore, 2004)

But since Tropos adopts i* concepts in its methodology, it also contains a goal-oriented approach. It is important to remember that each software has its own context, and that there exists a strong link between the system requirements and the system context. First the context could be considered to determine all the requirements that are applicable to a system and the different alternatives that can be adopted to response to the requirements and the quality of each one. Moreover, the system itself can generate some changes in its context to response to its requirements. Consequently, it follows from that a mutual influence and relation between requirements and context because the context influence the goals of the users and their choices to reach them.

Capturing this mutual relation is crucial to get a software that answers to the users' requirements in every situation. Tropos suggested then a requirement engineering framework, goal-based, in order to model and reason requirements for system operating in different contexts. The suggested framework is composed of 3 different parts, such as a modelling language, a reasoning techniques and automated support tool and a methodological process. (The Tropos Crowd, s.d.)

4.3. Methodology steps

Tropos is a 4-phases methodology that includes Requirements Engineering (in general) and software design. (Giorgini, Kolp, Mylopoulos, & Pistore, 2004)

1) Early Requirements

That concerns the first understanding of the system and its environment/domain. To achieve this, it is important to analyse the problem by paying attention to the organizational settings. An organizational model will be derived from this first phase and will introduce the actors and their situations (intentions like their respective goals, relationships between different actors, etc.). Intentions of stakeholders are modelled as goals. A goal analysis can be a support for capturing both functional and non-functional requirements (Castro, Kolp, & Mylopoulos, 2001). In total, early requirements allows to get 2 diagrams: an actor-based one and a goal-based one. The second one is a refinement of the actor-based diagram with more focus on the goals but is built during late requirements.

The actor-based model is inspired by the Strategic Dependency (SD) model in i* (see section 3.3.1.). It aims to show the different interactions and relationships that each actor has with other. "A dependency describes an "agreement between a depending actor and an actor who is depended upon" (Giorgini, Kolp, Mylopoulos, & Pistore, 2004). There exist different types of dependencies and it depends on the nature of its agreement. As Tropos adopts i* modeling language in its framework, the i* models are sufficient to represent the model of this first step. On the other hand, the second model in Tropos, goal-based, can be represented by the Strategic Rationale (SR) model in i* (see section 3.3.2.) because this model aims to show the intentions of actors by describing why they interact with other actors. (Giorgini, Kolp, Mylopoulos, & Pistore, 2004)

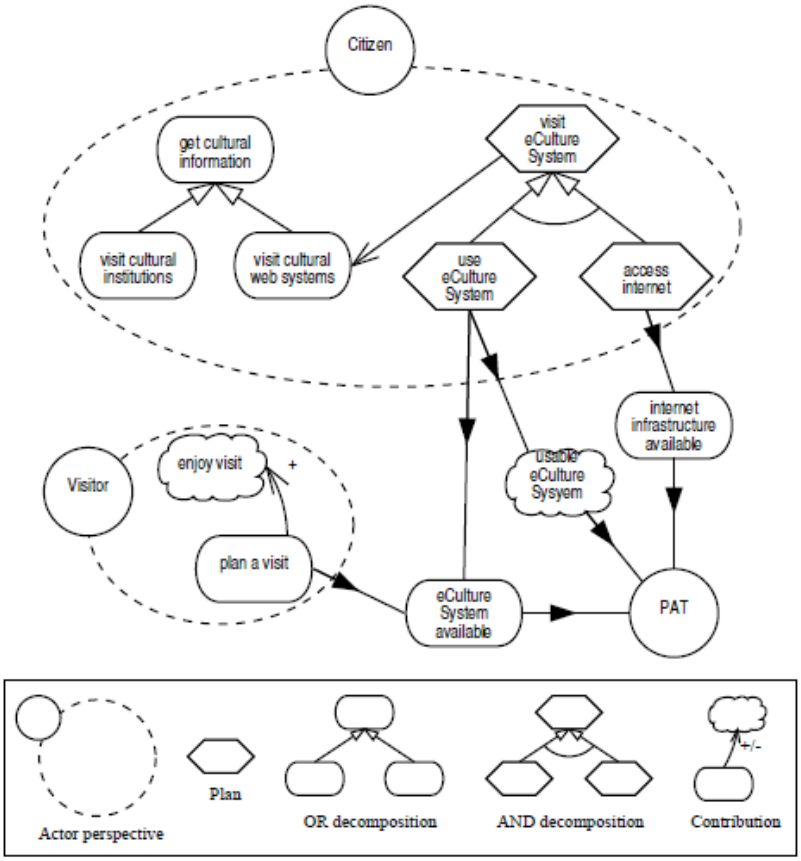


Figure 16: Early-requirements about e-Culture study case (University of Trento) (The Tropos Crowd, s.d.)

To capture more easily early requirements, there are some modeling activities to do. *Bresciani at al.* proposed a set of models to achieve it. It starts from the early requirements to their refinement and their evolution into models. In order, they are *actor modeling* (i.e. identify and analyse actors from the environment and the actors from the system), *dependency modeling* (i.e. identify the relationships between actors in terms of goal achieving, plan performing or resource providing), *goal modeling* (i.e. analysis of an actor goals, with means-ends/contribution and AND/OR analysis), *plan modeling* (i.e. an analysis method to complete goal modeling) *and capability modeling* (i.e. specify individual capacity for each actor). (Bresciani, Giorgini, Giunchiglia, Mylopoulos, & Perini, 2002)

2) Late Requirements

It consists in describing the system-to-be in its operational environment with its pertinent features and qualities. The system-to-be models an actor and the relationships he has with other actors in the organizational context. The functional and non-functional requirements are defined and clearly specified in this phase thanks to the system-to-be. Tropos represents a system as a set of 1 or more actors that have relationships with other actors in a SD model. It means that the system represents 1 or more actors who are contributing to the achievement of the stakeholders' goals. Tropos also allows to add some responsibility aspects in the system and to add it to some sub-actors after having decomposed the system. This decomposition and the assignment of responsibility to agents is made using means-ends analysis (Castro, Kolp, & Mylopoulos, 2001). It is during late requirements that the SR model in i* takes place to determine the link between the stakeholders' goals and the dependencies through which the actor expects that his goals will be fulfilled. (Giorgini, Kolp, Mylopoulos, & Pistore, 2004)

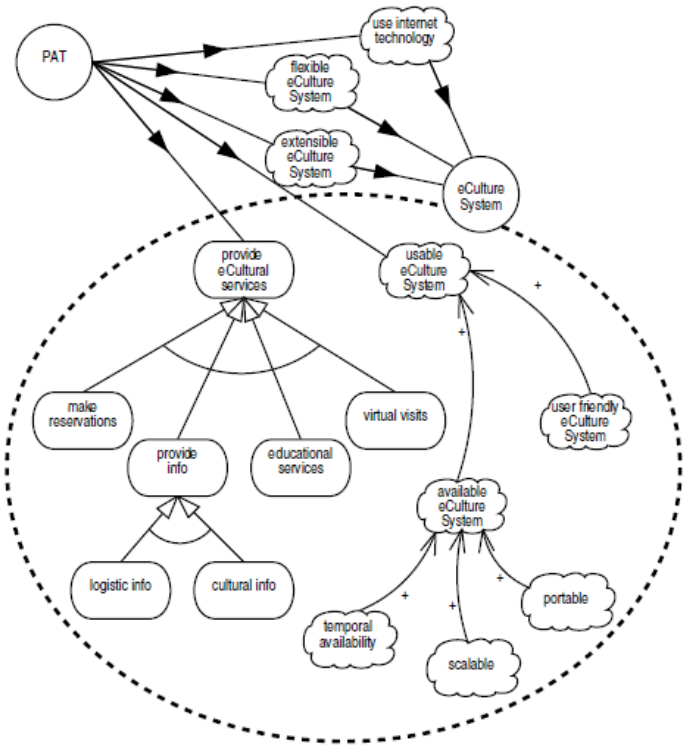


Figure 17: Late-requirements about e-Culture study case (University of Trento) (The Tropos Crowd, s.d.)

Figure 16 and *figure 17* describe respectively early and late-requirements of a e-culture system, derived from a study case from the University of Trento (Italy). We notice that in early-requirements, we got a general model showing the system as-is with the different actors and how they interact together. On the other hand, in the late-requirements model they have introduced the e-culture system as an actor in the to-be system. Requirements modeling contains then the 2 phases: early and late-requirements. As showed in the figure 16 and 17, the first one aims to depict the scenario as-is, and the other one aims to show the system-to-be and how social relationships happen in the system.

3) Architectural Design

The architectural design is the third step of the Tropos methodology. It is during this phase that the global system architecture is defined, through different subsystems interconnected by data, control and dependencies. This phase can be subdivided in 3 steps: (The Tropos Crowd, s.d.)

- The definition of the global architecture
- The identification of the capabilities, abilities and skills of the actors required to achieve their goals and plans
- The definition of agent types and assignment of 1 or 2 capabilities to each agent type.

The architecture of a system is obviously a very critical phase in software development. It corresponds to build a small and smartly manageable model of the system structure to describe how the system and its components will work together. Tropos allows to describe organizational architectural styles for different areas, namely for cooperative, dynamic, and distributed applications (like multi-agent applications). It is important to select these alternative architectural styles by respecting some criteria about qualities to get the desired system. These styles descriptions are used to guide the design of the expected system architecture and are based on several concepts and design alternatives that come from organization management area. It aims to be a support in

matching multi-agent system to its organizational context in which it will operate (Giorgini, Kolp, Mylopoulos, & Pistore, 2004). The analysis of the alternative architectural styles consists to refine qualities that are represented into soft goals to sub-goals that are more detailed and specific to evaluate each opportunity. (Castro, Kolp, & Mylopoulos, 2001)

4) Detailed Design

The detailed design of the system is the last phase. In this phase, we define in more detail the behaviour of each design components. Since each agent is specified at the micro-level, we need to specify in detail all their intentions (goals, beliefs, capabilities) and to show the different interactions between them. (The Tropos Crowd, s.d.)

Additional detail for architectural components of the system is added in this phase which allows to determine how the goals are achieved by each agent by using design patterns. But while more attention is given to design patterns, the literature gives priority to object-oriented patterns, compared to intentional and social ones which would be more relevant in the Tropos context. Tropos uses social patterns when it concerns finding a solution to a specific goal, that is defined at a design level with the identification of organizational styles and pertinent quality features. Moreover, detail design in Tropos also includes the step of agent communication specification and agent behaviour specification. (Giorgini, Kolp, Mylopoulos, & Pistore, 2004)

In order to support this last phase of Tropos methodology, *Castro at al.* suggest using existing agent communication languages that include message transportation mechanisms and other different tools and concepts. FIPA (Foundation for Intelligent Agents) have developed an extension of UML that is preconized by the authors (see figure 18). (Castro, Kolp, & Mylopoulos, 2001)

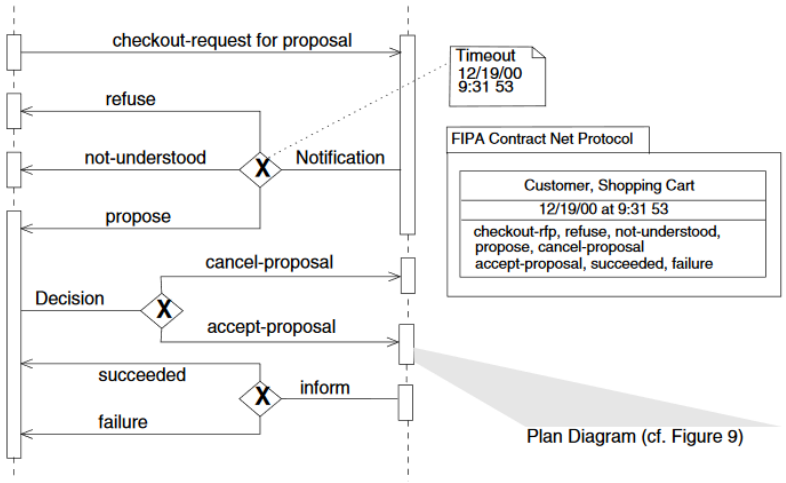


Figure 18: Example of an agent interaction protocol between Customer an Shopping Cart (Castro, Kolp, & Mylopoulos, 2001)

4.4. Formal Tropos language

As mentioned in section 4.1, Tropos includes a formal language called Formal Tropos that aims to add constraints, invariants, conditions (pre, post) to capture more requirements and information about the context. (Lapouchnian, 2005) Formal Tropos (*hereafter FT*) is a specification language that provides mentalistic notions of Tropos methodology and complements them with a rich temporal specification derived from KAOS. FT is the basis to analyse techniques for verification of requirements specification. Dynamic aspects of the Tropos models are described in FT language. In

FT, we emphasize not only on the intentional elements of the system but also on the situations and circumstances in which they happen, and the different conditions they need to achieve them. Consequently, it is possible to explain dynamic aspects of requirements specification at a strategic level, without needing an operationalization of the specification. FT allows then to give the opportunity to ask some questions like can we model valid operational scenarios, are goals achievable by actors, do dependencies represent a good relationship between actors, etc.

It is interesting to have this FT language because it allows to get relevant elements like goals, actors, dependencies and so forth of the system domain and also to describe relationships between them. The descriptions are made in 2 layers, an outer one and an inner one. The outer layer looks like a classic class declaration and associate some attributes to the elements to define them. The inner one is used to express some constraints about the object lifetime, considering a temporal logic. (Giorgini, Kolp, Mylopoulos, & Pistore, 2004)

An example of outer layer description is seen in Figure 19 which shows a case study about a Media Shop. We can notice that the managerial aspect is seriously considered in this layer. The figure shows intentional elements, actors and dependencies from SR model mapped into *classes*. FT uses *entities* to introduce the relevant non-intentional elements of the context. It is possible that several instances of a class are existing during the evolution of the system in software development. FT associates each class with a list of attributes, that are references to other classes and are used to specify which relationship takes place among different instances and classes. There exist different kinds of attributes, listed as:

- *Constant* attributes are used to represent static relations between class instances since it is "attributes whose values do not change with time".
- *Actor* attributes are used to link "a goal or a task to the corresponding actor".
- *Depender* and *Dependee* attributes are used to define which actor has which role in the dependency.
- *Mode* attributes are used for the intentional elements. They serve to specify the state of the achievement of a goal or a task. ***Achieve*** specifies that the corresponding actor wants to reach a state where the goal or the task is fulfilled. ***Maintain*** specifies that the goal or task need to be continuously maintain (e.g. for security).

```
Entity Item
Entity Cart
  Attribute items : set of Item
Actor Customer
Actor Medi@
Goal Dependency PlaceOrder
  Depender Customer
  Dependee Medi@
  Mode achieve
Task ShoppingCart
  Actor Media@
  Mode achieve
  Attribute constant cart : Cart
            constant po : PlaceOrder
```

Figure 19: Example of FT class declaration (Giorgini, Kolp, Mylopoulos, & Pistore, 2004)

Figure 20 shows an example of FT constraints that corresponds on the lifetime of class instances which defines the inner layer. The lifetime of intentional elements counts 2 critical events: their creation and their fulfilment. These 2 events are also characterized as constraints. As for the different kinds of attributes, FT exposes different constraints:

- *Invariant* constraints are used to define a condition that must be always true throughout all the lifetime of class instances. It is used to specify the relations on the value of an attribute or a cardinality constraint on an instance of a class.
- *Creation* and *Fulfilment* constraints are used in order to impose some new conditions for these 2 very critical moments in the intentional elements' lifecycle. The creation of a goal or a task corresponds to the fact that the corresponding actor (the depender or the owner) expects or desires its fulfilment. "*Creation* constraints should be satisfied whenever a new instance is created while *Fulfilment* constraints should hold whenever a goal or softgoal is satisfied, a task is performed, a resource is made available, or a dependum is delivered" (Giorgini, Kolp, Mylopoulos, & Pistore, 2004). It is possible to add more details to these 2 conditions with the use of keywords: **Trigger** means that the condition is sufficient, **Condition** means that the condition is necessary, and **Definition** means that the condition is necessary and sufficient. *Creation* and *fulfilment* constraints aims to link subordinate goals and tasks to their parent intentional elements.

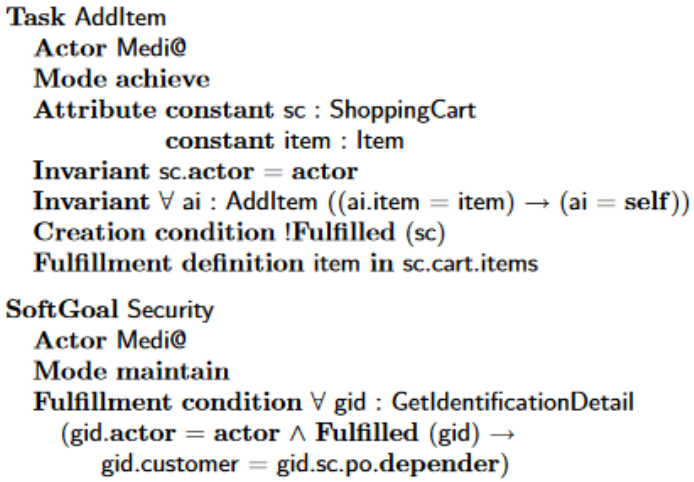


Figure 20: Example of FT constraints (Giorgini, Kolp, Mylopoulos, & Pistore, 2004)

Once FT class declaration and constraints are specified, we can formally verify them to identify possible errors, ambiguities, or under-specifications. It allows to get a feedback about FT specifications verification and to obtain some information to fix them. *Giorgini at al.* Have developed a (prototype) tool to support the verification process, called T-Tool and based on finite-state model checking. This tool provides a finite model showing all the possible behaviours of the context satisfying the specifications constraints. Afterwards, the T-Tool checks whether this finite model exhibits the expected behaviours. The T-Tool provides different checking functionalities: (Giorgini, Kolp, Mylopoulos, & Pistore, 2004)

- *Animation* of the specification corresponds to generate interactively a valid scenario (i.e. a scenario that respects all the possible temporal constraints in the FT specification). It allows

to get an intermediate feedback about the effects of the constraints and about a first identification of trivial errors and missing requirements.

- *Consistency checks* allow to verify that FT specifications are not self-contradictory. This phenomenon (inconsistent specifications) can occur due to complex relations among some constraints in the specification, and a complexity to detect it without using automated tools for analysis. During this check about the consistency of the specifications, the developed tool verifies that some valid scenarios exist well and that they respect well all the constraints, but also that each goal and dependency are achievable according to some scenarios and properties.
- *Possibility checks* allow to verify whether analysts have over-constrained the specifications, i.e. whether they have set aside the expected scenarios from stakeholders' point of view. **Possibility** properties are used in FT language to describe these expected scenarios. The possibility property should be associated to a scenario that we do not want to rule out.
- *Assertion properties* allow to verify whether requirements are under-specified or not, and whether they allow invalid scenarios. In FT, **assertion** declaration allows to express the conditions we want on a valid scenario. If assertion conditions are false, it would return an error because the condition which would be true are violated. This event happens during the verification phase of the specifications.

4.5. Modeling Concepts

As already explained above, since Tropos is an adaptation of the i* modeling language from Eric Yu, it also reuses the modeling concepts from i*, that are the Strategic Dependency (SD) model and the Strategic Rationale (SR) model. In addition to these common concepts with i*, Tropos also includes goal models. Tropos reuses then common concepts from i*, respectively **actor**, **goal** (and **subgoal**), **plan** (i.e. a way of doing something), **resource**, **dependency**, **capability**, and **belief**. (Bresciani, Giorgini, Giunchiglia, Mylopoulos, & Perini, 2002)

Traditional goal analysis models consist of a decomposition of a goal into subgoals. Each entity is related by AND or OR relationship. AND relationship means that all the subgoals must be satisfied to achieve the parent goal while OR relationship means that only one of the subgoals must be satisfied to fulfil the parent goal. But unfortunately, such a framework for goal analysis is not applicable in several useful domains due to a complex formalization of the goals, and a difficulty to well capture the possible relationships between them. The framework does also not work when there are contradictory goals or contradictory contributions to achieve a goal.

Therefore, Tropos suggests a formal model for goals analysis and specification that deals with qualitative relationships and possible inconsistencies among some goals. In the Tropos Goal Model, objectives can be modelled and analysed showing relationships among them (AND or OR relationships) but also positive or negative dimensions of the relationships. For example, we will use *++(G,G')* to represent that the satisfaction of the parent goal G implies satisfaction of subgoal G'. From this given goal graph, it is possible to reason from 2 different ways: Top-down and Bottom-up. Top-down reasoning means that we are more concentrated on a set of root goals that we want really to achieve with a certain assignment. The idea is to find an assignment to the leaf nodes consistent with the initial desired assignment. The objective is to propagate this initial assignment that allows to fulfil the leaf goals and contains some desiderata up to the root goals. Bottom-up reasoning means

that we are concentrated on a set of leaf goals with an assignment and that we want to propagate this assignment to higher-level goals. (Giorgini, Kolp, Mylopoulos, & Pistore, 2004)

4.6. Application areas of Tropos

As cited above, the main contribution of Tropos is about Requirements Engineering. However, it is not the only one area that uses efficiently the software development methodology because security and compliance took benefits in the use of Tropos. (The Tropos Crowd, s.d.)

Logically, security is taking more and more importance in the software systems development. Nevertheless, the definition of security requirements is generally considered after the conceptual aspect of the system. It is mainly because engineering methodologies do not integrate security issues throughout the developing phases. The current approach to incorporate the security dimension in a system consists in identifying the security software requirements after the definition of the system. This approach has consequently provoked the emergence of a lot of default (with security vulnerabilities) computer systems. According to the security paradigm, it would be possible to attenuate these problems thanks to a better incorporation of security and systems engineering. Tropos suggests then a modelling and analysis framework to take into account security issues at each step of software development. This framework contains several concepts as trust, ownership, and delegation to complete basic concepts, i.e. actor, goal, plan, and dependency. Moreover, Tropos offers a 3-steps framework for modelling and reasoning about risk and an extension of the modelling language i* takes care about norms and law aspects of a system.

- Norms and law: Human needs and environmental constraints must be carefully captured and translated into system requirements to ensure the reliability of the software. Tropos suggests here normative i* which is an approach that deals with challenges and issues linked to the requirements analysis in different kinds of environment.
- Risk: Very critical phases in system engineering are modelling and analysing risk. Tropos suggests then a goal-risk framework to answer to this issue. It allows to analyse risk at organization level and shares different techniques to help analysts to identify and enumerate efficient countermeasures to attenuate the risk in the system. The goal model of Tropos is extended by integrating 2 new entities, that are *event* (i.e. a risk, an opportunity, a vulnerability) and *treatment* (i.e. countermeasures, tasks, actions, etc).
- Security: *Secure Tropos* is the name of the Tropos extension intended for handling system security. It aims to model and analyse security requirements at the same time of functional requirements of the system. Secure Tropos provides a process for requirements analysing to guide the system engineers from requirements acquisition to their verification. There exist 2 different versions of Tropos. The first one is based on the modelling language self-i*, an extension of i*. Concepts like actor, goal, task, and resource are still common and are completed by a set of security-based concepts. The second version of Secure Tropos only extends the Tropos language and its development process. The language extension consists in defining again existing concepts with security in mind and in incorporating new concepts (like security constraints, security goals, security plans, etc.)

4.7. Limitations of Tropos

Tropos had been applied to some small case studies and got quite encouraging results. However, the methodology includes some limitations. First, there is a lack of tool to support the Tropos

methodology now available, and for the transition between the different phases of Tropos. Then, the methodology has not been used for developing full-fledged multi-agent system, so it is difficult to evaluate its power for this kind of system. (Giorgini, Kolp, Mylopoulos, & Pistore, 2004)

4.8. Study Case

For Tropos, I decided to adapt the case study already designed in i* and in KAOS, by declaring a part of the system in Formal Tropos. Because Tropos uses and is based on the SR and SD model from i*, it was not useless to build a new model. Therefore, the SR model – Reviewer and Chair (see figure 9) and mostly the Reviewer agent are formally declared here.

Actor Chair
Actor Reviewer

Entity Review
Entity Article

Dependency Proposal BeAccepted
    Type Goal
    Depender Chair
    Dependee Reviewer
    Mode Achieve
    Attribute constant proposal : Proposal
    Invariant ∃ proposal
    Fulfillment definition every proposals are accepted
Dependency Quality [good review]
    Type SoftGoal
    Depender Chair
    Dependee Reviewer
    Mode Maintain
    Creation definition ∃ article
    Fulfillment condition review & good review fulfilled
Task Review Articles
    Actor Reviewer
    Mode Achieve
    Attribute constant articles : set of articles
    Invariant ∀ ra : Review Articles (size.articles >= 1)
    Invariant ra = Effort Λ Security Λ Proposal BeAccepted Λ Make Review Λ Answer Review
    Creation condition ∃ article
    Fulfillment condition Articles BeReviewed created
SoftGoal Security
    Actor Reviewer
    Mode Maintain
    Fulfillment condition ra.fulfilled -> Articles BeRievewed
SoftGoal Effort
    Actor Reviewer
    Mode Maintain
    Fulfillment condition ra.fulfilled -> Articles BeRievewed

Goal Articles BeReviewed

Actor Reviewer

Attribute reviewed articles : set of reviewed articles

Mode Achieve

Invariant Need (Review Event Articles Λ Review Articles)

Fulfillment definition ∀ reviewed articles : reviewed articles **in** list of reviewed articles

Task Review Event Articles

Actor Reviewer

Mode Achieve

Attribute constant event articles : set of event articles

Invariant ∀ rea : Review Event Articles (size.eventarticles >= 1)

Invariant rea -> Articles BeReviewed

Fulfillment condition Articles BeReviewed created

Goal Proposal Be Accepted

Actor Reviewer

Attribute accepted proposal : set of accepted proposal

Mode Achieve

Invariant Need (Analyse proposal)

Creation definition AnalyseProposal.fulfilled

Fulfillment definition ∀ proposals : accepted proposals **in** list of accepted proposals

Task Make Review

Actor Reviewer

Mode Achieve

Attribute constant articles : set of articles

Invariant ∀ mr : Make Review (size.articles >= 1)

Fulfillment condition Review Articles created

Task Answer Review

Actor Reviewer

Mode Achieve

Invariant ∀ mr : Make Review (size.articles >= 1)

Fulfillment condition Review Articles created

Task Analyse Proposal

Actor Reviewer

Mode Achieve

Attribute constant proposals : set of proposals

Invariant ∀ ap : Analyse Proposal (size.proposals >= 1)

Invariant ap = (Receive Proposal Λ Consult Agenda Λ Consult Area Λ Answer Proposal)

Creation condition ∃ proposal

Fulfillment condition Proposal BeAccepted created

Task Receive Proposal

Actor Reviewer

Mode Achieve

Invariant ∀ rp : Receive Proposal (size.receivedproposal >= 1)

Fulfillment condition Proposal are received

Task Consult Agenda

Actor Reviewer

Mode Achieve

Fulfillment condition Agenda consulted

Task Consult Area

Actor Reviewer

Mode Achieve

Fulfillement condition Area consulted

Task Answer Proposal

Actor Reviewer

Mode Achieve

Fulfillment condition proposal answered

5. Z Notation

5.1. Description

Z is a language (but commonly called notation) created in 1980 by Jean-Raymond Abrial, that aims to make functional the specifications of a system by describing the behaviour of computing systems. Specifications capture critical properties of a system. They are a support for developers in the check task about the code. Indeed, the code of a program or a software can normally not violate the specifications. They also allow to ensure that the system is error-free. Over decade, Z Notation has evolved and allows now to identify a standard set of notations to capture essential features of a system.

Z (also named Zed) is not a programming or modeling language. Z is just a formal methods notation that aims to express the requirements of a computing system or software in terms of well-defined specification.

A formal method is a useful way to translate non-mathematical descriptions into a formal language that allows to describe functionalities of a system by using mathematics. After translating thanks to a formal method, requirements can be translated into precise and mathematical models in accordance with the model. It is a good way to avoid ambiguity in a model and to get the same interpretation between the clients and the developers about functionalities and features of the system. In a nutshell, using formal methods notation is a relevant way to match system's requirement.

Most of concepts, approach, and characteristics of the Z Notation that are explained hereunder come from the Z Notation reference manual. (Spivey & Abrial, 1992)

5.2. Approach

Z notation is a language to model requirements based on specifications system and that generates a specification document. A specification document is a document that contains interrelated passages from formal and mathematical text and informal prose explanation. The formal text corresponds to a sequence of paragraphs that describe and introduce schemas, global variables, and basic types of the specification. Each paragraph is built on the previous one. This method that consists to gradually build the vocabulary of the specification is called the *definition before use*.

Such an approach based on specifications allows users and clients to express their needs unambiguously and understandably. Programming languages are often difficult to understand for people who do not have knowledge and skills in this domain. Therefore, it is very useful to use formal method like Z notation to express specification in an understandable language to specify the parameters, functionalities, or features of a software.

While specifications take a big importance in Z, it is not the main approach of the notation. In contrast with KAOS, iStar or Tropos, Z Notation is neither a goal-oriented approach nor agent-oriented one. Z Notation is a model-based Notation. This specification language is based on set theory, predicate logic, and schema. A schema is built with a set of objects, linked together. Z is then the language that provides symbols and notations to create and describe schemas and to combine them. It is just a notation proving schema and symbol and not a method.

5.3. Modeling Concepts

Zed allows to describe a data base or a system by formalizing their specifications. To do this, we use "schemas", completed by the operations including in the Z notation. The schemas allow to show both a static view (i.e. the current state of the system) of the system and a dynamic one (i.e. describing the possible event that can occur in the system).

5.3.1. State-Space

The State-space schema is the first we create about the system. The first step is to choose a name for this schema, to complete the signature and the predicates. Predicates are properties over the signature. The signature, on its side, defines the words and the predicates define the chosen constraints on the variables.

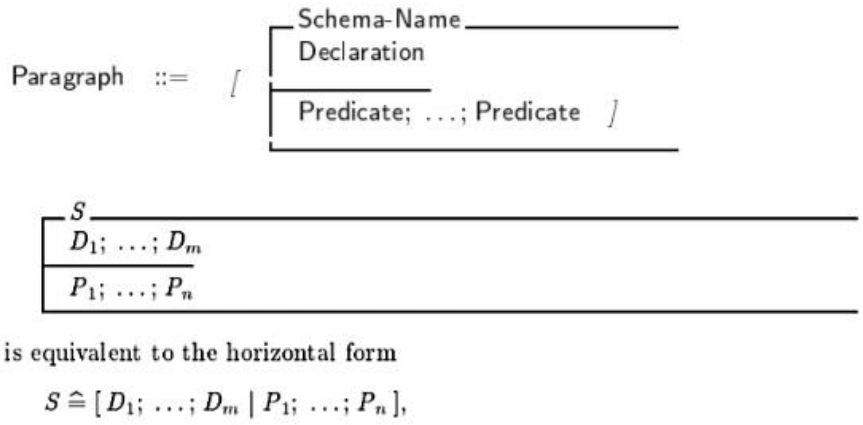


Figure 21: Schema definition (Spivey & Abrial, 1992)

A) Signature

Describing the signature is the first real step of the state-space schema. "The signature is a collection of variables, each with a type". It aims to declare the desired variables and assign a type to each of them. This step allows to define the terms that will be use in next parts. Once the variables are declared, it is time to determine their type. There exist different types, such as basic types, set types, cartesian product types or other schemas that are also types. It is also possible to have mathematical objects like relations or functions as type. The declarations are very important to ensure that all the specified expressions are well satisfying their assigned type. Nevertheless, it is important to distinguish those types with possible types in the programming language chosen to develop the software. The 4 existing types in Z Notation are:

- **Basic types**: This kind of type corresponds to the basic types generally known as an integer. Those basic types are individuals and do not have a very complex structure. Basic types can be any integer of Z (the set of integers) but it is also possible to define new basic types.
  To declare the type, we need to introduce it with square brackets. If the types can only have specific values, it is possible to enumerate them (it is also called ***free types***).
- **Sets**: A set is a collection of objects of the same type. Such collections are not ordered and cannot contain repeated objects. There exist many ways to declare this kind of type. First, we can enumerate all the elements of the set like {5,6,7,8,9}. The type of this set is PZ because it is a **power set** of Z. The other way to specify the sets is the use of properties that allow to

identify all the elements of the expected set by giving a property which is characteristics of the elements of the sets.

- **Tuples and Cartesian product types**: Tuples corresponds to the combination of different types. For example, we want to declare the new type DATE which is the combination of DAY, MONTH and YEAR. By doing the cartesian product of the 3 existing variables, we get a new one (DATE). The generated object is called a tuple.
  It is important to be careful with the brackets because if x, y and z are 3 objects that are members of the types t, u and v, their cartesian products can give different results.

  (x,y,z) is an object with type t X u X v

  ((x,y),z) is an object of type (t X u) X v

  Cartesian product types contain a disadvantage which is that it is not possible to put constraints between components. Therefore, other types, called **schema**, can support the expression of constraints because they include the predicate part that imposes some constraints on the components.

Z Notation also often use mathematical types that are defined thanks to the 4 fundamental types (i.e. basic, set, tuples and cartesian product, and schema).

- **Binary relation**: $X \leftrightarrow Y$ is a binary relation because it represents all the sets of the binary relation between the sets X and Y. Such a relation has a type P(X x Y) because it implies having a collection of tuples that indicates which element is related with which other element.
  Binary relations can be represented by 3 different ways:
  We can use a set of tuples: {(Name1, Address1), (Name2, Address2), …}
  We can use a maplet arrow: {Name1 → Address1, Name2 → Address2, …}
  We can use a function: Address (Name1) = Address1
- **Functions**: Functions are relations with specific properties. They allow to link each elements of X to at most one element of Y. There exist different functions:
  - *Partial* vs *Total* function: Partial functions are relations explaining "which relate each element x of the set X to at most one element of Y.
    "$\longrightarrow$" is a generic symbol used in Z Notation to represent a total function, that is a relation which relates all the element x of the set X to exactly one element of Y.
  - *Injections* vs *surjections*: The surjections are functions in which each element y of Y is related to at least an element of X (an element y can be related to many elements x). The symbol to represent a surjection is "$\twoheadrightarrow$". The injections are functions in which all the elements of Y are related to at most one element of X. The symbol to represent an injection is "$\rightarrowtail$"
  - Bijections: Bijections are function in which all the elements of Y are related to exactly one element of X. The symbol to represent a bijection is "$\rightarrowtail\!\!\!\!\twoheadrightarrow$".

Signature can also contain some properties. For example, when a new Schema type is defined. Take DATE for example, it is possible to mention that the variable day is at most 31.

B) Predicates

Predicates are used to ensure that properties mentioned in the signature part of the specification are true.

First, predicates have a domain and a range. Z Notation contains several basic functions that allow to capture the needed information. Domain (named 'dom') and range (named 'ran') functions are simplest examples. If R is a relation of type X↔Y, then the domain of R is the set of elements in X related to elements in Y and the range are all the elements of Y that X can be associated to.

But there exist many other symbols of predicates such as quantifiers (i.e. ∀,∃,$\exists_1$) and proposal connectives (i.e. ¬,Λ,V,→,↔).

Proposal connectives:

In the following tables, we will remind how work the proposal logic with the negation, the conjunction, the disjunction, the implication, and the equivalence. We will take for each of them at most 2 propositions, such as P and Q. In the results tables, T means it is True and F means it is False.

- ¬: Negation (NOT)

p	¬p	¬(¬p)
T	F	T
F	T	F

- Λ: AND Conjunction

  pΛq is true only when both p and q are true at the same time

p	q	pΛq
T	T	T
T	F	F
F	T	F
F	F	F

- V : OR Disjunction

  pVq is true when either p or q are true, or when both p and q are true at the same time

p	q	pVq
T	T	T
T	F	T
F	T	T
F	F	F

- →: Implication (IMPLIES)

  p→q is true when q is true or when p is false

p	q	p→q
T	T	T
T	F	F
F	T	T
F	F	T

- ↔: Equivalence (IF AND ONLY IF)
  p↔q is true when p and q have the same sign

p	q	p↔q
T	T	T
T	F	F
F	T	F
F	F	T

Quantifiers

There exist 3 different kinds of quantifiers in Z Notation:

- ∀ is the universal schema quantifier
  It means "every", for example ∀ dog is an animal
- ∃ is the existential schema quantifier
  It means "there exist", for example ∃ people born on 1st January 2000 in Belgium
- $\exists_1$ is the unique schema quantifier
  It means "there exist only one", for example $\exists_1$ people named Alexandre Amand and born in December 1997 (it is me)

Other symbols

Different other symbols can be used in Z Notation in the predicate part. For example, the equality or the membership are usable to express specification. Hereunder is an example with a DATE situation. It allows to add constraints and to better specify properties of variables.

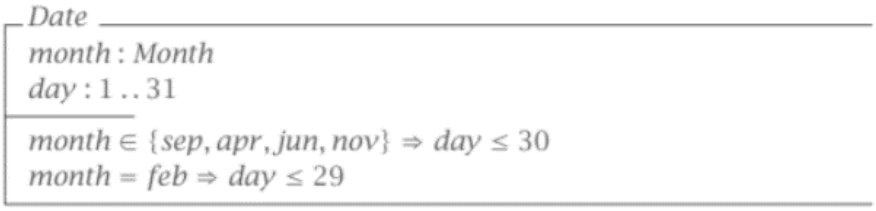


Figure 22:Example of constrained predicate

Invariants

Schemas include initial properties that must be respected, they are called *invariants*. That means that once the properties of a schema are defined, they need to be true even later when they are reused. State of database or system can vary over time. Therefore, we need to be careful that for any situation or operation, the initial properties are still satisfied.

Function and relation symbols

There exist many function and relation symbols in Z Notation. These symbols can appear before, after or between 2 arguments. Therefore, the symbols are respectively named prefix, postfix and infix.

A *postfix* is for example the relational inversion. It consists of an inversion of a homogenous relation and it is showed with the symbol '~' just after the relation. Relations are sets of ordered pairs so it gives the following example: (1,2)~ is equal to (2,1).

An *infix* is for example the "greater than" symbol (>). This relation means that we can link an element from a set to many elements of corresponding set (e.g. 2>1). The addition or subtraction symbols are then infix symbols.

A *prefix* is for example the minimum and maximum functions. We put them at the beginning of the operation.

Other symbols are various, but the more basic ones are inequality, non-membership, empty sets, subset, set union, et intersection and so one.

Generics

Generics are used every time we want to define a new symbol or when we need to have a look at the definition of a symbol. Generics corresponds to definitions of symbols in mathematical terms. Generics schema are represented with a double line on the top.

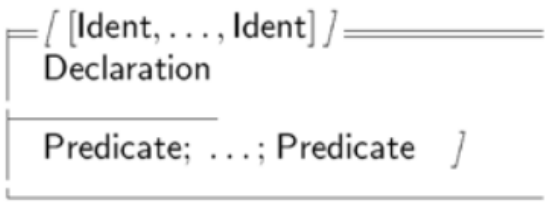


Figure 23: Generic constants (Spivey & Abrial, 1992)

Axiomatic global variable

Global variables are declared outside any schema and can be used in defining schemas. To use global variable as part of the mathematical library, a specification will often introduce global variables of its own. The scope of the components of this schema extends from their declaration to the end of the specification. The part into square brackets indicates that the dividing line and the predicate below are optional. In this way, declaration is an acceptable form of axiomatic global variable description. If the predicate part is absent, as default the predicate is true.

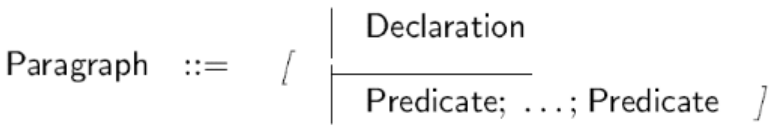


Figure 24: Axiomatic descriptions (Spivey & Abrial, 1992)

5.3.2. Operation

Abstract data types are defined in the state-space schema in Z Notation. The second part of schema allow to explain the operations that we will do on those data types. As mentioned in the Z Reference Manual, "each operation has certain input and output variables and is specified by a relationship between". So, it is important to identify the input and the output. Operations are described as a pair

of states, namely the state before execution (State) of the operation and the state afterwards (State').

A) Signature

First, we need to know if the operations we would like to specify will change the state of our binding or not. The binding is the state-space that gathers all the possible states (i.e. the instances). In Z Notation, we use the symbols Δ to specify if the operation change the state and the symbol Ξ if the operation does not change the state.

So, the Δ describes that we have 2 copies of the state: State before the operation and the state after the operation, named State'. Furthermore, with such an expression the predicates (i.e. the invariants) in the state-space, that were true before the operation, must be still true after the operation. It is the same course of action with the symbol Ξ.

The operations are also characterized by inputs and outputs. For example, for some features of the system, we need to ask for different elements. When the operation only changes the state of the binding, there is no output. The inputs are specified with the symbol '?' after the element in the schema, and the outputs are specified with the symbol '!' after the element in the schema. For example, take a situation where we have initially a birthday book as database. We want to add an operation to help to find a birthday in the database. See the following schema (figure 25) for the operation FindBirthday where the input is the name and the output is the date. Here the BirthdayBook will not be changed by the operation, consequently it takes the symbol Ξ.

B) Predicates

Z Notation contains an important part of mathematical definitions with its standard library. Those mathematical definitions allow to explain basic operations of algebra. Many of these operations have a strong connection with the subset ordering. In fact, among them, we have empty sets (∅), subset relation (⊆), proper subset relation (⊂), set union (U), set intersection (∩) and set difference (\).

We can take the example of the operation FindBirthday mentioned just above.

$$
\begin{array}{|l}
\hline
\textit{FindBirthday} \\
\Xi BirthdayBook \\
name? : NAME \\
date! : DATE \\
\hline
name? \in known \\
date! = birthday(name?) \\
\hline
\end{array}
$$

Figure 25: Signature and predicates of an operation schema (Spivey & Abrial, 1992)

We can now think about the predicate part of this operation schema. The first line is the precondition to ensure the link between the two parts and to show what is the input we want to know in the database. Then, to complete the specification, we take the input number we receive as outcome and we make corresponding it to the name we put also in input (and which is associated to this phone number).

Another existing operation symbols in Z Notation is the overriding (⊕). It means that the relation Q⊕R relates everything in the domain of R to the same objects as R does, and everything else in the domain of Q to the same objects as Q does.

5.3.3. How to handle errors

Operations schema described in the previous part are not robust enough. In fact, we are ignoring what it will happen if an error occurs. An error could be for example the fact that an input in incorrect or unknown by the database. Therefore, we need to specify the behaviour of the system for such situations, that is an error message, a bug warning, a false or fake output, etc. This part of Z Notation aims to strengthen the specifications of the operations by describing all the events that can happen.

There exists then different solution to answer to this lack of robustness.

- ***Report***: It consists to preview the error with a message. It is a free type that we have to define. The message can take 3 different values: Ok, Variable already known, or Variable not known. If the input variable is already known, we do not change the state of the database and we return the message 'variable already known'. If the operation works successfully, we return the message 'ok'. We can see examples in the following figures.

$$
\begin{array}{|l}
\hline
AlreadyKnown \\
\Xi BirthdayBook \\
name? : NAME \\
result! : REPORT \\
\hline
name? \in known \\
result! = already\_known \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
NotKnown \\
\Xi BirthdayBook \\
name? : NAME \\
result! : REPORT \\
\hline
name? \notin known \\
result! = not\_known \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
Success \\
result! : REPORT \\
\hline
result! = ok \\
\hline
\end{array}
$$

Figure 26: Handling errors schmas (Spivey & Abrial, 1992)

- ***Combination of schema***: Schema combination allows to gather the different operators useful to combine schemas. Combining schemas allows to extend the specification of an operation schema to explain how an error is handled. After having define the operation without considering the errors, we must define the errors and combine them with the initial operation. For example, we can combine errors and FindBirthday operation as follow:

$$RFindBirthday \mathrel{\widehat=} (FindBirthday \land Success) \lor NotKnown.$$

Figure 27: Robust operation FindBirthday with combination of schema (Spivey & Abrial, 1992)

It allows to get robust operation because we put together separate parts of specification corresponding to normal operation and error handling. However, combining schemas is not so easy because it needs to ensure that the schemas are type compatible. It means that the variables they have in common have the same types in both signatures. If the 2 schemas do not have variables in common, they are always type compatible. However, if we combine 2 schemas that have variables in common, they mandatory need to be of the same type to ensure the type compatible of the operation.
In order to combine 2 type compatible schemas, we can use other logical connectives such as ∧(conjunction), ∨(disjunction), ⇔(equivalence), ⟹(implication). The logical negation can also be used to combine schemas, like the schema hiding (\) and the schema projection (↾) (i.e. that we hide all the components from the first schema except those that are in common with the second one).

5.4. Main usage of Z

Thanks to its static and non-procedural approach, Z Notation has a wide range of applications in the IT field. In order to prevent bugs and malfunctions, it first of all makes it possible to express a behaviour or a property to be adopted according to specific requests. This is not about interpretation, but about the application of rigorous processes based on set theory.

Concretely, the Z notation is used for description and computer modelling in the broad sense of the term. This works for software as well as an operating system or any other program. As mentioned previously, its universal aspect ensures functionalities adaptable to many development environments. For example, such a structure allows the management of dictionary systems (see the birthday book example). It is thus possible to add, delete, search, or replace data easily. We can therefore define these sets of information on a construction or enumeration basis. (JDN, 2020)

5.5. Benefits and critics

The use of formal method is encouraged when the requirements are clearly stated and that it is nearly impossible to have misunderstanding. Consequently, the requirements must be complete, unambiguous, verifiable, precise, concise, and consistent. If all these constraints are satisfied, the models will be used for the implementation of the system and for formal verification. Mathematical proof that the implementation meets the specification under all circumstances have to be verified.

Another advantage of the use of Z Notation is the error handling aspect. Indeed, all the previous languages or frameworks do not allow to directly handle errors in the models. It allows to gain time and gain in unambiguity for the system and its future implementation.

Nevertheless, a formal method like Z Notation also involves some inconvenients. The first one is the mathematical skills needed to well understand all the schema. It is very important to have some skills to define, specify and explain the different schemas, the different operations, the signatures, the predicates and so one. Another inconvenient is the development and verification cost of the method which is very costly. A last important disadvantage of Z Notation is the difficulties we are dealing with

when we must represent complex problems. The formal method requires so much detail and understanding of the problem that it is often difficult to model complex issues.

5.6. Study Case

The Expert Committee study case is developed in Z in the following figures. Since the study case include lots of operations, just the 3 main ones were modelled here to show how work the Z Notation.

Initialisation :

Init

ChairDB ReviewDB PaperDB
paper = ∅ proposal = ∅ review = ∅

Figure 28: Z application of Expert Committee - Initialization

State-space:

ChairDB

chair : ℙ Chair proposal : Chair ⇻ Proposal
chair = dom proposal ∃1 chair

ReviewDB

reviewer : ℙ Reviewer review : Reviewer ⇻ Review
reviewer = dom review reviewer ∈ {reviewer1, reviewer2, ...}

PaperDB

author : ℙ Author paper : Author ⇻ Paper
author = dom paper author ∈ {author1, author2, ...}

Figure 29: Z application of Expert Committee - State-space schemas

Operations :

WritePaper

Δ PaperDB
author? :Author
text? : Text
paper! : Paper

$text? \notin Paper$
$paper' = paper \cup \{author? \mapsto text?\}$
$paper! = text?\ (author?)$

MakeProposal

Ξ PaperDB
Δ ChairDB
chair? : Chair
paper ?: Paper
proposal! : Proposal

$paper? \in Paper$
$proposal' = Proposal \cup \{chair? \mapsto paper?\}$

MakeReview

Ξ ChairDB
Ξ PaperDB
Δ ReviewDB
reviewer? : Reviewer
paper? : Paper
review! : Review

$paper? \in (Paper \cap Proposal)$
$review! \notin Review$
$review' = Review \cup \{reviewer? \mapsto paper?\}$

Figure 30: Z application of Expert Committee - Operations schemas

Errors handling :

Ok

report! : *MESSAGE*

---

report! = 'Ok'

AlreadyReviewed

Ξ ReviewDB
review? : Review
report! : *MESSAGE*

$review? \in Review$
report! = 'Paper already reviewed'

Figure 31: Z application of Expert Committee - Errors handling schemas

6. BPMN

6.1. Description

BPMN stands for **B**usiness **P**rocess **M**odel and **N**otation and is a graphical representation supporting business process specifications in a business process model. BPMN had initially been developed in 2004 by BPMI (Business Process Management Initiative) and is now maintained by the OMG (Object Management Group) since 2005. The current version is BPMN 2.0, released in January 2011. It is at this time that the name turned into Business Process model *and* notation, in order to show the introduction of execution semantics. (OMG, 2011) The idea was to create an expressive and quite formal modelling language for business processes, that would be easy to understand for final users but also by domain experts faced out. (Chinosi & Trombetta, 2012)

The first idea about the creation of BPMN was providing a clear and readily understandable notation for all the business users, from the analysts who capture the specifications to the developers who are in charge of the system implementation, but also by integrating in the process the future users who will use and maintain the system/software. BPMN is then very useful to reduce the existing gap between the business process architecture and the process implementation. (White, 2004)

6.2. Approach

BPMN is based on flowcharting technique that consists to model process by building graphical models of business process operations. It means that a BPMN model is a network composing by graphical objects (i.e. activities) and flow controls that aim to define the order of the tasks, and how they are organized and structured (White, 2004). The first and main goal of BPMN is to provide to business users, from business analysts to technical developers, a readily understandable notation. (Chinosi & Trombetta, 2012)

Documentation of complex procedures and use cases descriptions are usually difficult to understand and contain some errors. BPMN, with its formal and simple notation provide then a way to model them easier. Indeed, modellers want to enrich the descriptions of models thanks to diagrams to be sure to that the users and readers understand all the opportunities of a process. BPMN allows to do that and response to the wish of domain experts, interested in finding a method to study the properties of business processes illustrated by graphical elements. It is a support to verify that the representations are correct, that there is no interrupting condition, no deadlock, no infinite loop and so forth in the process. On the other way, the analysts prefer get information about data, what are the inputs, the outputs, when are they created in the process. BPMN is also a solution to this lack of information generated by traditional modelling notations or methodologies. (Chinosi & Trombetta, 2012)

Business process modelling allows to share and communicate a large variety of information to different kinds of audiences. BPMN is then designed to response to the creation of end-to-end business processes, from different levels of fidelity. Two basic types of business models can be built thanks to BPMN, namely collaborative (i.e. public) B2B processes and internal (i.e. private) business processes. (White, 2004)

Collaborative B2B processes consist of the interactions between minimum 2 business entities. Such diagrams are usually represented from a global point of view. That means that the modeller does not put himself in the shoes of one of the participants, but he shows the global links and interactions

between the participants. Interactions correspond to a flow of activities and exchange between the participants. A collaborative B2B process should only show what is visible for the public and will not consider internal activities. (White, 2004)

From the other side, internal business processes are usually more focused on a single business organization viewpoint. It allows to model internal activities generally not visible to the public. If swimlanes are used, external participants of the business process will often be modelled as black boxes and the emphasis is put on the internal and detailed business process. Message flow will be used between 2 separate pools and sequence flow will be used to relate activities in the same pool. Since BPMN allows to represent different levels of precision, the focus is first put on the higher-level process and afterwards lower levels processes are represented with sub-processes and other internal markers to add details on the diagram. The choice of the degree of precision depends on the modellers decisions because BPMN is a notation and is then completely independent of any process modelling methodology. (White, 2004)

Chinosi and Trombetta also declared that BPMN could model a third type of business processes, namely abstract processes. An abstract business process is a process containing interactions between participants who are the resource that performs activities present in the workflow. (Chinosi & Trombetta, 2012)

BPMN is not a pure graphical notation while it consists of 3 basic shapes with some extensions. In total, BPMN counts around 52 graphical elements but has a formally defined execution semantics. BPMN, thanks to several concepts like sub-processes, expand or collapse, is also able to handle complexity and contains many invisible attributes.

BPMN can be observed from 3 different levels, according to the Bruce Silvers's method. The first level is named *Descriptive BPMN*. It consists of limited set of symbols to hierarchically model the business process. In this first level, the focus is put on the understanding and the handling of complex real-world processes, considering the business aspect. The second level is named *Analytical BPMN*. It consists of a full palette of symbols that incorporates events and exceptions handling. The consistency and technical meaning of shapes is carefully considering here. The focus is put on the link between business part and IT part with a refinement of level 1. The third and last level is named *Executable BPMN* and corresponds to the XML language to design the executable process. (Silvers, 2009)

6.3. Modeling Concepts

BPMN is a modelling notation. It means that it provides a set of symbols and basic concepts with explanation about the way to use them. In this section, all the basics of BPMN are showed and described. It means that it contains the symbols and a description about how they work together in a diagram. BPMN allows to define a Business Process Diagram (BPD) which is made up of a set of graphical elements. Elements in BPMN are chosen to be distinct enough to not confuse them. They are made from different shapes, with different symbols inside and are easy to understand for most of analysts, even the less experienced ones.
The graphical elements of BPMN can be divided into 4 specific categories to help the reader of a model to understand it easily. The four basic categories of graphical elements are (White, 2004):

- Flow objects

- Connecting objects
- Swimlanes
- Artifacts

An advantage of BPMN is that modellers can create easily understandable diagrams with only the core elements and the connectors, with a low level of precision about the documentation and the communication purposes. To get higher level of specifications, modellers should use detailed graphical elements, such as the specific objects with a type on events or tasks for example (i.e. graphical elements detailed by internal markers). Diagrams with high level of precision are more often built when the model need to be carefully analysed.

BPMN is a notation that is often reviewed and extended to answer to issues that analysts and developers are dealing with. BPMN 2.0, the newest major version of the notation has extended its scope in different areas. First, it formalizes the execution semantics for all the existing BPMN elements and defines an extensible mechanism for graphical and process model extensions. Moreover, BPMN 2.0 refines event composition and correlations, and extends the definition of interactions between humans. The new version defines 2 new kind of models, respectively Choreography and Conversation models, that allows to better model interactions. Some known inconsistencies and ambiguities from the previous version are also be resolved. BPMN 2.0 contains some new constructs and gives more importance to data because data is not more just an artifact but has its own element category including data input or output, collection data objects, data store and messages. All these modifications and extensions show that BPMN is the reference when we talk about business process modelling. The continuous interest about the notation allows BPMN to be always reliable and powerful to capture specifications and model them. (Chinosi & Trombetta, 2012)

6.3.1. Flow objects

The flow objects correspond to the core elements of a BPD, and they are essential for a modeler to build a powerful and reliable diagram. In this first category of graphical elements, we can distinct 3 types of flow objects.

A) Event

Events are represented by a circle and means that something is happening in the process flow. They can affect the flow of the process and they generally have a trigger (i.e. a cause) and a result (i.e. an impact). Circles are open but they can contain different possible symbols inside to differentiate the trigger or the result. There exist 3 types of events, such as the start event (represented by a thin (green) circle), the intermediate circle (represented by a double circle) and the end event (represented by a thick (red) circle).

Figure 32: Representations of events in BPMN (White, 2004)

B) Activity

Activities are represented by rounded-corner rectangles and corresponds to a work made by the company, of one of its member or department (i.e. a task). Activities can be either atomic or non-atomic (i.e. compound). It means that activities can be either a task or a sub-process that is differentiated by a plus sign in the bottom center of the rounded-corner shape.

Figure 33: Representation of a task in BPMN (White, 2004)

C) Gateway

Gateways are represented by diamond shapes, as in several other notations. They aim to control the flow of the business process because they are used either for convergence or divergence. It means that gateways are used to represent traditional decisions made in a sequence flow. There exist different kinds of decisions that we can take, such as the forking, the merging, or the joining of paths. Like for the events, it is possible to add details on these graphical elements thanks to the use of internal markers to indicate the type of behaviour control.

It is also possible to represented higher level specifications by using event-based gateways. It means that the gateway is depending on an event and this detail is modelled by adding the corresponding kind of event in the diamond shape.

Figure 34: Representation of a Gateway in BPMN (White, 2004)

6.3.2. Connecting objects

Connecting objects are used to link mainly flow objects together, but also the other graphical elements like swimlanes and artifacts. They allow to structure the business process model. There exist 3 types of connecting objects.

A) Sequence flow

Represented by a solid line with a sold arrowhead, a sequence flow is used to represent the order (i.e. the sequence) of the performed activities in the business process.

Figure 35: Representation of a sequence flow in BPMN (White, 2004)

B) Message flow

Represented by a dashed line with an empty arrowhead, a message flow is used to represent the flow of messages between 2 different participants of the process that can be either business entities or business roles. It means that one sends a message, and another receives it. In BPMN, 2 different process participants are represented by 2 separated pools in the diagram. Message flows are also often used between 2 events from message, signal, or timer type.

Figure 36: Representation of a message flow in BPMN (White, 2004)

C) Association

Represented by a dotted line with a line arrowhead, an association is used to link data, text, or other artifacts (see section 6.3.4.) with a specific flow object. Association links allow to show what is the input and what is the output of an activity.

Figure 37: Representation of an association in BPMN (White, 2004)

6.3.3. Swimlanes

In BPMN and in other process modelling methodologies, the concept of swimlanes is used as a mechanism to organize and structure the activities into distinct visual categories. These categories aim to represent the different participants in the business process and contain activities and tasks of each one. It allows to represent different functional capabilities or responsibilities of an actor in the process. About this concept of swimlanes, BPMN counts 2 different constructs.

A) Pool

A pool illustrates a participant of the business process. It is represented as a graphical container to separate a set of activities from different pools. When a diagram involves different business entities or participants, we will use pools. It is important to mention that activities from separate pools are considered as self-contained processes. It is then impossible for a sequence flow to cross the boundaries of a pool. As mentioned in the previous section, message flows are used to represent and illustrate communication links between separate pools (i.e. separate participants).

Figure 38: Representation of a pool in BPMN (White, 2004)

B) Lane

Lanes are sub-partitions with a pool. It allows to extend the size of the pool (vertically or horizontally). Lanes are used to organize and separate into different categories the activities. In other words, if a pool corresponds to a department or a service of a company, a lane could be a manager or an employee of this department. Lanes are used to differentiate tasks associated with a specific role of a company. In lanes, sequence flow may cross the boundaries, but message flows cannot be used between different flow objects in lanes from a same pool.

Figure 39: Representation of lanes in BPMN (White, 2004)

6.3.4. Artifacts

BPMN had been developed to allow modelers to have more freedom and flexibility in the business process design with some extensions compared to other basic process notations. BPMN offers then the possibility to add detail and context about a specific situation of a system. Symbols from BPMN that allow that are named artifacts. They serve to add some details on a diagram about the context of the business process to model. It is possible for modelers to create their own types of artifacts in order to add the level of details they want on the diagram. Though some artifacts can be added, this will not affect the flow of the activities, gateways, or sequence flows. There exist 3 different main types of artifacts.

A) Data object

It consists of a mechanism to represent how and when data is required or generated by process activities. They are then linked to activities through associations.

Figure 40: Representation of a data object in BPMN (White, 2004)

B) Group

Represented by a rounded corner rectangle drawn with dashed lines, a group is used to document or analyse a purpose. However, it cannot affect the sequence flow of a business process.

Figure 41: Representation of a group in BPMN (White, 2004)

C) Annotation

Corresponding to a mechanism to provide additional text and annotation from the modeller destined to the reader. In other words, annotations are used when the modeller want to add some details about the process or want to make sur that the reader will well understand the diagram, without any ambiguity.

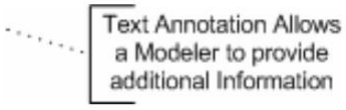


Figure 42: Representation of an annotation in BPMN (White, 2004)

All the graphical elements explained just above correspond to the core elements of BPMN and allow to model specifications with a lower level of precision. To get a higher level of precision, it is important to use all the possible elements provided in BPMN, and overall, the internal markers for events, gateways, and activities. *Figure 43* is a summary of all the BPMN 1.2 elements.

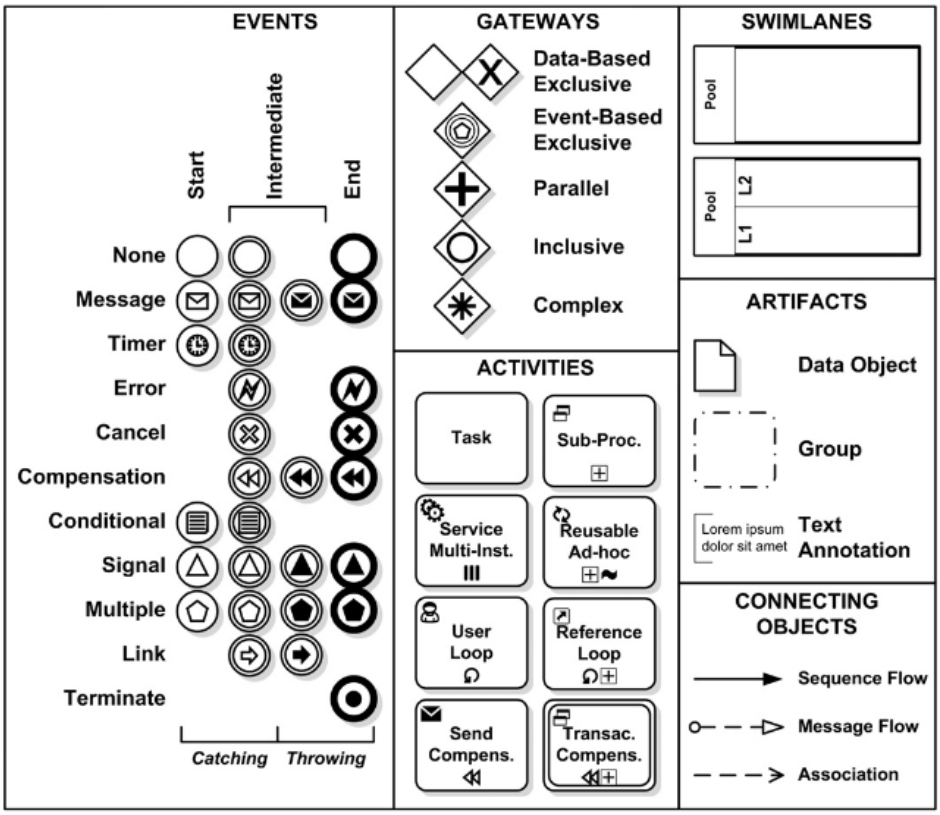


Figure 43: BPMN elements' summary (Chinosi & Trombetta, 2012)

This higher level of precision needs more complex use of the BPMN graphical elements.

Activities: Usually, when the first activity completes, the next one starts immediately. However, it is possible to face to a normal completion, or an abnormal completion. The first one is when the object that triggers the flow is out of an activity. All the parallel paths of a subprocess must be reached to complete the subprocess. Abnormal completion consists of the interruption of an activity or a subprocess by exceptional conditions. To represent that, we need to attach one or more events to boundary of task (e.g. a timer event to mention that after 30 minutes, the task is either achieved or closed). We can distinguish 2 flows with abnormal completion: the normal flow and the exception flow (i.e. the flow out of boundary event). If the exception flow is executed, the normal flow will not be executed.

Sub-processes represent end-to-end process and allow to reduce the size of a diagram. They can be used to represent reusable or global task and can be independent. A kind of sub-process is the event sub-process that is a process starting at any point in the global process, that is not part of the normal flow. Its start event has a trigger, and each time the start event is triggered (e.g. everyday at 8am), the event sub-process will start. Event sub-processes can be either interrupting or non-interrupting.

Activities can also represent repetitions. There are 3 possibilities. The first one is the loop activity. It means the process will “do… while…”. The next one is the multi-instance in parallel (|||). It means that the activities performed simultaneously. The last one is the sequential multi-instance. It is the same concepts that the second one, but the activities are performed successively.

Events: They are executed immediately and instantaneously. Events can be non-interrupting for event sub-processes (represented with dash line). To add precision on the diagram, internal markers are used. The markers correspond to a type of trigger. The main common triggers are:

- *Message*: It means that an external signal (i.e. a message) is directed to this specific process
- *Timer*: The process is scheduled
- *Conditional*: It means that we have to refer to data condition
- *Signal*: It is close to the message trigger, with an external signal but broadcasted to any listening process.
- *Multiple:* It can be any of multiple signals

There is generally one start event in a business process diagram. More than one start event is used when we are modelling channel-dependent start. A good practice is to use more than one end event for each distinct end state but generally it is recommended to use maximum 2 end events. Termination, error and cancel end events abnormally complete the process and immediately end the process. From its side, intermediate events are now very important and powerful in BPMN. They can be placed in the sequence flow or attached to an activity. Their representation is full if the intermediate event is interrupting, and in dashed lines if the intermediate event is non-interrupting.

Events are represented either filled or empty. The filled representation is for the throwing events (i.e. an event is thrown, and the process is reached or continues) and empty representation is for the catching events (i.e. an event starts or the process continues once an event is caught). In other words, send task consists of throwing a message and receiving task consists of catching a message.

Gateways: Like the events, gateways have no performers and are then executed immediately and instantaneously with no time lapse. Gateways are very useful to separate the process according to some decisions. It allows to model business rules.

Since the gateways correspond to a decision-making in the process, there exist different types. Gateways are used to split and to join activities. We can refer to the proposal connectives (see section 5.3.1) to understand how work the gateways.

- The exclusive OR means that only one alternative path is chosen.
- The inclusive OR means that one or more alternative paths are chosen.
- The parallel gateway is an AND relationship. It means that 2 or more alternative paths are chosen, and all parallel paths are followed.

A special type of gateway that mixes event and gateway is the event-based OR gateway. It is useful to model 2 or more alternative path, whose only one is chosen. Such a gateway is based on events that can happens next. (Snoeck, BPMN 2.0 : Advanced Modelling)

6.3.5. Modelling guidelines

While BPMN is a notation, some authors have developed a large number of guidelines to support modellers and analysts to create diagrams. Those guidelines make BPMN can become a technique, and no more just a notation that provides a set of symbols. *Corradini at al.* provide a framework to help users create understandable BPMN models. This framework counts 50 BPMN guidelines, collected, synthetized, and homogenized from 89 sources of guidelines available in the literature. Guidelines can be restricted for the model. For example, it is generally suggested to not exceed more

than 50 elements in a process, or to have maximum 2 end events. Another kind of guidelines is about the layout and the presentation of the model or also about the naming of activities and events. (Corradini, et al., 2018)

There exist different tools (online or offline) to exploit BPMN, create models and illustrate business processes. Some of them are better because they considering more modelling and practical guidelines. See the paper of *Snoeck at al.* to get more information about the relevance of each tool. (Snoeck, Oca, Haegemans, Scheldeman, & Hoste, 2015)

6.4. Benefits and limitations

In comparison with EPC (Event-driven process chains), BPMN is a good modelling notation because it is possible to transform a model from one notation to BPMN (or vice versa) with the same expressivity and only a very small loss of information. BPMN is even more powerful than EPC because for a same model, it needs up to 40% less elements, but using a bigger set of symbols. It means that BPMN models are easier to read and understand. (Levina, 2012)

BPMN can also be compare to UML activity diagrams (see next section) because the 2 notations can be used to represent the same processes and that elements from the 2 notations can correspond. In this way, a study showed that despite activity diagrams involve more constraints, the readability of BPMN or UML activity diagrams are the same for unexperienced users. (Geambasu, 2012)

BPMN was imagined to be understandable for any developer or user. It means that the notation uses universal and famous symbols to represent the elements of a system. For example, activities/tasks are represented by rectangle and a decision is modelled in the shape of a diamond. Different other notations have the same basics. Therefore, BPMN has a huge force which is its facility to understand and read a model, even if you are a newcomer. Tough the notation is simple, it even allows to represent complex business process, that is another advantage by using BPMN. (White, 2004)

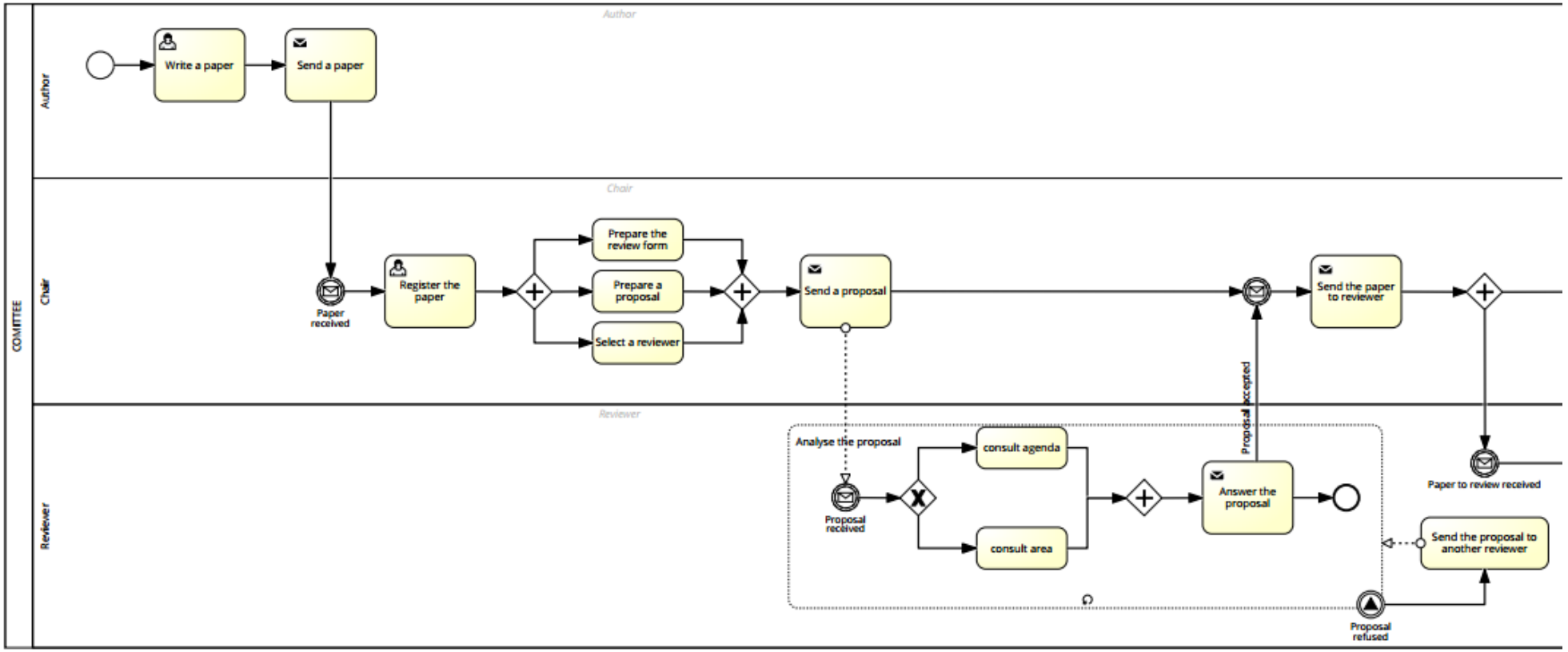
BPMN is established as the standard business process modelling notation. It means that its development is a crucial step to reduce the gap between all the different existing process modelling tools and notations. BPMI working group tried to improve and make BPMN more powerful by taking inspiration from other modelling notations. Another advantage of BPMN is that traditionally, business process modelling techniques needed to translate the models from original business process to the implementation models and it could lead to some difficulties and errors. (White, 2004)

Although BPMN contains and generates a lot of benefits, the popular notation also includes several weaknesses. First, BPMN can generate some ambiguity and confusing when some BMPN models are shared. Then, it contains a lack of support for routine and knowledge work but also when BPMN models are converted to executable environments. Business rules and decision-making lack some tools to support them. Security and some roles have also a lack of support and it is the same issue for some resource constraints (for example multiple tasks that require shared resources like a workspace). The temporal dimension of timed tasks and tasks with uncertainty about time or resources is also not optimized in BPMN. (Wikipédia, 2014)

6.5. Study Case

BPMN offers only a sequential view of a model. But in the case study we used, it is difficult to really establish a clear sequence flow of the operations. The global flow of the system (between author-

chair-reviewer) is modelled apart (see *figure 44*). Two other diagrams (*figure 45* and *figure 46*) explain how the chair and a committee member can interact, and also how the chair and the coordinator interact together to organize the committee meeting.


COMITTEE
Author
Chair
Reviewer
Author
Write a paper
Send a paper
Chair
Paper received
Register the paper
Prepare the review form
Prepare a proposal
Select a reviewer
Send a proposal
Send the paper to reviewer
Proposal accepted
Reviewer
Analyse the proposal
Proposal received
consult agenda
consult area
Answer the proposal
Paper to review received
Send the proposal to another reviewer
Proposal refused

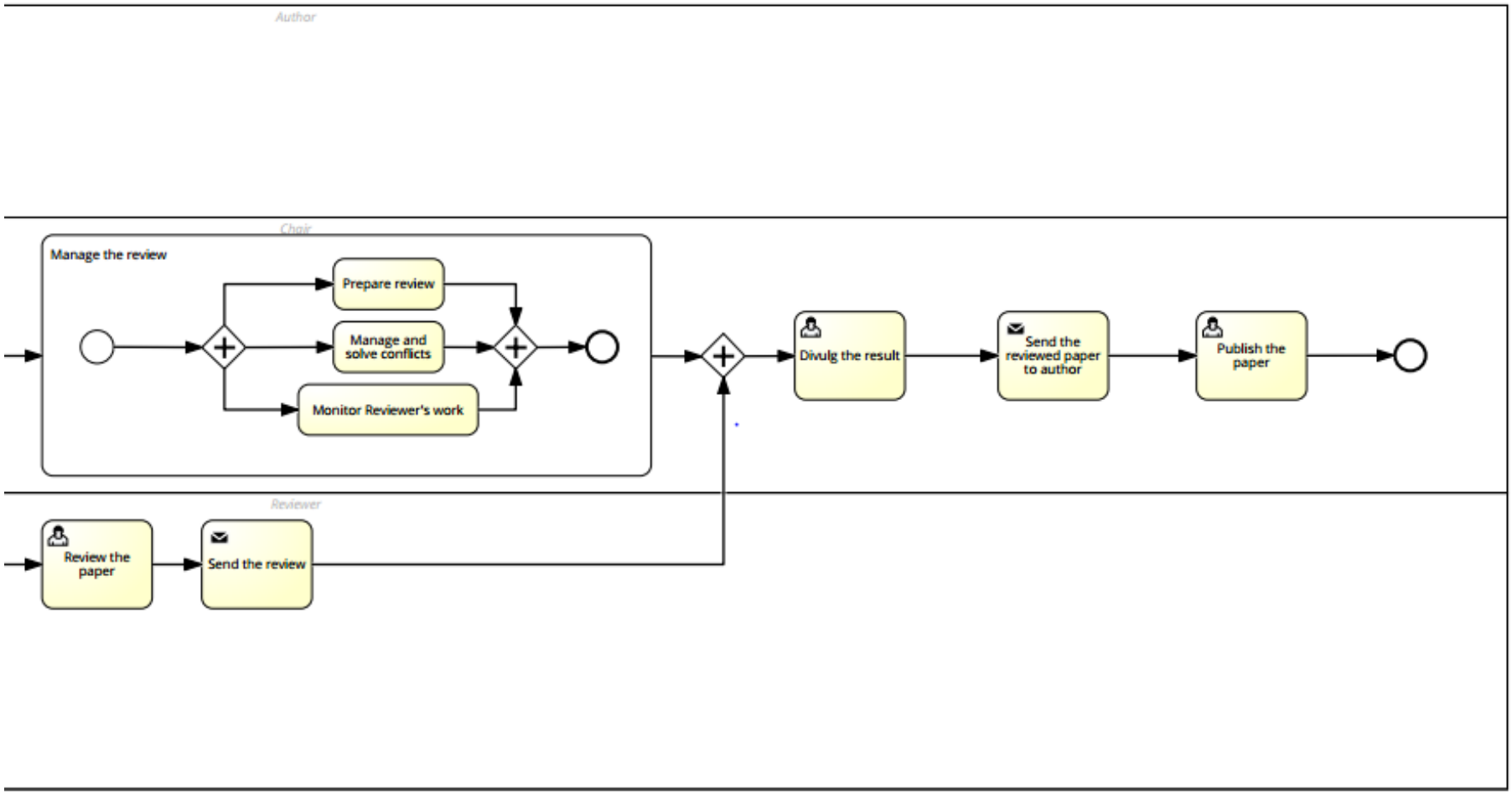


Figure 44: Expert Committee - BPMN Application : Global BP model

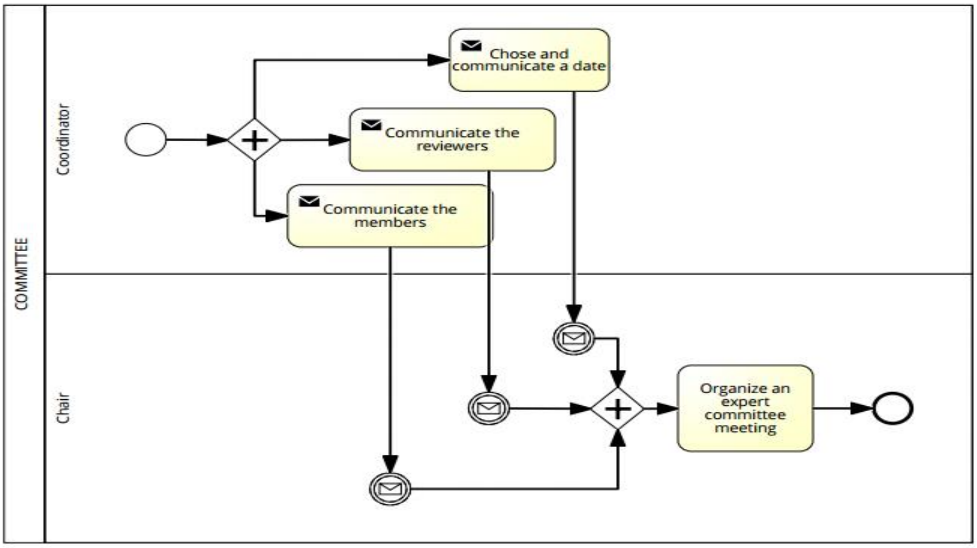


Figure 45: Expert Committee - BPMN Application :Meeting organization

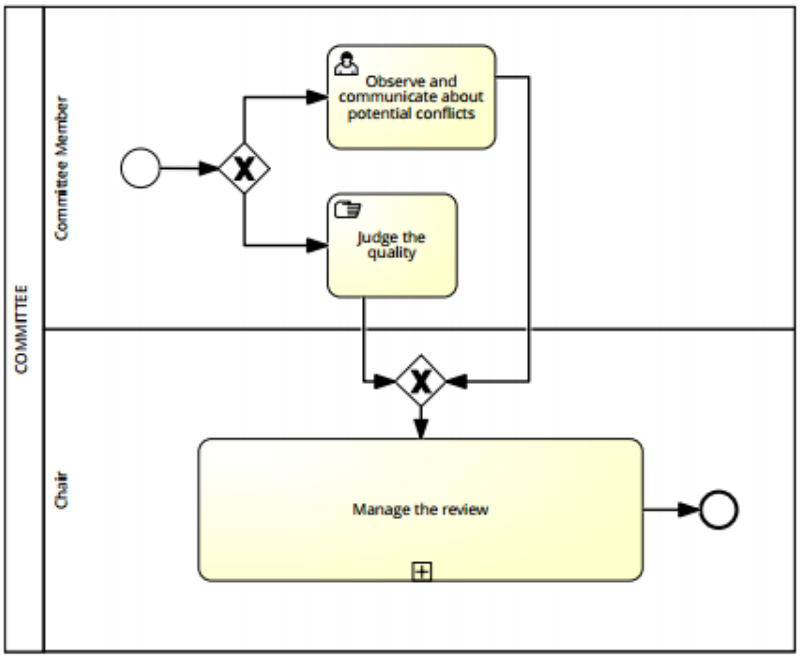


Figure 46: Expert Committee - BPMN Application : Manage review

7. UML

7.1. Description

UML stands for **U**nified **M**odeling **L**anguage and is a graphical language that aims to provide a normalized method to visualize and support system design. As its name implies, UML is born from the fusion between several previous object modelling languages. UML was first developed in 1996 by Grady Booch, James Rumbaugh and Ivar Jacobson and is maintained since 2005 by the OMG. “UML is the standard language for specifying, visualizing, constructing, and documenting all the artifacts of a software system” (Booch, 2005).

In the 80’s and 90’s, when it was the beginning of the information revolution and the development of informatics, there were lots of different methodologies and notations to model requirements. Unfortunately, it was too difficult to learn each of them, and use them properly. Each of them had its own strengths and weaknesses but it was impossible to use the proper methodology or notation on the right moment. It is then around 1993 that several researchers decided to focus on this issue and imagined the Unified Method, based on a set of aspects from other methodologies and notations that they thought interesting in different situations. They introduced this method in 1995. Two years later, they provided and introduced the Unified Modelling Language to the OMG after refining their first work. UML is then an answer to an old issue and was developed to facilitate the work and the life of both analysts and developers. Since UML is quite old, the OMG takes care to refine, improve and maintain the language to keep it valid and reliable. (Quatrani, 2003)

UML is basically intended for software-intensive systems, so it can be used in a lot of domains such as the enterprise systems, the banking and financial services, the telecommuniations, the transportation, the defense, the retail, the medical electronics or even in the area of distributed web-based services. UML is a very reliable language that is expressive enough to model systems that are not for software, like legal system or the behavioural and structural aspects of healthcare system. (Booch, 2005)

7.2. Approach

UML diagrams are initially based on **activity diagrams** concepts. Like BPMN models, activity diagrams allow to represent the flow of control. Activities illustrate generally action states and the flow is traversed as soon as an activity is achieved. This kind of diagram is usually used early in the analysis and design of the process in order to show the business workflow to model. They can also be used to show where use cases may be in an activity to model what it is happening in a particular use case. Activity diagrams also allow to illustrate and represent easily the swimlanes of a system by separating the activities belonging to specific actors. It is a good way to show who is responsible for what activities. (Quatrani, 2003) UML is an appropriated language for modelling systems, both from enterprise information systems to smaller web-based applications. It is also very useful for hard real time systems. The fact that it is based on several other languages, it offers a general view to develop a system. (Booch, 2005)

In addition to activity diagrams, UML also considers **use case diagrams**. This kind of diagram is firstly based on the actors participating to the system. In Use Case Diagrams, an actor represents someone (or something) who is external to the system, but who interacts with it. They are illustrated as stick figures. The question to ask about use case diagrams is “who is going to interact with the system?”. A use case is in fact a sequence of linked transactions performed by actors in a system. It corresponds

to provide value to the actors. After having identified all the actors who interact with the system, the next step is to document the use cases. Documenting the use cases consists of specifying the flow of events, from the actor's point of view. The goal is to explain what the system needs to provide to the actor, once the use case is executed. In use case diagrams, we can differentiate 2 kinds of scenarios. The first one is the normal flow, called the "happy days" scenario. It corresponds to a flow where everything works. The other one represents the abnormal flow and is called the "rain day" scenario. Documentation of the use cases always show a normal start (i.e. happy days scenario) but the details added by this documentation must provide information about what happens if the system does not work. Use case diagrams are then great diagrams that graphically shows a nice overview of the system, illustrating the actors and the functionalities that the system has and must provide.

UML also integrates **sequence diagrams**. It consists of diagrams showing object interactions that are performed in a time sequence. Objects and interactions that we need to achieve the specified functionalities can be determined using the flow of events. Such a diagram can be very powerful and interesting to capture requirements and overall user interface requirements that are generally more difficult to find. Sequence diagrams are then efficient to show what is going on, to drive out requirements and to work together with customers. We need to create sequence diagrams until we do not find new objects to model.

Collaboration diagrams are also incorporated in the UML approach. They display object interactions organized around objects and their links to another. In contrast with sequence diagrams, objects are not ordered according to time in collaboration diagrams. An advantage of this kind of diagrams is that it is easier to see and observe messages that go between 2 objects for a particular use case when the scenario is longer. Collaboration diagrams are just another view of a scenario, compared to sequence diagrams. The choice of a diagram or the other one depends to the analysts' and developers' decisions.

Another type of diagram that are very often used when we talk about UML is the **class diagrams**. A class corresponds to a collection of objects that have common structure, common behavior, common relationships and common semantics 3 horizontal parts, class can be found by examining the sequence and collaboration diagrams. The class name is in the first compartment of the rectangle box. The second one shows the structure of the class (i.e. the attributes) and the last one shows how it performs (i.e. its behavior, its operations). Except for the class name, the other compartments can be facultative. Using the vocabulary form the system domain is very important to name the classes. Class diagrams allow to show a static view of the system by representing a set of classes and their relationships in the environment. While operations for the class diagrams can be found by examining interaction diagrams, structure of classes are got by asking domain experts and by looking to requirements.

Obviously, UML was imagined by taking ideas from other kind of diagrams like the **state transition** ones, the **component** ones, or the **deployment** ones but these last cited diagrams had brought less contribution to the elaboration of UML.

7.3. Modeling concepts

Class diagrams is a big part in the UML. The UML modelling elements that we can find in class diagrams include the classes and their structure and behavior, the relationships including association, aggregation, dependency or inheritance, the cardinality or navigation indications and the role names.

Concerning the relationships that represent in UML a communication path between different objects, we can differentiate 3 types:

- *Association*: It consists of a bi-directional connection between classes. Represented in UML by a line connected the related classes, an association means that 2 associated classes can interact because they "know" each other.
- *Aggregation*: It consists of a stronger form of relationship where the link is between a whole and its part. It allows to announce to the developer that the related objects are stronger coupled. Aggregation is represented in UML by a line connected to a diamond next to the class corresponding to the whole.
- *Dependency*: It consists of a weaker form of relationship and shows the link between a client and a supplier, in which the client has no semantic knowledge about the supplier. It means that a class (i.e. the client) needs the services of another (i.e. the supplier) but the client ignores that the supplier exists. Dependency is represented in UML by a dashed line that points from the client to the supplier.

The cardinality specifies how many objects are participating in the relationship. It corresponds to the number of instances of one class related to an instance of another class. Association and aggregation relationships require to establish 2 cardinalities, that is one for each end of the relationship. The number of instances is showed by writing the number itself and the multiplicity of many is represented by *. The direction of the navigation is shown by an arrow. Since associations and aggregations are bi-directional, we often restrict the navigation to one direction. The navigational direction is shown by an additional arrowhead when it is restricted.

Another type of relationship exists. It is the inheritance one that represents the relationship between a superclass and a subclass. It allows developers and analysts to add additional behaviour to the system. It is represented as a triangle. (Quatrani, 2003)

7.3.1. UML conceptual model

The conceptual model of UML allows to better understand how works the language, and how to use it. This conceptual model implies knowing the 3 major elements of UML models. The first one is the UML's building blocks, that are the basis of the language. The next one is the rules that allows to understand how work these blocks together. The third and last element consists of some mechanisms applied throughout the language.

A) UML's Building Blocks

In UML, we can differentiate 3 types of building blocks, named things, relationships, and diagrams. Things correspond to the abstractions of classes in a model, these classes are related together through relationships and collections of things are grouped by diagrams.

About things, we can also differentiate 4 different kinds of things: structural, behavioral, grouping and annotational ones.

Structural things: It consists of the noun of UML model, that means there are static parts of a model that illustrate both conceptual and physical elements. There exist different kinds of structural things but the most common is the class.

A class: It is a description of a set of objects, sharing same attributes, operations, relationships, and semantics. A class is graphically represented by a rectangle, including its name, its attributes, and its operations.

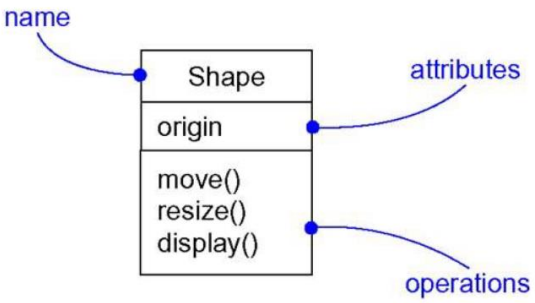


Figure 47: Example of a class and its name, attributes, and operations in UML (Booch, 2005)

Behavioral things: It consists of dynamic parts of a model, it means there are the verbs of the model, that represent its behavior in time and space. Behavioral things can be either an interaction or a state machine.

An interaction is the behavior generated by messages exchanges among some objects in a specific context in order to achieve a goal or an action. Such a behavioral thing involves several elements like messages, action sequences and links. Graphically, an interaction is simply represented as a named directed line.

A state machine is also a behavior but specifying the sequences state that an object has during its lifetime, responding to an event. This behavioral thing involves also other elements like the states, the transitions (i.e. the flow), the events (i.e. the triggers) and the activities. A state machine is graphically illustrated as a rounded rectangle.

Grouping things: It consists of the organizational parts of a model. It means grouping things are boxes that allow to decompose a model into. The most common grouping thing is called a package.

A package is a general mechanism, allowing to organize elements into some groups. Both structural, behavioral and grouping things can be grouped in a package. A package has only a conceptual dimension, it means that it only exists during the development time.

Annotational things: It consists of explanatory parts of a model. It means that annotational things are comments applied on model's elements to add some information about like a description, a remark, or a clearing. A kind of annotational thing is a Note. A note allows to add a constraint and a comment to an attached element, or even a collection of elements. It is graphically illustrated by a rectangle box with a dog-eared corner, with the textual information inside. Another kind of annotational things is a requirement, that allows to specify the desired behavior of the model.

There exist 4 types of relationships in the Unified Modelling Language that are the basics for relational building blocks in the UML. They allowed to create models.

- Dependencies: It corresponds to a basic dependency where an element (called the independent thing) can have an impact on another element (the dependent thing) when its state is changing.

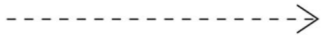


Figure 48: Dependency in UML (Booch, 2005)

- Associations: An association is structural relationship, used to model a set of links among objects. A subtype of associations is the aggregation, that represent the relationship between a whole and its parts. Association is represented with the multiplicity of the connection and the role names.

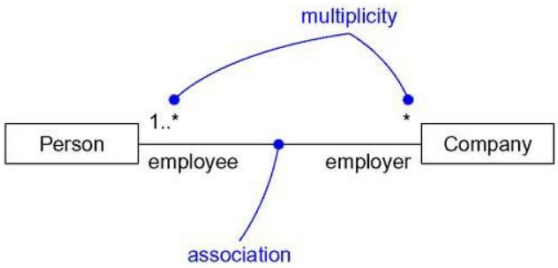


Figure 49: Association in UML (Booch, 2005)

- Generalizations: Also called specializations, this relationship allows to illustrate the link between child elements and their parent element. The child will get the same behavior than its parent.

Figure 50: Generalization in UML (Booch, 2005)

- Realizations: This relationship acts between classifiers. Thanks to the realization, one classifier will specify a contract that another classifier will have to respect. Realizations take place between interfaces and the components (i.e. class) that realize the activity or between use cases and the collaborations between actors that realize them.

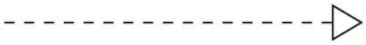


Figure 51: Realization in UML (Booch, 2005)

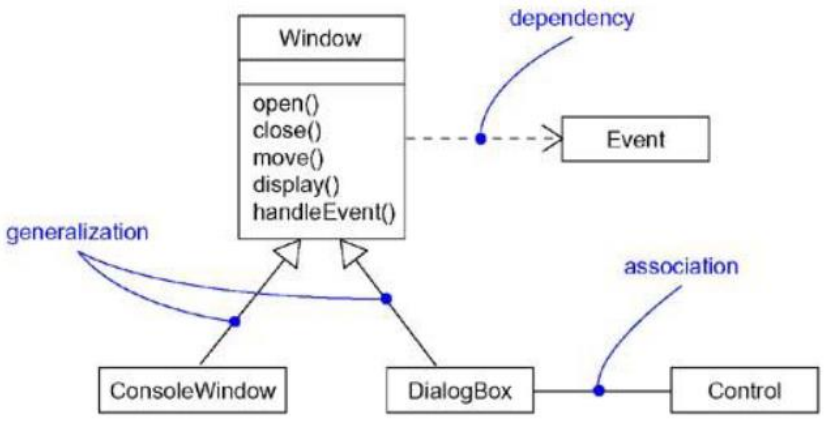


Figure 52: Example of a model with relationships in UML (Booch, 2005)

As explained in section 7.2., UML is a language whose the approach is based on multiple diagrams. Diagrams allow to illustrate and visualize a system from different perspectives. A diagram is a way to represent a set of elements, and to show how they are related and how they work together to ensure the operational dimension of a system. In theory all elements can appear in all diagrams, but it is usually not the case. UML, in order to be more reliable and powerful, combines 9 different diagrams, taking advantage from each of them to be as consistent as possible. The 9 diagrams included by UML are: (Booch, 2005)

- **Class diagram**: The most common diagram for object-oriented systems. It represents a static view of it.
- **Object diagram**: It represents static snapshots of some instances of a system. We could compare the class diagram as a meta-model of an object diagram. It shows the static view but in real cases.
- **Use Case diagram**: It represents several use cases of the systems, making in action the actors and their relationships. It also shows a static view of the system but is very important to illustrate some behaviours.
- **Sequence diagram**: It represents the interactions in a system, and particularly the time-ordering of messages.
- **Collaboration diagram**: It also represents interactions in a system but puts the emphasis on structural dimensions of objects that interact, i.e. those which send and receive messages.

  Sequence and collaboration diagrams are considered as interaction diagrams. It means that these diagrams represent some interactions, involving messages in the relationships between some objects or actors. They show then a dynamic view of a system because some pieces of information are shared in the model.
- **Statechart diagram**: It represents the state machine of a system, i.e. all the states, the transitions, the events, the activities that a system can contain. It shows then a dynamic view of a system because the emphasis is put on the order of each elements. It allows to well illustrate reactive systems.
- **Activity diagram**: It also represents the state of a system, but only from an activity to another one. The flow of control is a very important aspect of activity diagrams that show a dynamic view of a system.
- **Component diagram**: It represents the organization of system components like classes or interfaces. It shows then a global static view of the future implementation of the system.
- **Deployment diagram**: It represent the configuration of the model nodes and the components that are attached to them. It shows the static view of the design of the system. Obviously, deployment diagrams and component diagrams are lined since they both involve the state of system components.

UML can provide a model that combines all these diagrams, or just several ones. It is important to notice that this list of diagrams is not a closed one and can be exhaustive.

B) UML's Rules

All the previously mentioned UML's building blocks involve some rules that must be satisfied to generate well-formed models. It is the case in UML like in other languages. It means it is mandatory to respect these rules to get a semantically self-consistent model.

UML involves 5 semantic rules.

1. For the names: It is about what we can call things, relationships, etc.
2. For the scope: It is about the general context that allow to give a meaning on a name.
3. For the visibility: It is about the fact that the names must be reusable and seen by other users.
4. For the integrity: It is about the correctness and the consistently between relationships.
5. For the execution: It is about what must be simulated or run for a dynamic model.

UML involves 3 other rules to satisfy during the development of the system.

1. Elided: It means that several elements can be hidden in order to simplify the general view.
2. Incomplete: The development can lead to a model where several elements are missing.
3. Inconsistent: The development phase can also lead to a model where the integrity is not guaranteed more.

The software development life cycle can not avoid such less-than-well-formed models but UML will always support you to get the most efficient model because be wondering important questions, it would be possible to make the model well-formed over time. (Booch, 2005)

C) UML's Common Mechanisms

Building a model is like building a house. When you are building a house, you can choose an architectural style and respect its features. It is the same for UML's models. When you are building a model in UML, there exist 4 common mechanisms which consistently applied together, simplify the build of the model. (Booch, 2005)

1. **Specifications:** UML is not just a graphic language. Because all the graphic notations, the building blocks, refer to a specification which allows to obtain in a textual way the semantic and syntactic details of the model. For example, for a UML class, a specification will make it possible to provide all the attributes, operations and behaviors of this class. In addition to this, the specification also gives the signature of the class, which is not always mentioned in the graphical part of the model. This graphic notation which precisely could only show a small part of the specification.
   Where the graphical notation of UML makes it possible to visualize a system in a global way, the UML specifications make it possible to indicate the details of this same system. Specifications can be made after designing a model, or just during the conception of each element.
2. **Adornments**: In UML, almost all elements are graphically represented in a direct and unique way. That helps to provide a visual representation of most of the (most important) aspects of these elements.
   Let us take the example of a class in UML. Its notation is intentionally easy to draw since classes are the most recurrent elements of UML models, and object-oriented models in general. The class notation will therefore show its most important aspects, such as its name, attributes, and operations. In addition, the specification of the class will provide additional details such as whether it is abstract or not, or about the visibility of its attributes and operations. These details, contained in the specification, can consequently also be present in

the graphical notation of the class, thanks to adornments. The adornments are small symbols or text, usually very discreet, which provide additional information about the class.
The following figure shows for example an abstract class of which 2 operations are public, 1 is protected and another is private.

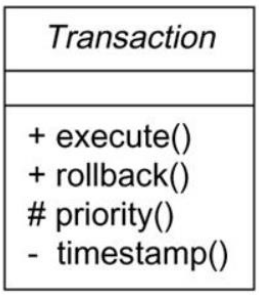


Figure 53: Adorned class in UML (Booch, 2005)

3. **Common divisions**: Object-oriented systems, in their modelling phase, often offers minimum 2 ways of divisions.
   The first one is the division between class and object. A class is an abstraction while an object is a more concrete instance of that abstraction. It means that UML offers the opportunity to model classes as well as objects. The same dichotomy is possible between all kinds of building block in UML, for example with use cases and use case instances. UML distinguishes then an object from a class by using the same symbol than the class, but by underlying the object's name.
   The second division is between interface and implementation. While an interface allows to specify and declare a contract, the implementation will realize and execute this contract. The implementation is also responsible for satisfying the semantics of the interface. UML allows then to represent both interfaces and their implementations. Like for the first division, almost all building blocks in UML share the same dichotomy. For example, facing with use cases and the collaborations that realize them.
4. **Extensibility mechanisms**: UML is opened-ended. It means that it is possible to extend the language to make it more consistent, to be sufficient to express different nuances in different domains across all time. The extensibility mechanisms of UML include:
   - **Stereotypes**: They serve to extend the vocabulary of the UML. They allow to create new kinds of building blocks, from existing ones, to get specific solutions to the problem. Stereotypes can be used to represent exception in a programming language like Java for example.
   - **Tagged values**: They allow to extend the properties of building blocks. It means you can create some new details in the specification of this element. For example, it is possible to add details about the date and the author of a particular version of an element. It will allow to edit it more easily over time.
   - **Constraints**: They allow to extend the semantics of UML's elements. It means you can add some rules or modify some existing ones.

   These 3 extensibility mechanisms, together, allow to get a language that deals better with the initial problem. Although the extensions need to be controlled, they help UML to adapt its language to different software technology and so to become more powerful again.

7.4. Benefits and limitations

One of the benefits about UML is its versatility. Indeed, activity diagrams (for example) can be used in different phases of the lifecycle of the model, whether in the design phase or simply to show how works the flow between the methods of a class.

UML counts lots of extension that allow it to be more powerful and efficient than the basic UML. It allows to answer to issue in more various environment and to satisfy more requirements. (Quatrani, 2003) UML is become a universal language thanks to its flexibility and its versatility.

While UML is a very expressive language, and is based on other modelling languages or diagrams, it is not difficult to use and understand it provided that the 3 major elements of UML are known. These 3 elements are the building blocks of the language (see section 7.3.1.), the rules to apply to understand how the blocks work together, and some common mechanisms that must be applied through the UML. (Booch, 2005)

It is important to notice that UML is often used (and is efficient) to represent object solutions. The expression of a such solution and its comparison and evolution are easier with UML.

Although UML is the most common language to model software design, it is possible that it causes some troubles to the modelers and developers. The needed time to handle and maintain UML models can be a disadvantage. UML diagrams need to be synchronized with the code of the software, and this synchronization adds time on a software development project. Small companies and independent developers could not be able to manage this extra work. Moreover, UML is not always useful for software developers because they work directly with the code and not with schemas. On the other hand, UML diagrams are useful for project managers because diagrams help them to illustrate how the desired software will work. It is important to pay attention to the complexity of the diagrams. Indeed, if the diagrams are too heavy, it will complexify the work of the developers to first understand the models, and then implement them. A solution is to only incorporate basic elements and high-level elements in UML models. (web)

To resume, UML is a formal and normalized language that allows to gain in precision, to get stability and to use different tools. It is also a powerful support of communication. Complex and abstracts representations can easily be understood when they are expressed in UML. Nevertheless, the language has also some disadvantages like the fact that it needs a learning and a period of adaptation to know and be able to use it. UML is not at the origin of the object conception but it unifies the different approaches and gives a more formal definition of them.

7.5. Study case

The Expert Committee study case was adapted in 3 UML diagrams. *Figure 54* shows an UML class diagram illustrating the global system. A state-chart diagram shows how the actor and objects interact in *figure 55*. Finally, a sequence diagram explains the relationship between the chair and the reviewer in *figure 56*.

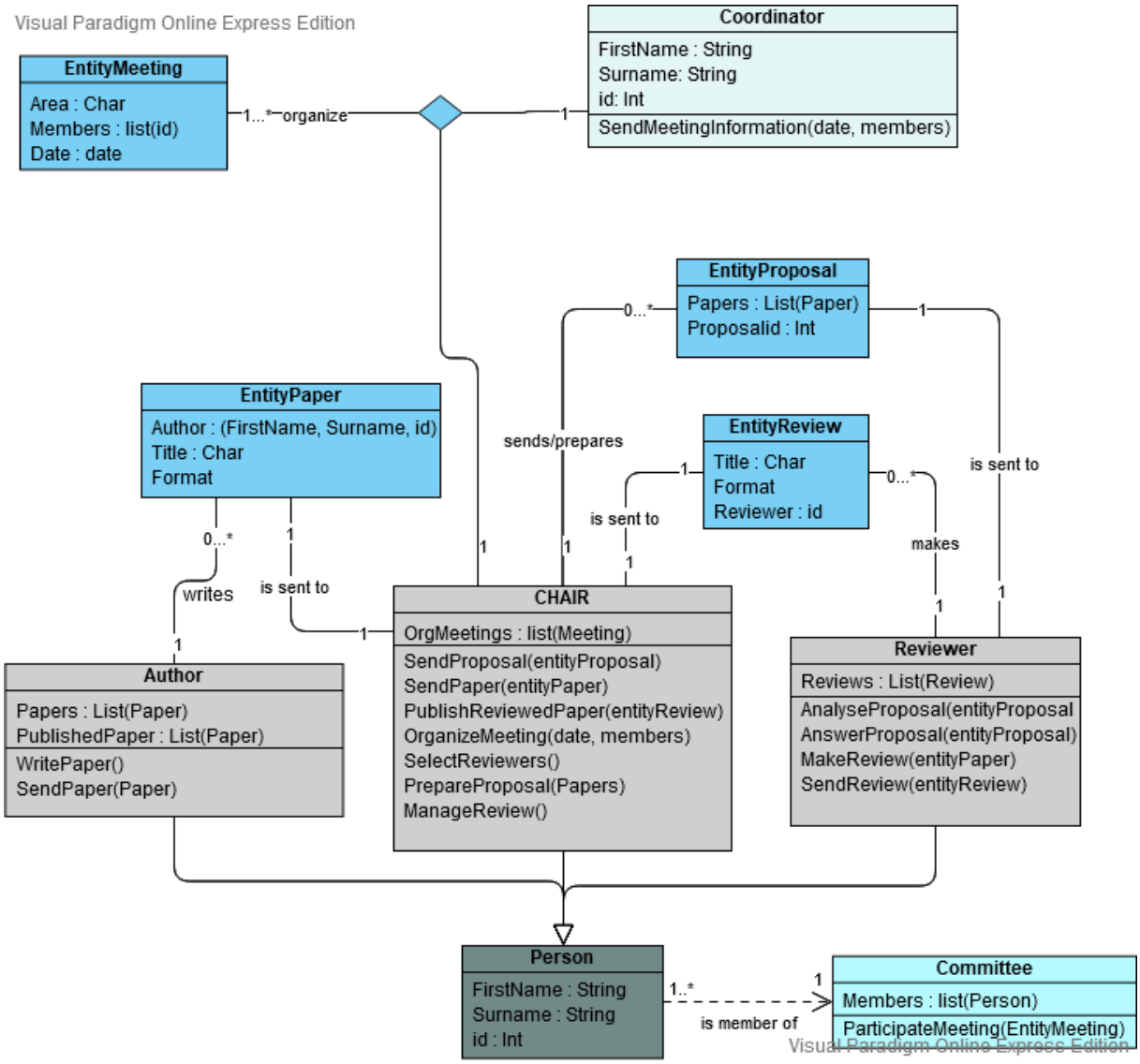


Figure 54: Expert Committee - UML Class diagram

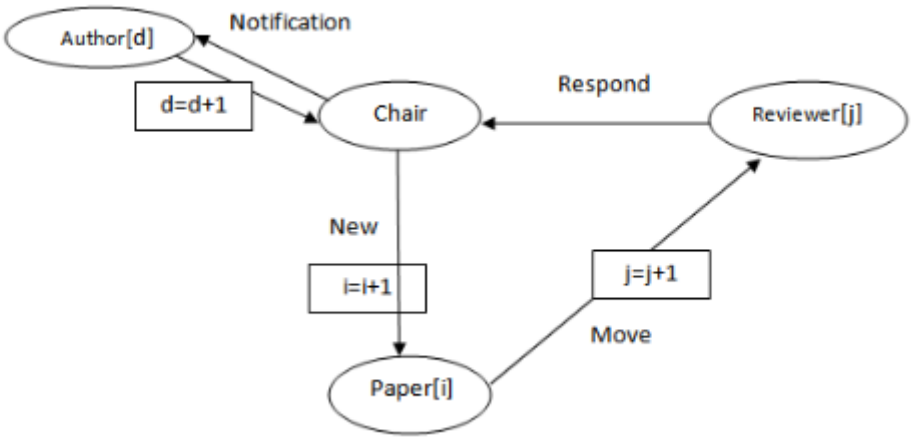


Figure 55: UML application of Expert Committee - State-chart diagram (Hanandeh, Alsmadi, Al-Shannag, & Al-Daoud, 2005)

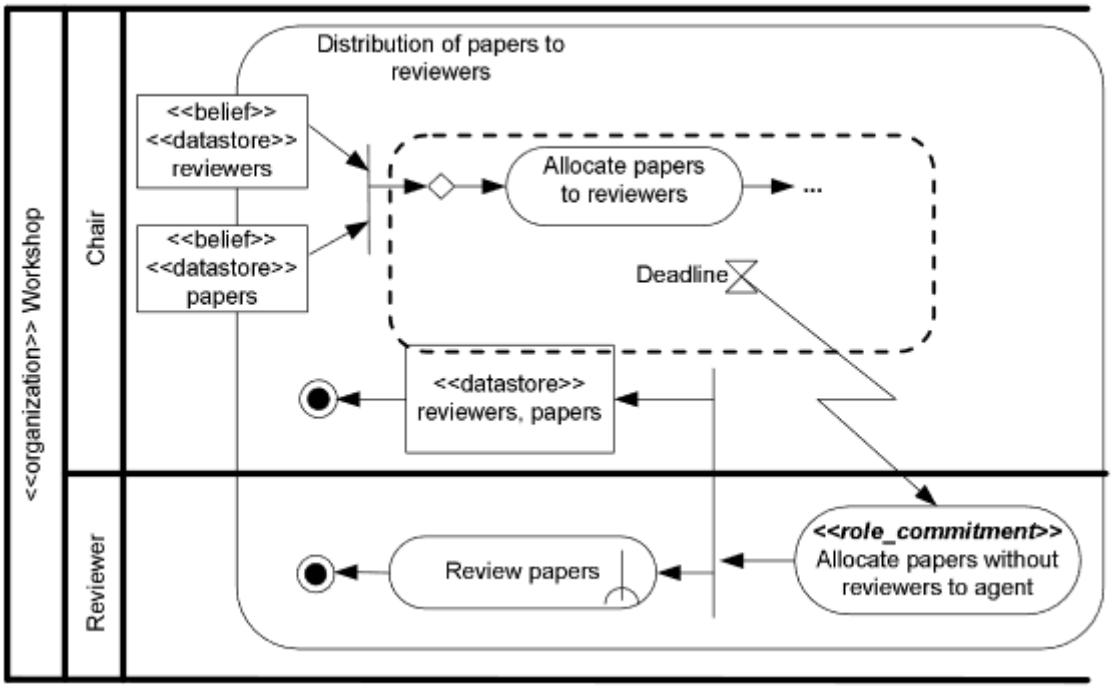


Figure 56: UML Sequence diagram of Expert Committee system (Silva, Noya, & Lucena, 2006)

8. Conclusion: Comparison Table & Recommendations

This final section is a summary and comparison between the 6 languages, frameworks, methodologies seen in section 2 to 7. The comparison is based on a set of attributes and features that characterized each of them. The goal is to provide an overview about the specificities of each models to be able to make a choice between different ones when a business analyst has to capture requirements or model a system.

8.1. Comparison table

KAOS
The fame
KAOS is a university reference **FRAMEWORK**
Requirements Phase
Early-phase Late-phase
Ontology
Social Intentional
Approach
Goal-oriented: Goals justify the system and why and how it acts. Agent-oriented: Agent concept is also present in KAOS that allows to represent interactions.
Static or Dynamic systems?
KAOS models both static and dynamic systems but does not include a temporal dimension.
Covered type(s) of requirements
Functional requirements Non-Functional requirements
Application domains
Not specified but KAOS is general enough to cover any area
Public
KAOS is quite simple to learn Analyst must have a very good knowledge of the system environment to create efficient model.
Maintenance/extension
UNKNOWN
Number of required models
4 models: ➔ *Goal Model* ➔ *Responsibility Model* ➔ *Object Model* ➔ *Operation Model*
Errors handling/security
Only conflicts handling
Cardinality concepts
NOT INCLUDED
Behavioral concepts (intentions, capabilities, beliefs, responsibility, etc.)
KAOS is a model from intentional ontology and covers different aspects like intention, capability, and responsibility of agents.
Scenario
KAOS meta-model allows to support repetitive scenario

Decision making
NOT INCLUDED (except by modelling task named with decision-making)
Extra specifications
NOT INCLUDED (in the KAOS model)
Ease of use
KAOS is a basic framework with simple graphical elements. There are just few elements, therefore KAOS models can be characterized as simple models.
Time
Time is a big disadvantage in KAOS because it takes lots of time to create a complete model. Justified only from 100-man days project.
Cost
In case of big or medium-size system, the cost to create KAOS models will be interesting. In case of small system, the cost will be too expensive for what KAOS delivers.
Main benefits
KAOS integrates completeness, traceability, and unambiguity → **POWERFUL FRAMEWORK**
Supporting tools
KAOS is a university reference but is not very spread in prof world so there does not exist many tools to support model's creation. KAOS is supported by the Objectiver that provides an easy interface and documentation support. Unfortunately Objectiver contains some academic limitations due to the fact that it is a commercial tool. However, the KAOS framework includes the Grail assistant to support the acquisition strategy.
Compatibility with other models
KAOS can be complementary with UML where entities become classes and where associations become binary associations.
iStar
The fame
iStar is known as an efficient goal-oriented **LANGUAGE**
Requirements Phase
Early-phase
Ontology
Social Intentional
Approach
Goal-oriented Agent-oriented : Very developed with actor, agent, role and position concepts.
Static or Dynamic systems?
The preference goes to static systems because iStar does not include a temporal dimension in its 2 models but even in static view of a system, iStar only shows the intention aspects.
Covered type(s) of requirements
Functional requirements: iStar specify what the system should do Non-functional requirements
Application domains
Very large: ○ Air traffic ○ Agriculture ○ Healthcare ○ Telecommunications ○ Business Processes

○ …
Public
An iStar model is more complex thana KAOS model. The graphical elements and number of dependencies make more difficult to create and understand iStar models. Analysts must precisely specify the system (stakeholders are involved in the process).
Maintenance/extension
iStar is maintained and the newest version is iStar 2.0
Number of required models
2 models: ➔ *Strategic Dependency model* ➔ *Strategic Rationale model*
Errors handling/security
iStar integrates security dimension of the system.
Cardinality concepts
NOT INCLUDED
Behavioral concepts (intentions, capabilities, beliefs, responsibility, etc.)
Agent's interests and behavior are considered in iStar: Agents are not fully controllable They act with intention, not randomly (agents are independent). Agent' opportunities and vulnerabilities are covered in iStar
Scenario
iStar does not really represent scenario but more a global view of a system and what the system should do
Decision making
NOT INCLUDED
Extra specifications
NOT INCLUDED
Ease of use
iStar is more difficult to interpret than KAOS model because there exist more graphical elements but it stays a simple language.
Time
In iStar, there are as many SR models as there are actors in the system. In case of small systems it is not a problem but
Cost
More modelling takes time, more cost is high Ok for small systems (5/6 actors) Not ok for bigger systems
Main benefits
iStar considers system's and agent's interests together, and not separately
Supporting tools
iStar is spread enough so there exist different tools to model in iStar (piStar, OME, …). Such tools give support to create SR and SD models but have also some limitations, about operations editing for example. However, iStar has an obvious lack of formal definition of its elements.
Compatibility with other models
Two systems can collaborate if they interact together iStar is compatible with TROPOS
TROPOS
The fame

Tropos is a **METHODOLOGY/FRAMEWORK** inspired and based on iStar
Requirements Phase
Early-phase Late-phase
Ontology
Social Intentional
Approach
Agent-oriented with some GORE techniques
Static or Dynamic systems?
Only intentional aspects are covered (like in iStar)
Covered type(s) of requirements
Functional requirements Non-functional requirements can be modelled thanks to supplementary constraints and invariants
Application domains
Useful for collaborative, dynamic and distributed application Tropos has not yet proven itself for all types of systems (e.g. multi-agent systems).
Public
First, the public who could potentially use Tropos need to have a good knowledge of iStar. Moreover, Tropos has a formal language (Formal Tropos) which is quite complex and asks for some time to develop its 2 layers.
Maintenance/Extension
Tropos is an extension of iStar itself. Secure Tropos is a famous extension of Tropos and integrates aspects like risk, security, legal norms.
Number of required models
3 models: ➔ Strategic Dependency model (from iStar) ➔ Strategic Rationale model (from iStar) ➔ Goal model Formal Tropos is divided in 2 layers: ➔ Inner layer: Class declaration ➔ Outer layer: Constraints declaration
Errors handling/security
Tropos integrates risk, security, and legal norms
Cardinality concepts
Conditions, invariants, and constraints added in Formal Tropos can provide cardinality information.
Behavioral concepts (intentions, capabilities, beliefs, responsibility, etc.)
Intentional aspects are integrated in Tropos
Scenario
Tropos uses design patterns that allows to represent repetitive scenarios.
Decision making
NOT INCLUDED
Extra specifications
Formal Tropos allows to add some details about the system thanks to constraints/conditions
Ease of use
For modelling dimension, Tropos is not more complex than iStar. FT takes time to understand all the specificities of the language because it is not a common language (but formal)
Time

To be completed in Tropos, analysts need to create 3 models and to specify the system in a formal language, so it takes more time than a simple language (that include only models)
Main benefit
Tropos supports both early and late-phase of requirements ➔ Early-phase for the system as-is ➔ Late-phase for the system-to-be
Supporting tools
The Tropos methodology is supported by some tools (like T-Tool, GR Tool, etc.) but there is clearly a lack of powerful tool to support it.
Compatibility with other models
Since the framework is based on it, it is compatible with iStar.
Z NOTATION
The fame
Z is an old **NOTATION** (created in 1980)
Requirements Phase
Early-phase requirements Late-phase requirements
Ontology
Dynamic: because operation schemas are a big part of the notation
Approach
Model-based: Z Notation does not emphasize agent or goal aspects of the system. It is focus on the model itself and is very powerful to illustrate a database or a system.
Static or Dynamic systems?
Z Notation can represent both static and dynamic systems. Database modification or operations are easy to model. It is also the case for database exploring (without modifying the database).
Covered type(s) of requirements
Functional requirements Non-functional requirements
Application domains
Z Notation is mainly used in IT field. It is even more useful when preventing bugs and malfunctions are important (software, operationg system, etc.)
Public
Z Notation is a formal methods notation. It means that it is not very intuitive at first sight. Moreover, some mathematical concepts like predicate logic, set theory, cartesian product and schema are the basis of Z declaration. An analyst without very good mathematical skills would have some difficulties to create Z schemas.
Maintenance/extension
UNKNOWN
Number of required models
In Z Notation, there are 3 different schemas: ➔ State-space schemas ➔ Operations schemas ➔ Errors handling schemas
Errors handling/security
Z Notation allows to ensure that the system is error free by handling errors in specific schemas
Cardinality concepts
Cardinality can be represented indirectly thanks to quantifiers.

Behavioral concepts (intentions, capabilities, beliefs, responsibility, etc.)
NOT INCLUDED
Scenario
There is no concept in Z for repetitive scenario. Since the notation is based on the global model, we can consider that it is not a real attribute of Z Notation.
Decision making
NOT INCLUDED
Extra specifications
In Z Notation, variables are typed. It can help the developed for the implementation and add some details about the system. Moreover, it is possible to add details by adding some constraints and invariants in the schema.
Ease of use
First, Z Notation required very good mathematical skills. It is not very intuitive and readable as schemas because it contains a mix of boxes, text, Greek letters and other special symbols but it is easy to learn it. When system is big and complex, the task of representing it with Z Notation is very complex.
Time & Cost
A huge disadvantage of Z Notation are the development and verification costs
Main benefit
Z Notation allows to have a clear identification of inputs and outputs of a system
Supporting tools
There are very few Z tools. (Z word tool, Z-editor, …)
Compatibility with other models
UNKNOWN
BPMN
The fame
BPMN is a worldwide reference as a **NOTATION** to illustrate business processes.
Requirements Phase
Late-phase requirements BPMN only shows what the system should do but does not take care about why the system do that (i.e. early-phase requirements)
Ontology
Dynamic: A big feature of BPMN is its temporal dimension.
Approach
Operation-based: BPMN represents the sequence flow, and provides a clear view of
Static or Dynamic systems?
Dynamic systems: BPMN is a powerful notation for dynamic processes or systems. It clearly shows the interaction between different actors, emphasizing what they do, and in which order they do it.
Covered type(s) of requirements
Functional requirements
Application domains
BPMN is very efficient and powerful for business processes because it shows a clear view of the sequence flow. Collaborative (multi-agents) and internal business processes are the main application domain of BPMN. Pay attention that ambiguity and confusion are generated with BPMN when different models are shared. Moreover, there is a lack of knowledge with a BPMN diagram when the environment is executable.

BPMN is more efficient for small and simple business processes. When there are too many agents, the diagrams become quickly too complex and unreadable.
Public
Everyone can be able to create and understand a BPMN diagram because core elements are very basic and it can be sufficient to create an understandable diagram.
Maintenance/Extension
BPMN is a reference for business process modeling. It is frequently maintained and there exist different extensions to improve its efficiency. The current version is BPMN 2.0
Number of required models
From 1 to … If the business process is very small, only a diagram will be sufficient. Nevertheless, if the business process is very complex, it would be interesting to split the general process in some different subprocesses to avoid an unreadable view of the system.
Errors handling/security
BPMN integrates a concept of abnormal completion of a task. It means that if errors are potentially waited in the process, the diagram can show them by differencing normal and abnormal completion of a task or a (event)(sub)process. Security is not supported in BPMN.
Cardinality concepts
NOT INCLUDED
Behavioral concepts (intentions, capabilities, beliefs, responsibility, etc.)
NOT INCLUDED However, specifying the task type allows to add information about who is responsible for and how the task is handled.
Scenario
A BPMN diagram corresponds to a scenario. Smaller repetitive scenarios can be modelled as Event subprocesses to have a clearer view.
Decision making
The gateways (BPMN objects) allow to represent decision making with AND/OR/XOR split and join. Despite the existence of gateways, there is a lack of tools to support decision making and business rules in BPMN. A DMN diagram is more adapted to represent business rules for example.
Extra specifications
Artifacts are BPMN elements that allow to add some details about the business process or the system.
Ease of use
BPMN is very intuitive. It is easy to understand for final users, even if they have no BPMN knowledge. However, the complete BPMN counts 52 graphical elements
Time & cost
Time and cost in BPMN are clearly acceptable and are not excessive. BPMN is a reliable notation.
Main benefits
BPMN allows to illustrate things that are commonly not visible to the public. BPMN is a notation that is independent from any methodology: The analyst has the freedom to add details with a lot of precisions or not.
Supporting tools
BPMN is supporting by a huge number of guidelines about the design of the diagrams and the way to create them. Moreover, there exists a lot of online tools to create BPMN diagrams (Signavio, bpmn.io, lucidchart, etc.)

Compatibility with other models
BPMN is close to UML activity diagram but with less constraints.
UML
The fame
UML is a standard and universal **LANGUAGE** to specify, visualize, build, and document software system artifacts.
Requirements Phase
Early-phase requirements Late-phase requirements UML is very complete and can with several models cover the 2 phases
Ontology
Static Dynamic Social UML is very large because it is composed by 9 different models. If the analyst wants to represent a static system, he can use class diagram for example. If he wants to model a dynamic system with social interactions, he can use activity or sequence diagram for example.
Approach
UML models are globally model- and operation-based. However, it is possible to integrate some behavioral and temporal aspects of a system, even in static view.
Static or Dynamic systems?
Static and dynamic systems are modellable.
Covered type(s) of requirements
Functional requirements Non-functional requirements ➔ It depends on the chosen UML model.
Application domains
UML is powerful for software intensive system. Some application domains are: ○ Banking ○ Finance ○ Telecommunications ○ Transportation ○ Defense ○ Retail ○ Medical electronics ○ Web-based services But can also represent systems that are not destined to a software, like: ○ Legal system ○ Behavioral system
Public
UML is a very famous language, used by a huge number of people (professional like developers or analysts but also lambda person who wants to represent a system taking only few UML concepts).
Maintenance/extension
UML is frequently extended and is maintained by the OMG group. UML is then still relevant today.
Number of required models
It depends clearly on the choice of the analysts and the system they have to represent. With class and object diagrams, only 1 model is sufficient. With use case and activity diagrams, it is better to split the system in subsystem to get clearer

models. Sequence diagrams can be split if the system is very complex.
Errors handling/security
UML includes concepts like “happy day” and “rain day” scenarios that allow to represent errors or abnormal processes in the system.
Cardinality concepts
Depending on which UML diagram is chosen, it is possible to represent cardinalities of relationships. Class diagram allows that.
Behavioral concepts (intentions, capabilities, beliefs, responsibility, etc.)
Swimlanes, present in several UML models, allow to separate distinctly actors and show responsibilities.
Scenario
Best UML models to represent repetitive scenarios are use-case diagrams, sequence diagrams and activity diagrams. Pay attention that all the UML models do not allow to represent scenarios: Class diagrams, object diagrams, etc.
Decision making
Class diagrams include the concept of node that allow to illustrate decision making. In activity diagram, there is a concept called decision point that allows to represent decision making.
Extra specifications
Annotations is an example of UML concepts that allow to add some details on a model about the system.
Ease of use
Although UML is composed by 9 different models, it stays quite easy to understand it. It nevertheless requires a small learning of the language.
Time & Cost
The time and cost to create UML models depends on which models you chose to represent. However, the conception and maintenance time of models are disadvantages in UML. For example, class diagrams must be synchronized with the code implementation. This activity takes time because usually developers directly work with code and schemas.
Main benefits
In UML, there are enough available models that everything is representable. Moreover, tough tha UML is very famous among developers and analysts, it is not standardized. It means that you can represent a system just the way you want it, without clearly using the real UML concepts and elements. UML is not very rigorous according who is using it.
Supporting tools
There exist lots of off and online tools to create UML models. For example, *figure 54* was created on VisualParadigm Online.
Compatibility with other models
UML is often used to acquire specifications but also directly before implementing a system. It is compatible with a lot of other models like BPMN, KAOS, Entity-Relationships model, etc.

8.2. Recommendations and tips

8.2.1. KAOS and iStar

KAOS and iStar are not very different and have globally the same goal. However, there are several differences that are important to know before modelling.

First, KAOS is directly focus on the goal concept (also shared by iStar) and the goals are refined into subgoals to make them operationalizable. iStar is not directly focus on goal

concept but on the dependencies between actors because the first step is to create the SD model that is more focus on dependencies than goals. It means that where KAOS is focus on goals, iStar is focus on the agents' intentionality and dependencies.

Secondly, KAOS provides a better treatment of conflicts and security dimension than iStar.

Thirdly, there is a difference in the way that KAOS and iStar represent non-functional requirements. In KAOS, non-functional requirements are in fact refined goals while they are represented as softgoals in iStar. Moreover, the softgoals concepts, only present in iStar, allow to add some details about non-functional requirements, with positive and negative dimensions.

Fourthly, iStar provides a distinction between the types of actors (agent-role-position) while KAOS brings any detail about actors.

Fifthly, KAOS handles better the task concept than iStar. While this last one only represents tasks by showing the dependency between 2 actors with *depender-dependee* concept, KAOS is more focus. KAOS links the task with the actor that realizes it and provides more details as the triggering event or the goal that the task allows to achieve.

Finally, KAOS is a better way to model potential conflicts because KAOS considers possible obstacles of a process or a system. In iStar, risk is only analysed through the rationale inside actors.

8.2.2. Tropos

Tropos is also a goal and agent-oriented model but is less famous than KAOS and iStar. The real benefit generated by Tropos is that the methodology covers the early-phase of requirements. It allows to get a deep understanding of the environment and not about just the system itself. Another advantage of Tropos is that it covers the full range of software development phases, from requirements to implementation.

8.2.3. Z Notation

Z Notation is a way to describe computing systems through mathematical text. Z Notation is then very useful when an analyst has to represent a system or capture requirements for future procedural, object-oriented or imperative programming language. Using Z Notation to illustrate and describe physical systems including some storage elements is also very interesting and powerful.

8.2.4. BPMN

BPMN is a very powerful and reliable way to illustrate business process. It means that the workflow is the main concept of BPMN. Using BPMN is then very interesting to visualize, document, analyse and discuss a process using a common notation. Using BPMN is recommended when the analyst is dealing with some troubles on other workflow diagrams because BPMN is intuitive. When an analyst must model a process from scratch or wants to improve existing process (without documentations or not documented), BPMN is a good solution. Modelling in BPMN is often to support the willingness to improve a process. Finally, another interesting usage of BPMN is the requirements elicitation for a process or an application.

8.2.5. UML

The most important thing in UML is the choice of the model, and it is the responsibility of the analyst. If he chooses the wrong model to illustrate a system, he will miss details, will waste time and money and will be focus on irrelevant issues.

The UML models can be differentiated for their own usage. In order to capture requirements, use case diagrams are recommended. If the goal is to structure the system, it would be better to use a class diagram or a component diagram. Finally, if the analyst wants to show the system behavior, he will use a sequence diagram, an activity diagram, or a state-machine diagram.

The level of precision of the model, whatever its type, depends also on the analyst's choice.

Finally, a right, powerful and efficient usage of UML is to represent a system, a process, or requirements by creating separately different UML models and then studying them together.

To conclude, the choice of the language, notation, methodology or framework that the analyst will use to represent requirements or a system depends on several major features (e.g. the size, the goal, the expected cost, the available time, the phase of development, the phase of requirements, etc.). This master thesis allows to get description of 6 famous models and gives relevant information about different situations in which each model can be used in state of another one. A very important aspects of Requirements Engineering is the behavior of the analyst who is in charge of the project. Creating models is a human task that takes into account personal choices.

References

Booch, G. (2005). *The unified modeling language user guide.*

Bresciani, P., Giorgini, P., Giunchiglia, F., Mylopoulos, J., & Perini, A. (2002). TROPOS : An Agent-Oriented Software Development Methodology. *AAMAS Journal*, 203-236.

Castro, J., Kolp, M., & Mylopoulos, J. (2001). A Requirements-Driven Development Methodology. *International Conference on Advanced Information Systems Engineering*, (pp. 108-123).

Chinosi, M., & Trombetta, A. (2012). BPMN: An introduction to the standard . *Computer Standard & Interfaces* , 124-134.

Corradini, F., Ferrari, A., Fornari, F., Gnesi, S., Polini, A., Re, B., & Spagnolo, G. (2018). A Guidelines Framework for Understandable BPMN Models . In *Data & Knowledge Engineering* (pp. 129-154).

Dardenne, A., Lamsweerde, A. v., & Fickas, S. (April 1993). Goal-directed requirements acquisition. In *Science of Computer Programming* (pp. 3-50).

Darimont, R., Delor, E., Massonet, P., & Lamsweerde, A. v. (1998). GRAIL/KAOS : an environment for goal-driven requirements engineering. *Proceedings ICSE*.

Eric YU, P. D. (1995). From Organization Models to System Requirements: A "Cooperating Agents" Approach. *Conference on Cooperative Information Systems* , 9-12.

Fabiano Dalpiaz, X. F. (2016). *iStar 2.0 Language Guide.*

Geambasu, C. V. (2012). BPMN vs. UML Activity Diagram For Business Process Modeling. *Journal of Accounting and Management Information Systems*, 637-651.

Giorgini, P., Kolp, M., Mylopoulos, J., & Pistore, M. (2004). The Tropos Methodology : An Overview. In *Methodologies and software engineering for agent systems* (pp. 89-106). Boston.

Hanandeh, F. A., Alsmadi, I., Al-Shannag, M. Y., & Al-Daoud, E. (2005). Mobile agents modelling using UML. *Int. J. Business Information Systems*, 419-432.

IEEE. (1998). Recommended Practice for Software Requirements Specification. New-York.

JDN, L. r. (2020, 05 12). *Notation Z: définition et utilisation.* Retrieved from Journal du Net: www.journaldunet.fr/web-tech/dictionnaire-du-webmastering/1445198-z-notation-en-langage-informatique-definition-et-utlisation/

Lamsweerde, A. v. (2000). Requirements engineering in the Year 00 : A research perspective. *22nd International Conference on Software Engineering* .

Lamsweerde, A. v. (2001). Goal-oriented requirements engineering: A guided tour. In *Proceedings fifth ieee international symposium on requirements engineering* (pp. 249-262).

Lamsweerde, A. v., & Letier, E. (2003). *From Object Orientation to Goal Orientation : A Paradigm Shift for Requirements Engineering.*

Lamsweerde, A. v., Dardenne, A., Delcourt, B., & Dubisy, F. (1991). The KAOS Project : Knowledge Acquisiiton in Automated Specification of Software. *AAAI Spring Symposium Series*, (pp. 59-62). Stanford.

Lamsweerde, A. v., Delcourt, B., Delor, E., Schayes, M., & Champagne, R. (1988). Generic lifecycle support in the ALMA environment. *IEEE Trans. Software Engineering* , 720-741.

Lapouchnian, A. (2005). *Goal-Oriented Requirements Engineering : An Overview of the Current Research.* University of Toronto.

Levina, O. (2012). Assessing Information Loss in EPC to BPMN Business Porcess Model Transformation . *IEEE 16th International Enterprise Distributed Object Computing Conference Workshops* , (pp. 51-55). Pékin.

Marcelino-Jesus, E., Sarraipa, J., Agostinho, C., & Jardim-Gonçalves, a. R. (2014). *A Requirements Engineering Methodology for Technological Innovations Assesment.*

Mylopoulos, J., Chung, L., & Nixon, B. (1992). Representing and using nonfunctional requirements : a process-oriented approach. *IEEE Transaction on Software Engineering*, 483-497.

Oliveira, A. d., Cysneiros, L., Leite, J., Figueiredo, E., & Lucena, C. J. (2006). Integrating scenarios, i*, and AspecT in the context of mutli-agents systems. 204-218.

OMG. (2011, 12). *Business Process Model and Notation*. Retrieved from OMG : www.omg.org/spec/BPMN/2.0/

QRA. (2020). *Functional vs Non-functional Requirements*. Retrieved from QRA corp: https://www.qracorp.com/functional-vs-non-functional-requirements/

Quatrani, T. (2003). *Introductiont to the Unified Modeling Language.*

RESPECT-IT. (2007). *KAOS Tutorial.*

Robinson, W. (1989). Integrating multiple specifications using domain goals. *Proceedings 5th International Workshop on Software Specification and Design*, (pp. 219-226). Pittsburgh.

Silva, V. T., Noya, R. C., & Lucena, C. J. (2006). *Using the UML 2.0 Activity Diagram to Model Agent Plans and Actions.* Rio de Janeiro.

Silvers, B. (2009). *BPMN Method & Style .*

Snoeck, M. (n.d.). BPMN 2.0 : Advanced Modelling . Research Group of Management Information Systems : Faculty of Business and Economics , KU Leuven .

Snoeck, M., Oca, I. M.-M., Haegemans, T., Scheldeman, B., & Hoste, T. (2015). Testing a Selection of BPMN Tools for theyr support of modelling guidelines. *IFIP Working Conference on The Practice of Enterprise Modeling* , (pp. 111-125).

Spivey, & Abrial. (1992). *The Z Notation.*

Teruel, M. A., Navarro, E., opez-Jaquero, V., Montero, F., & Gonzalez, P. (2012). Comparing Goal-Oriented Approaches to Model Requirements for CSCW. *LoUISE Research Group*, 169-184.

The Tropos Crowd. (n.d.). *Tropos Project*. Retrieved from Tropos Project: www.troposproject.eu

Universalis. (2020). *Universalis : Systèmes informatiques* . Retrieved from Universalis: www.universalis.fr/encyclopedie

web, C. I. (n.d.). *Les inconvénients de UML.* Retrieved from Connaissances Informatiques: www.ordinateur.cc/logiciel/autres-logiciels-informatiques/144820

Werneck, V. M., Oliveira, A. d., & Leite, J. C. (2009). Comparing GORE Frameworks : i-star and KAOS.

White, S. (2004). Introduction to BPMN. *Ibm Cooperation 2.0.*

Wikipédia. (2014). *Business Process Model and Notation.* Retrieved from Wikipédia: www.en.wikipedia.org/Business_Process_Model_and_Notation

Xavier Franch, L. L. (2016). The i* framework for goal-oriented modeling. In *Domain-specific conceptual modeling* (pp. 485-506).

YU, E. (1997). Towards Modelling and Reasoning Support for Early-Phase Requirements Engineering. *3rd IEEE International Sympoiu on Requirements Engineering* , 226-235.

YU, E. (2009). Social Modeling and i*. In *Conceptual modeling : Foundations and applications : Essays in honor of John Mylopoulos* (pp. 99-121).

Yu, E., & Mylopoulos, J. (1998). Why goal-oriented requirements engineering. *Proceedings of the 4th International Workshop on Requirements Engineering*, (pp. 15-22).

Zowghi, D., & Coulin, C. (2005). Requirements Elicitation : A Survey of Techniques, Approaches, and Tools. In A. Aurum, & C. Wholin, *Engineering and Managing Sofware Requirements* (pp. 19-46).