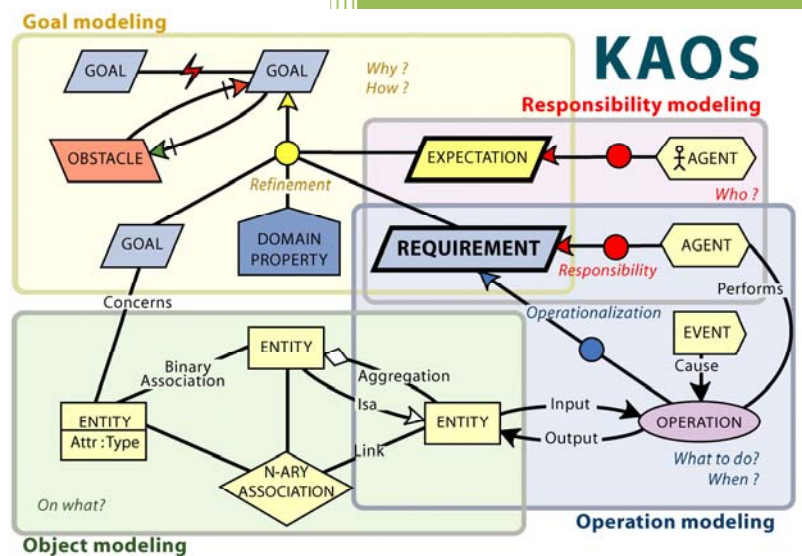


A KAOS Tutorial



Tutorial

V1.0

Oct. 18, 2007

Revision Table

Version	Date	Modified pages	Modified sections	Comments

Table of Contents

TABLE OF CONTENTS.....	3
1. INTRODUCTION.....	5
1.1. FOREWORD	5
1.2. OUTLINE	5
1.3. WHO'S WHO	5
1.4. GETTING MORE INFORMATION ABOUT THE KAOS APPROACH	5
2. KEY IDEAS UNDERLYING KAOS.....	6
2.1. BUILD A REQUIREMENTS MODEL	6
2.2. JUSTIFY YOUR REQUIREMENTS BY LINKING THEM TO HIGHER-LEVEL GOALS	6
2.3. BUILD A MODEL OF THE WHOLE SYSTEM, NOT JUST THE SOFTWARE PART OF IT	7
2.4. BUILD A RESPONSIBILITY MODEL	7
2.5. BUILD A CONSISTENT AND COMPLETE GLOSSARY OF ALL THE PROBLEM-RELATED TERMS YOU USE TO WRITE THE REQUIREMENTS	8
2.6. DESCRIBE HOW THE AGENTS NEED TO BEHAVE IN ORDER TO SATISFY THE REQUIREMENTS THEY ARE RESPONSIBLE FOR	8
2.7. BASE THE REQUIREMENTS DOCUMENT ON THE REQUIREMENTS MODEL	9
2.8. VALIDATE YOUR REQUIREMENTS BY FIRST REVIEWING THE MODEL	9
2.9. USE A DEFENSIVE APPROACH TO THE BUILDING OF A REQUIREMENTS MODEL	9
3. THE ELEVATOR CASE STUDY.....	11
3.1. PROBLEM STATEMENT	11
3.2. GOAL MODEL	11
3.2.1. REQUIREMENTS PATTERNS.....	11
3.2.2. APPLICATION TO THE ELEVATOR PROBLEM.....	13
3.3. RESPONSIBILITY MODEL	23
3.4. OBJECT MODEL	26
3.5. OPERATION MODEL	30
3.6. DEALING WITH OBSTACLES	34
3.7. REQUIREMENTS DOCUMENT GENERATION	37
3.7.1. REQUIREMENTS DOCUMENT STRUCTURE.....	37
3.7.2. HOW TO FILL IN THE TEMPLATE FROM A KAOS MODEL.....	39
4. DISCUSSION.....	41
4.1. IS IT WORTH PAYING ATTENTION TO A KAOS ANALYSIS?	41
4.2. TRACEABILITY	41
4.3. COMPLETENESS	41

A KAOS Tutorial

4.4.	NO AMBIGUITY	42
4.5.	FOR WHICH PROJECT SIZE IS IT WORTH THINKING OF USING KAOS ?	42
5.	<u>CONCLUSION.....</u>	<u>44</u>
6.	<u>METHODOLOGY SUMMARY</u>	<u>45</u>
6.1.	KAOS META-MODEL	45
6.2.	KAOS GLOSSARY	45

1. Introduction

1.1. Foreword

KAOS is a methodology for requirements engineering enabling analysts to build requirements models and to derive requirements documents from KAOS models.

In this tutorial, you will learn how to build a KAOS model step by step and how to generate a requirements document based on this model.

Objectiver is a tool designed to support KAOS. All diagrams in this tutorial have been generated by this tool.

1.2. Outline

The tutorial is structured as follows:

- Key ideas underlying KAOS. The section introduces KAOS; it gives the main ideas and processes you have to keep in mind to build a successful requirements document with KAOS.
- Analysis of a case study, where the requirements will be gathered for the design of a new elevator system.

The case study will let us show you how the KAOS approach favors the identification of interesting properties and alternative unexpected designs. But you should keep in mind that KAOS can be used for any type of information system. We have been using it for many years in different industries such as steel, mechanics, telecommunication, health care. We also used the KAOS approach for public administrations, ...

1.3. Who's who

KAOS originates from a cooperation between the University of Oregon and the University of Louvain (Belgium) in 1990. Research, extensions and improvements are still being made to the methodology on a regular basis at the University of Louvain.

Respect-IT is a spin-out company of the University of Louvain. Respect-IT has put the method to practice on dozens of industrial cases in different sectors. Respect-IT has built and is now distributing Objectiver, a tool that supports KAOS.

1.4. Getting more information about the KAOS approach

Scientific papers and slide presentations shown during conference keynotes can be found on the following Web site:

<http://www.info.ucl.ac.be/research/projects/AVL/ReqEng.html>

Return on experience and reports on the KAOS approach can be found on Objectiver's web site :

<http://www.objectiver.com/>

2. Key ideas underlying KAOS

Here follow the key ideas you should know before getting into KAOS requirements engineering.

2.1. Build a requirements model

Nowadays more and more development teams appreciate modeling techniques for specifying solutions. The first key idea behind KAOS is to build a model for the requirements, that is, for describing the problem to be solved and the constraints that must be fulfilled by any solution provider. KAOS has been designed :

- to fit problem descriptions by allowing you to define and manipulate concepts relevant to problem description,
- to improve the problem analysis process by providing a systematic approach for discovering and structuring requirements
- to clarify the responsibilities of all the project stakeholders
- to let the stakeholders communicate easily and efficiently about the requirements.

2.2. Justify your requirements by linking them to higher-level goals

Goals are desired system properties that have been expressed by some stakeholder(s). Here's for instance a goal excerpted from the Elevator case study:

“Each time a passenger calls an elevator from floor f_1 to go to floor f_2 , the elevator system eventually takes him to f_2 .”

With KAOS, the analysts discover the new system goals by interviewing current and future users and by analysing the existing systems, reading the available technical documents, etc... KAOS enables the analysts to structure the collected goals into directed, acyclic graphs so that:

- each goal in the model (except the roots -- the top-most strategical goals) is typically **justified** by at least another goal that explains **why** the goal was introduced in the model
- each goal (except the leaves, the bottom goals) is **refined** as a collection of subgoals describing **how** the refined goal can be reached.

Near the top of the graph stand business or strategical goals. At the bottom (leaves) stand system requirements. The example given above is a business goal: it gives a property pertaining to the core business of an elevator system. It can't be considered to be a requirement : the “eventually” qualifier is not precise enough. Here's a better candidate to be a requirement :

“When there's a call for an elevator on floor f , the first elevator passing by floor f and heading in the requested direction will be stopped at floor f , except when full.”

where the concept of a ‘full elevator’ must be defined somewhere else in the requirements document.

Business and strategical goals are expressed in terms of the stakeholders' vocabulary. Lower-level goals are typically expressed using words from the stakeholders' vocabulary as well as

specific technical terms introduced in the model on purpose and where necessary.

Identifying goals is not proceeding exclusively from either a top-down (going from higher business goals to lower technical requirements) or a bottom-up approach. In most cases the two approaches shall be used at the same time. Often analysts start by unveiling intermediate goals first. Then they proceed by looking for higher-level (strategical) reasons to each new goal (by asking “**why** do we want that ?”). To also discover more specific subgoals, analysts have to ask themselves questions such as “**how** shall we attain that objective ?”.

In real systems, some goals can be **conflicting**. Goals are conflicting if the system can reach a state in which it is not possible to satisfy both goals simultaneously. For instance, performance goals may conflict with safety goals; information goals may conflict with security and privacy goals. Desires from different user roles may also conflict. It is particularly important to identify any conflicting goals as soon as possible in the software life cycle (the best being during requirements analysis). Dealing with conflicts or more generally with obstacles that prevent goals from being achieved allows you to build a more complete requirements document and a more robust system at the first shot.

The KAOS **Goal Model** is the set of interrelated goal diagrams that have been put together for tackling a particular problem.

2.3. Build a model of the whole system, not just the software part of it

Limiting the scope of a requirements analysis to the software system to build is not enough. The software system is to be used within a specific environment; it is very important to identify, record and take into account all the requirements and assumptions about that part of the environment that interacts with the software system. In the following, information system will stand for the software system to be and the part of the environment with which it interacts.

Requirements on agents interacting with the software system to be developed are known as **expectations** in the goal model. They are introduced to show how the software system and its environment have to cooperate to achieve goals of the information system. It provides a clear separation between the responsibilities set on the software system and those set on the environment. Obstacles can be introduced in the model in order to allow the software system to cope with an interacting environment failing to fulfil the expectations put on it.

2.4. Build a responsibility model

Aside of goals, agents represent an important type of KAOS concepts. Agents are either human beings or automated components that are responsible for achieving requirements and expectations.

In the elevator case, the passenger and the elevator controller are two examples of agents.

KAOS requires analysts to associate each requirement or expectation with an agent responsible for it. In fact, with KAOS:

- a requirement is a low-level type of goal to be achieved by a software agent. The software agent is responsible for it.
- an expectation is a type of goal to be achieved by an agent which is part of the system environment.

In many cases goals are assignable to several agents rather than a single one. With KAOS, a distinction is made between these two possibilities. **Assignment** is used when several agents may be made responsible for some requirement or expectation, whereas **responsibility** is used when there's only one agent who is responsible for it.

Basically, this difference gives the analyst a criterion to stop refining goals into subgoals: refinement is no longer necessary as soon as a goal has been placed under the responsibility of a single agent.

KAOS allows responsibility diagrams to be derived from the model, each diagram showing all the requirements/expectations an agent is responsible for.

The KAOS **responsibility model** is the set of derived responsibility diagrams.

2.5. Build a consistent and complete glossary of all the problem-related terms you use to write the requirements

Standards-compliant (e.g. IEEE830) requirements documents need to include a glossary of all specific terms that are used. It is generally a tedious and boring task to create and maintain a glossary section; consistency and completeness must be preserved.

With KAOS it still is difficult to build the glossary but KAOS allows analysts to work on the glossary progressively and simultaneously during goals and requirements definition by building a KAOS **object model**. An object model contains objects, a.o., agents, **entities** and **relationships** among them. The notation used in the object model complies with the one used in UML for class diagrams.

Examples of entities in the elevator problem are: *elevator* and *floor*.

Examples of relationships are:

- $at(l,f)$ holding iff *elevator* l is currently located on *floor* f
- $call(p,f)$ holding iff an *elevator* has been called from *floor* f by *passenger* p and the call has not yet been cleared by the system.

Requirements documents glossaries are built by traversing the object model and listing all the concepts it contains.

2.6. Describe how the agents need to behave in order to satisfy the requirements they are responsible for

Software agents are responsible for requirements. Agents also have capabilities. The **operation model** with KAOS sums up all the behaviors that agents need to have to fulfil their requirements. Behaviors are expressed in terms of operations performed by agents. Those operations work on objects described in the object model: they can create objects, provoke object state transitions or trigger other operations through sent and received **events**.

Examples of operations performed by the passengers in the elevator problem are: *press button*, *enter*, and *exit*; examples of operations performed by the elevator are: *open doors*, *close doors*, *move up*, and *move down*.

A KAOS operation diagram typically composes operations performed by one or several agents to achieve a requirement. Compositions are made through data flows (the output of an operation output becomes the input of another operation) or control flow (an event sent by an operation triggers or stops another operation). An operation diagram thus describes how the

agents need to cooperate in order to make the system work. With KAOS, the operation model is connected to the goal model: the analysts justify operations by the goals they “operationalize”. An operation with no justification means that either there’re still missing goals in the model or that the operation is not necessary. Conversely if some requirements are left without “operationalization”, they may just be wishful thinking.

2.7. Base the requirements document on the requirements model

Modeling with KAOS is full of benefits: the analyst can limit himself to the essential questions; it’s easier to keep things simple and clear and to communicate the project requirements efficiently.

At the end of a requirements analysis, what you typically need is not a model but a requirements document. With KAOS, the requirements document is built based on a template document. Information is extracted from the model to fill the template. For instance, the glossary section is built by traversing the object model. The requirements are inserted in the document by traversing the goal model from top (the business/strategical goals) to bottom (the requirements). Requirements on the system architecture are derived from the responsibility model and requirements for the system behavior from the operation model.

The result is a consistent, unambiguous and complete document. The document’s consistency and completeness are guaranteed by the underlying KAOS model. Changes are made within the KAOS model, not in the documents that are output from the model. Documents can be regenerated as needed to reflect the latest changes to the model. Within a KAOS model, each concept is defined only once (no ambiguity, no contradiction). If the analyst follows the KAOS methodology (each goal is justified by higher-level goals and refined into requirements, requirements are placed under the responsibility of agents and operationalized in the operation model), the requirements document will be complete with respect to the identified goals. Completeness of the goal model is ensured by reviewing the different diagrams during validation sessions with stakeholders (domain experts, users, ...).

2.8. Validate your requirements by first reviewing the model

Our long experience of building high-quality requirements documents has proven that it’s more efficient to validate the requirements by organizing collective reviews of the Kaos model rather than asking people to read a long technical document on their own, and then to set up meetings to discuss their remarks. Virtual meetings using shared screens on distant stakeholder’s locations can also be used. The KAOS model reveals to be a remarkable mean to communicate about the requirements. Many companies have noticed that users and IT analysts most often do not understand each other very well. KAOS provides the right connection between the two worlds: users quickly feel confident with goal and responsibility models; analysts like the object and operation models.

2.9. Use a defensive approach to the building of a requirements model

The idea is similar to “defensive programming”. It consists in investigating in a systematic way what could go wrong in the system-to-be, that is how and why some requirements could no longer be satisfied. Circumstances under which such cases can occur are named **obstacles**.

A KAOS Tutorial

When obstacles are discovered, the analysts can follow different strategies to address them, e.g., add new requirements that would prevent the obstacle from occurring, mitigate the impact of unavoidable obstacle on the system, etc.

Exceptions or degraded operation modes can be specified in this way.

Consider your requirements document as a reference that shall need updating during the project development life cycle

A common mistake is to consider a requirements document as a frozen document used for negotiating with solution providers and no longer updated after that time. Typically requirements are changing during the development phasis and sometimes in other phases as well. The requirements document should therefore be updated as needed during the entire software life cycle. One of the most frequent cause of project failure is the inability to tackle the problem of changing requirements. The KAOS requirements model underneath the requirements document clearly helps anyone to analyze the impact of requirements changes, that is, to determine the other changes that are implied by a new change request. KAOS provides a high-level view on the system-to-be: what it does, on what, why, how, by whom and when.

KAOS is independent of the type of development model you decide on using: waterfall, iterative, incremental, ... In the waterfall model, for instance, a large amount of time is spent at the beginning to acquire all requirements and write a requirements document that will become the reference for development. Sometimes 'time to market' constraints or the unavailability of precise requirements from the beginning compel development teams to follow an iterative and incremental development process. KAOS models can be developed and updated incrementally to reflect the last requirements acquired for instance by means of prototypes, simulation or other acquisition techniques. Keeping a model up to date and deriving a new requirements document from there is definitively much easier that directly trying to keep a requirements document up to date.

Most of the applications are built to last, sometimes well longer than initially expected. After some years, people who are in charge of maintaining the application won't probably be those who have developed or maintained the system in the early days. Having an up-to-date requirements model will improve the system understandability by new developers and allow them to capitalize on the past.

3. The Elevator case study

This section aims at investigating the requirements analysis for a well-known system: an elevator system. After stating the problem, this tutorial guides you through the construction of the four KAOS models: the goal, responsibility, object, and operation models. Then it investigates how threats on the system can be modeled and used to make the model more robust. Finally, we will present a systematic way to derive a requirements document from a KAOS model.

3.1. Problem statement

You've just been hired by an elevator design company to improve performance and quality of software development within the company. You've directly pointed out a major weakness in the way software is developed : there is currently no formal requirements engineering method in use. As a first challenge, you are asked to build a KAOS model for a new elevator system to be designed.

3.2. Goal model

3.2.1. Requirements patterns

Requirements can be gathered by means of open interviews. A more efficient way to gather requirements is to conduct less open interviews by reusing requirements patterns. One of the long-term benefits of investing in KAOS technology consists in progressively modelling generic patterns of requirements. These patterns can be used on new cases to guide the identification of requirements. Figure 1 shows such a pattern.

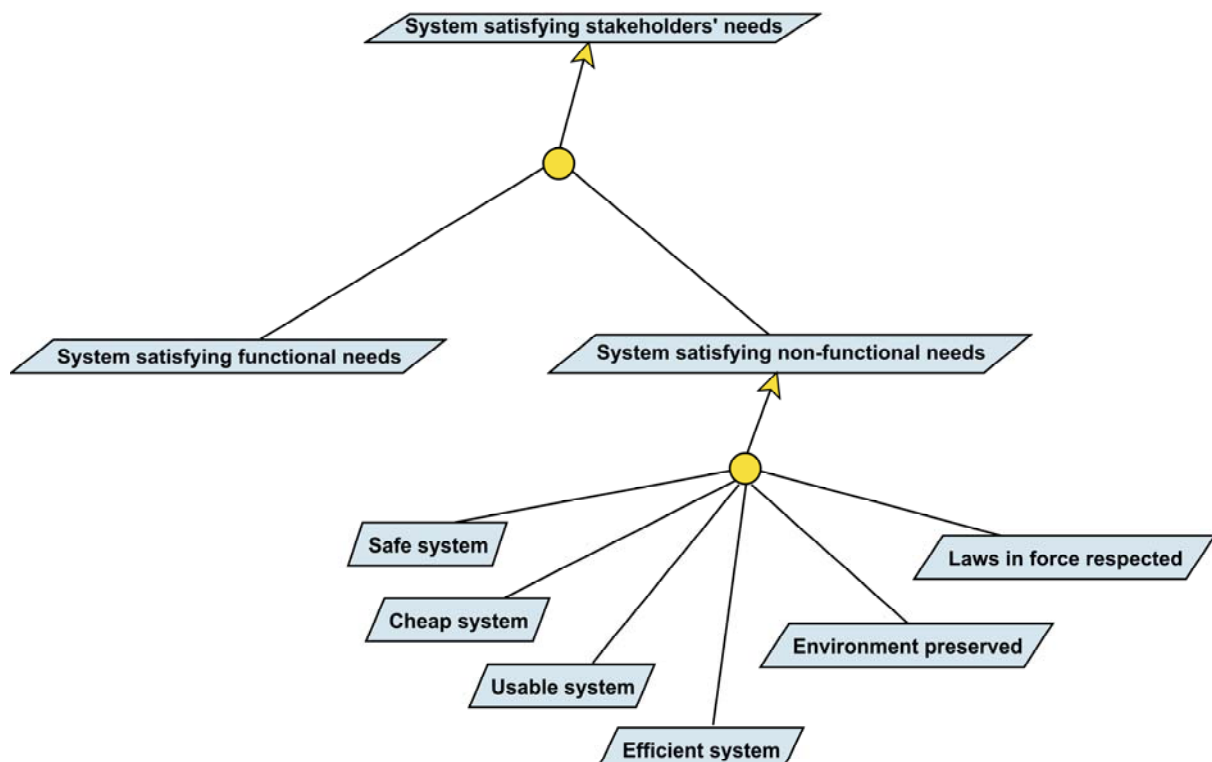


Figure 1 Generic Goal Pattern

A KAOS Tutorial

Each parallelogram in the figure represents a goal. Yellow circles represent refinements of a parent goal (the one pointed to by the yellow arrow) and a list of subgoals. The diagram can be read as follows:

The goal of the system is to build a system that satisfies all stakeholders' needs: functional and non-functional ones. Non-functional goals are classified as follows:

- *the system must be safe, cheap, usable, and efficient;*
- *the system must preserve its environment;*
- *the system must respect the laws in force.*

As we will see later on, there are different tactics for decomposing goals into subgoals. The tactic that was used here is a **case-driven decomposition**: the subgoals enumerate all the cases that must be covered to fulfill the father goal. For instance, functional and non-functional needs together cover all the needs.

Each leaf in the above diagram can be decomposed in turn into subgoals. For instance, the following goal diagram investigates the economical aspects; the tactic used here is a **milestone-driven decomposition**: first systems are built, then they must be run and maintained. The figure shows also a **conflict** between two goals: “System cheap to build” and it must be “Robust and reliable”. Reducing costs during system construction can badly impact robustness and reliability of the resulting system; conversely, improving robustness and reliability of the system could increase significantly the global cost of building the system.

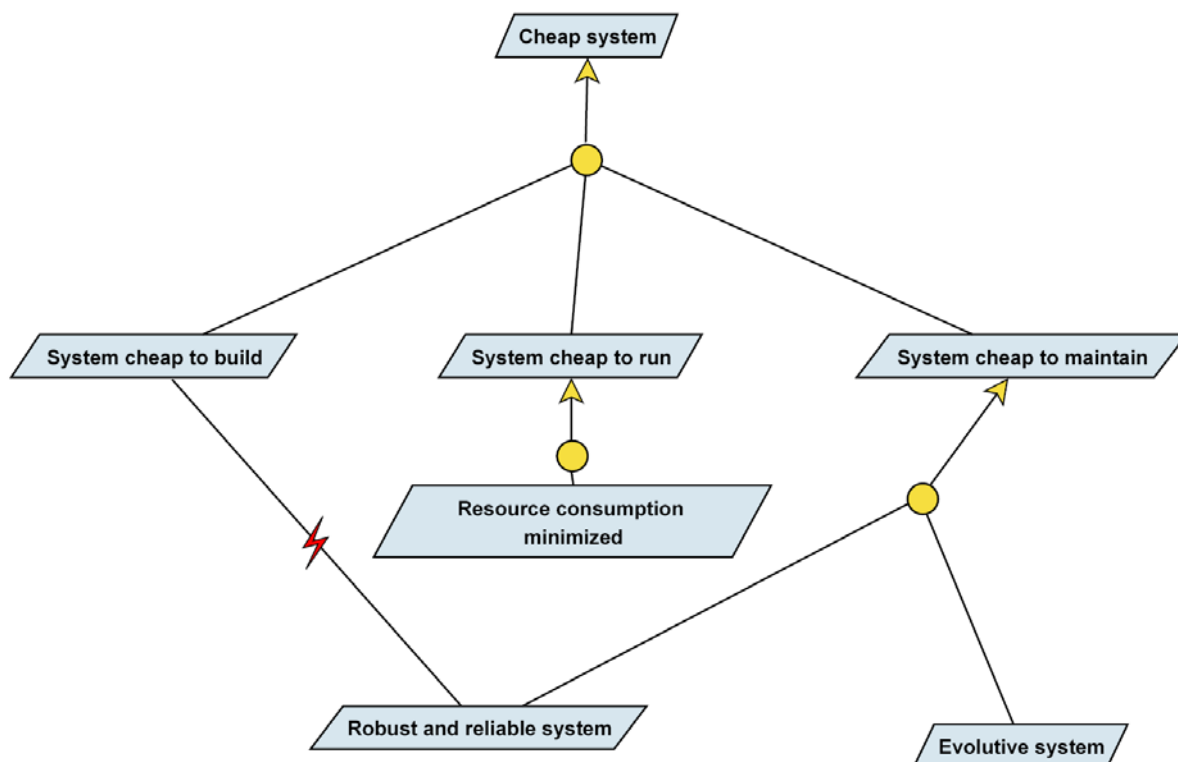


Figure 2 Generic Goal Pattern

A KAOS goal model is a directed graph (which is more general than a simple tree), which means that a given goal can appear on different diagrams to refine different higher-level goals. For instance, the goal “Robust and reliable system” appears on Figure 3 and Figure 4. This goal contributes to the safety, usability, and cheap maintenance goals for the system. It’s

important to record in the model all the different reasons for a goal to be needed. It eases future impact analyses as the model makes it clear that reducing robustness or reliability of the system can have an impact on the system usability, safety and maintenance costs.

Other useful patterns will be introduced during our case study.

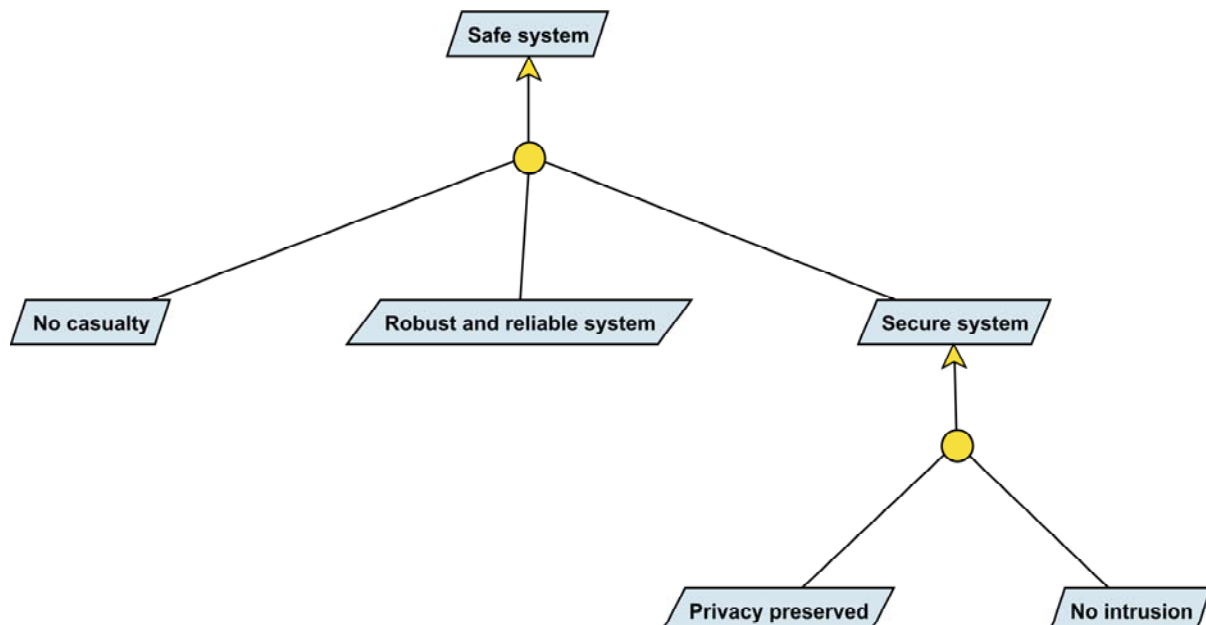


Figure 3 Generic goals: "Safe System"

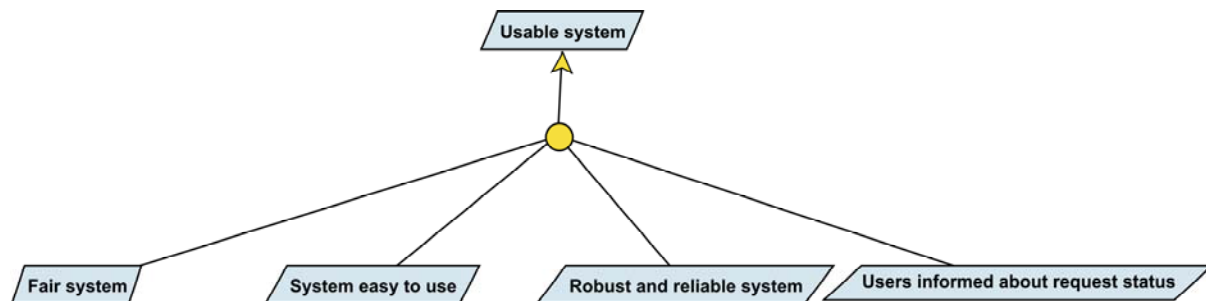


Figure 4 Generic goals: "Usable system"

3.2.2. Application to the elevator problem

Let's begin our problem analysis by instantiating the generic pattern above. What does 'System satisfying functional needs' mean in the context of the elevator problem? It means that all transportation requests must be satisfied. As for the non-functional goals, we will forget about the following goals that do not seem to be relevant in this case: "Environment preserved"; "Laws in force preserved", though one or the other might have to be preserved in a real project. Figure 5 shows our first candidate goal diagram.

The scope of this tutorial does not permit to cover each goal in details. In the following sections, we will study the following goals:

- "Transportation requests satisfied"

- “Safe elevator system”

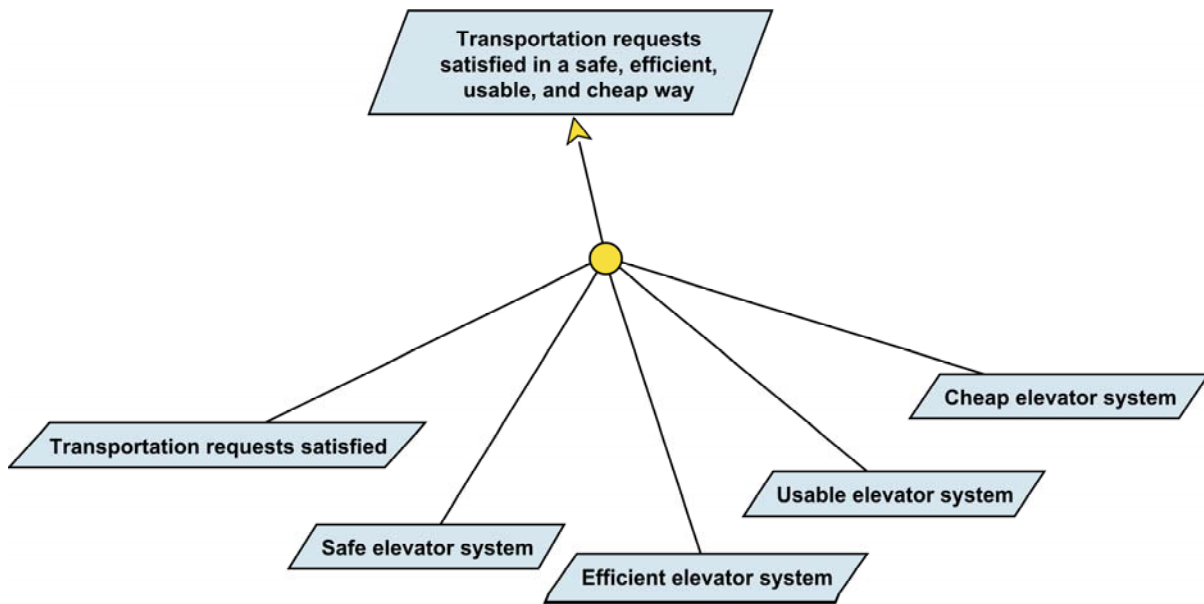


Figure 5

3.2.2.1. Transportation requests satisfied

Transportation to destination is the service expected by passengers. A generic pattern for service requests is the following one (Figure 6):

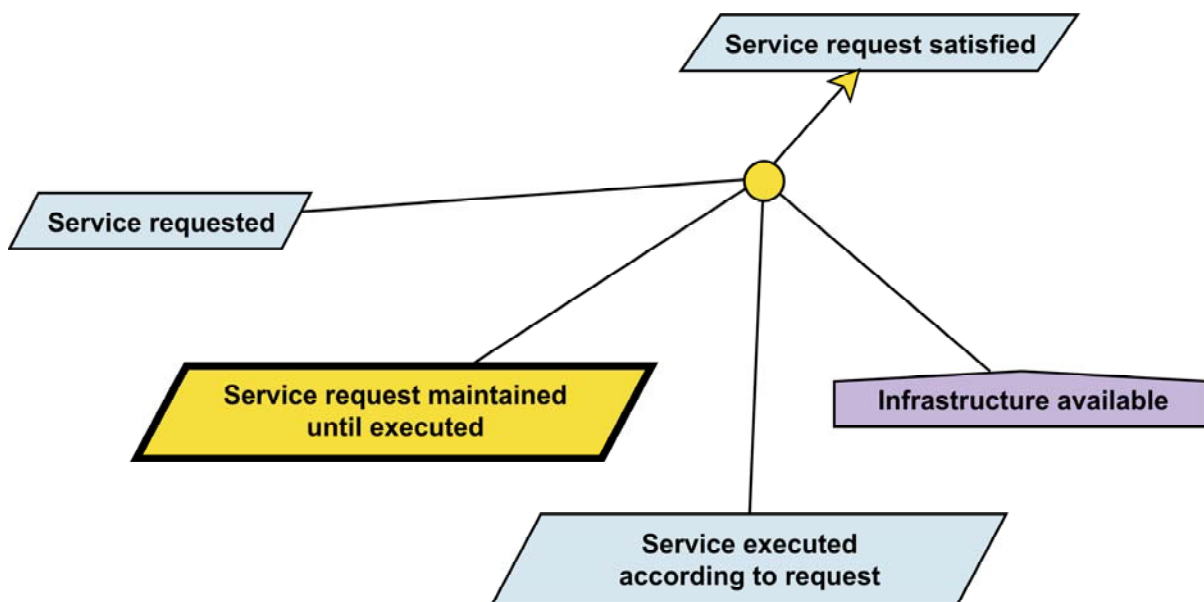


Figure 6. Generic Pattern

The pattern should be read as follows:

To satisfy a request for a given service, the passenger must issue a request and maintain it until executed; the system must respond by providing a service that satisfies the passenger's request.

The goal “Service request maintained until executed” is in fact an expectation on the environment: we expect passengers not to change their mind until the system is done with it.

Otherwise the top goal could not be achieved. Expectations do not need to be refined further. They can however be questioned by considering obstacles later on during the analysis process (see section on Obstacle Analysis).

Figure 6 introduces another kind of concept: **domain properties**. Domain properties are properties relevant to the application domain. They are used in refinements to prove that a refinement is complete. The domain property on Figure 6 should be read as follows:

In order to satisfy a service request, an infrastructure to perform the service must be available.

They are two types of domain properties:

- **domain hypotheses**. They are domain object properties expected to hold. They can be used when arguing about sufficient completeness of goal refinements. For instance, a domain hypothesis for the elevator system could be: *“the elevator system has at least one cage to carry passengers”*.
- **domain invariants**. They are properties known to hold in every state of some domain object, e.g. a physical law, regulation, or a constraint enforced by some environmental agent. For instance, for light buttons, we could state that *“the light is either on or off”*.

When creating KAOS models naming conventions are strongly recommended. Look at the previous figure and notice the way goals have been named: a word followed by verb in its passive form. For instance, we have written *“Service requested”* instead of *“Request service”* or *“The passenger must request the service”*. The reason is to avoid confusion between goals and operations (agent behaviors). Goals basically refer to system states we want to achieve or maintain, cease or avoid. They do not refer to system state transitions. To illustrate that point, let's rephrase the pattern above this way :

To reach the system state in which a request for a service is satisfied, the system has to reach a first state in which the service has been requested, a second state in which a service has been provided in response, in such a way that the service satisfies the request (the request is also expected to be maintained until the service is provided).

Let's now instantiate and apply that pattern to our elevator problem (see Figure 7).

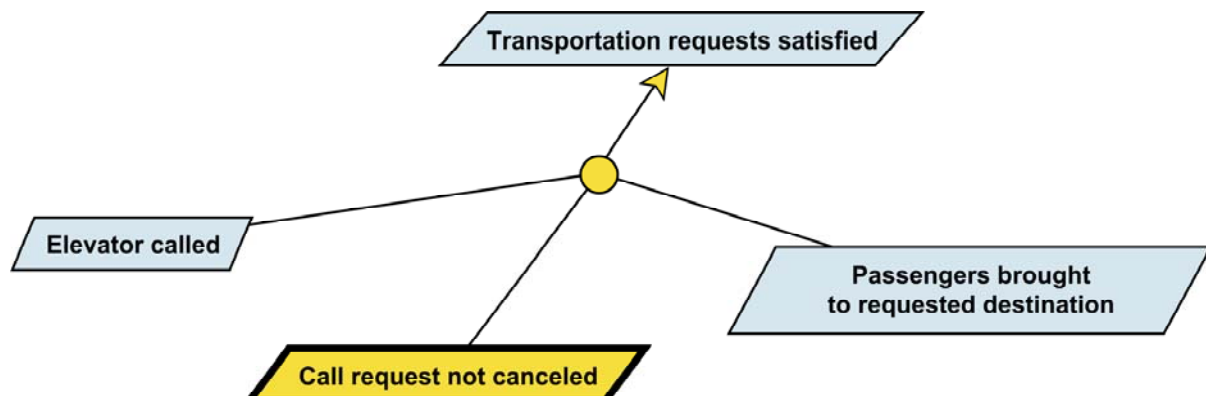


Figure 7. Application of the pattern to the elevator problem

The *“Call request not canceled”* expectation means that the upper goal can't be satisfied if the passenger changes his mind: so, if a passenger calls an elevator, but does not step inside when the doors open, his original request will never be brought to completion.

3.2.2.2. Elevator called

Let's now look at the "Elevator called" goal. A generic pattern also exists for it (Figure 8):

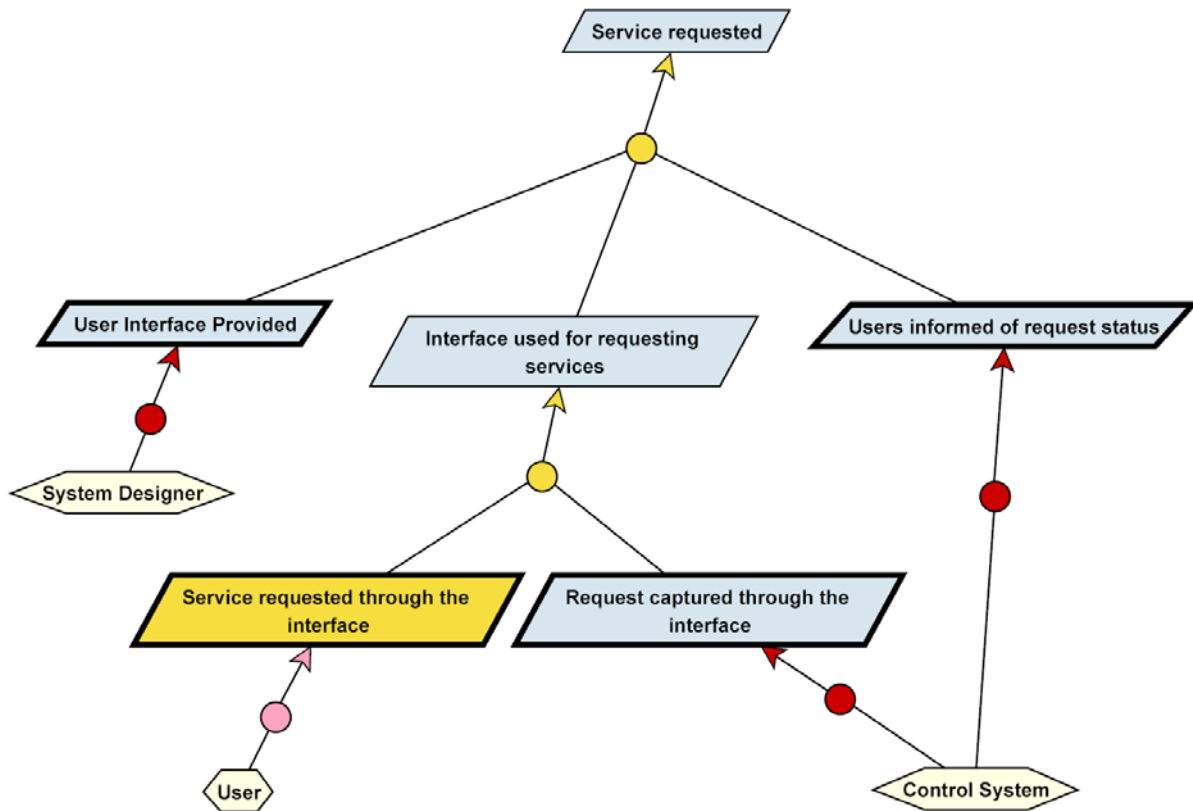


Figure 8. Pattern

We have seen that regular goals are represented as parallelograms. In this diagram, several **requirements** have been used. They are represented as thick-bordered parallelograms. A requirement is in fact a goal which has been placed under the responsibility of an agent. **Agents** are represented as yellow boxes with angle corners: User, System Designer and Control System are the agents on this diagram. Red circles stand for the **responsibility** relationship that connects an agent to a requirement for which the agent is responsible. Pink circles are used for expectation **assignment** to some agent.

Requirements may be refined by other requirements (not goals) but all of these shall implicitly be placed under the responsibility of the agent that is responsible for the father requirement.

With KAOS, a goal model is considered to be complete when all its leaves are either expectations, domain properties or requirements. This gives us a first completeness criterion:

***Completeness criterion 1:** A goal model is said to be complete with respect to the refinement relationship 'if and only if' every leaf goal is either an expectation, a domain property (DomProp) or a requirement.*

Now notice in the diagram above how each requirement has been placed under the responsibility of some agent. For each requirement we know who's responsible for it. In all real projects, it's of crucial importance to identify who's responsible for what and avoid situations in which two agents think the other one is responsible for a given requirement or none of them knows that he's the one to be responsible for the requirement. This gives us a

second completeness criterion:

Completeness criterion 2: *A goal model is complete with respect to the responsibility relationship ‘if and only if’ every requirement is placed under the responsibility of one and only one agent (either explicitly or implicitly if the requirement refines another one which has been placed under the responsibility of some agent).*

Let’s now apply the pattern above to our elevator problem. A classical way to communicate simple requests to a system is by using push buttons. Let’s assume such an interface for our elevator: Figure 9.

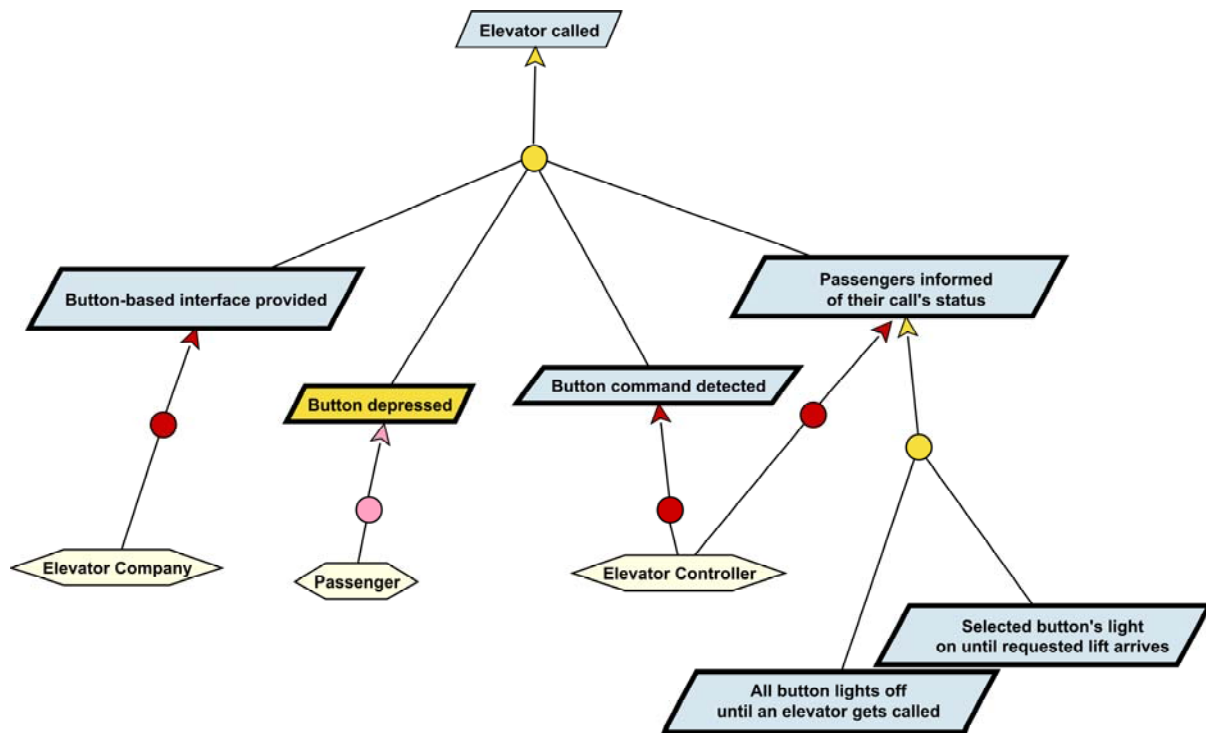


Figure 9 Elevator called

Notice how we refined the requirement “*Passengers informed of their request’s status*” through the use of two subrequirements. Both are implicitly placed under the responsibility of the same agent (the *elevator controller*).

3.2.2.3. Button-based interface provided

The question we now have to deal with is : which kind of interface do we need for the elevator system ? The interface must enable a passenger to tell the elevator system that he wants to go from floor f1 to floor f2. Giving that one can come with the following possibilities :

- A destination selection panel is provided on each floor. A passenger on floor 1 wishing to use the elevator to go to floor 2 pushes the button corresponding to floor 2. No selection panel is needed in the elevator cage as the system already knows where the incoming passengers want to go.
- A two-buttons panel to allow selection of up or down direction is installed on each floor. A destination floor selection panel is installed inside the elevator cage. The passenger presses the button corresponding to the floor where he wants to go.
- A one-button selection panel is installed on each floor. The passenger pushes the

A KAOS Tutorial

button to call the elevator. A destination floor selection panel is installed inside the elevator cage. The passenger presses the button corresponding to the floor where he wants to go.

- An infrared cell is installed on each floor. The cell is able to detect passengers waiting in front of elevators. A destination floor selection panel is installed inside the elevator cage. The passenger presses the button corresponding to the floor where he wants to go.

With Kaos, alternatives for achieving goals are represented by distinct refinements : one yellow circle per alternate solution. The different possibilities discussed above are modelled as shown on Figure 10.

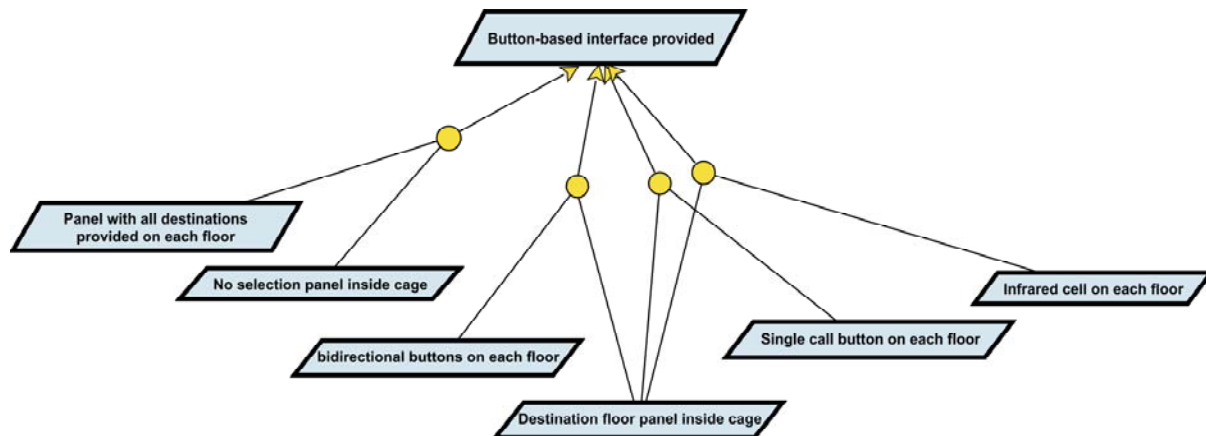


Figure 10. Alternatives: "Button-based interface provided"

In this figure, each refinement (each yellow circle) shows a possible design. Often, selection among the alternatives is made by comparing the alternatives with respect to some qualitative criteria. For instance, the design in which each floor is equipped with a destination floor panel contributes to the elevator system efficiency (in a multi-elevator system, one can optimize the cage movements if the system knows all the requested destinations even before passengers step in the elevator) but it's also the most expensive solution. As for the infrared cell solution, it could be questioned about the reliability of the requests that the system shall receive. There will be cases where people standing in front of elevators are in fact not waiting for an elevator. These cases will also degrade efficiency of the whole system.

Qualitative goals (and more generally all non-functional goals) should be included to the goal model (see Figure 1 above); the qualitative comparison we made above for the elevator's panel can be described within the model by refining a qualitative goal when a design contributes positively to that goal or by creating a conflict if it contributes negatively. Figure 11 shows this analysis.

According to this analysis, the design having a bidirectional button panel on each floor seems to be the best compromise (no conflict with qualitative goals and positive contributions to the goals of building a cheap and efficient elevator system).

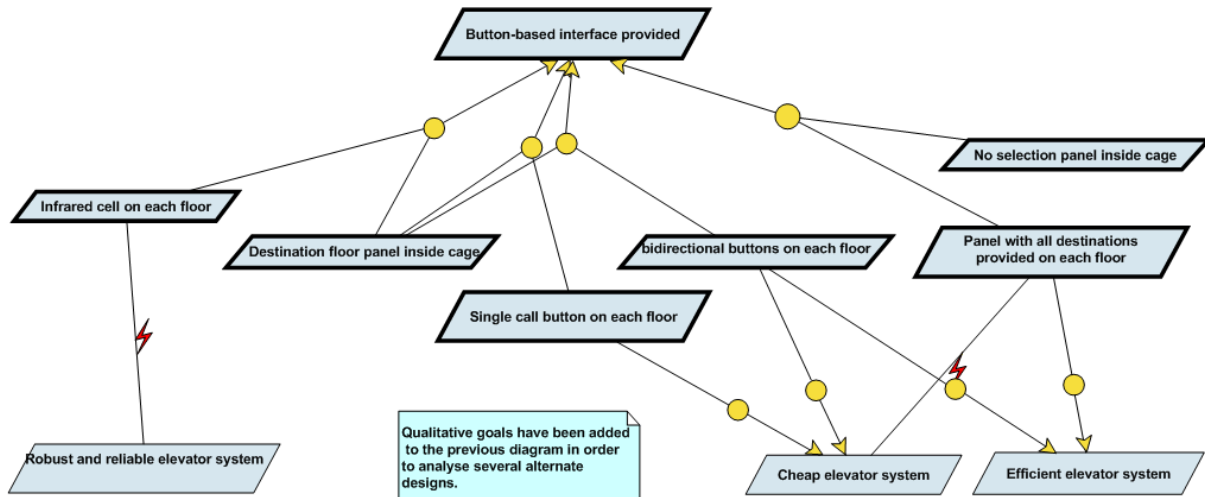


Figure 11. Button-based interface provided

3.2.2.4. Button depressed

Based on the bidirectional button panel design, we can now refine the goal “Button depressed” which appears on Figure 9. The revised goal diagram is shown on Figure 12.

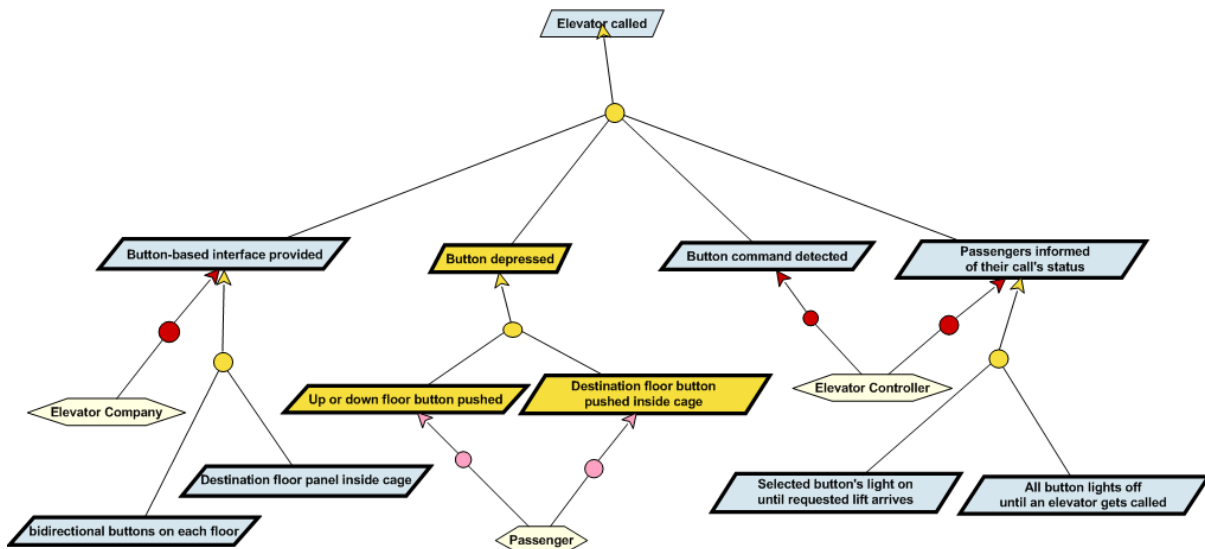


Figure 12. Elevator called (for selected design)

3.2.2.5. Passengers brought to requested destination

Let’s now return to Figure 7 to refine the goal “*Passengers brought to requested destination*”. Applying the milestone tactic introduces subgoals requiring that the cage must eventually stop at the floor from where the passenger has called, the passenger has to enter the cage, select a destination floor, and the cage has to move and stop to requested destination (Figure 13).

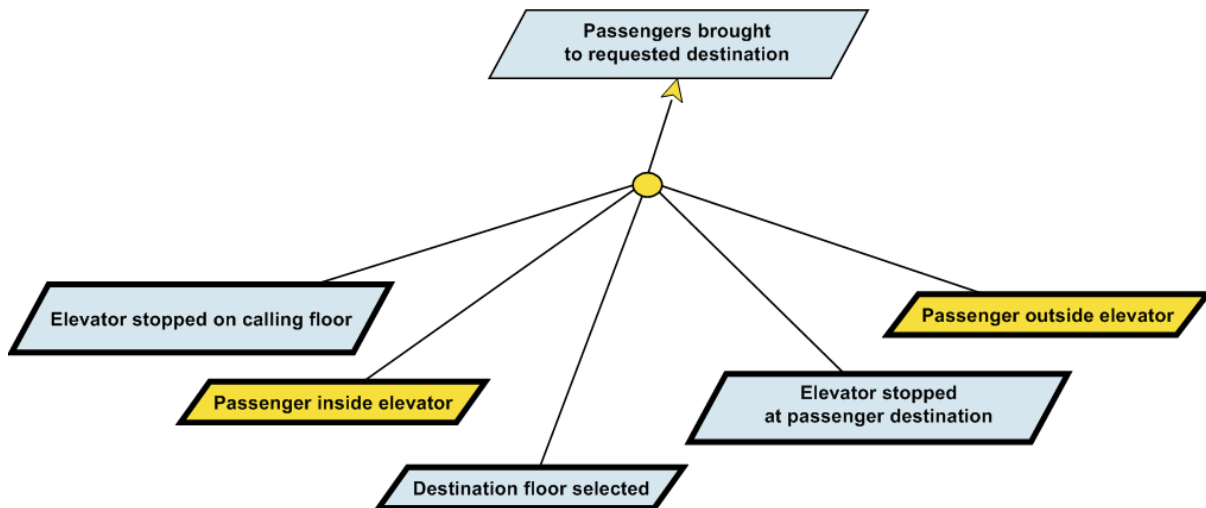


Figure 13. Passengers brought to requested destination

3.2.2.6. Safe elevator system

Let's have a look on Figure 3 and analyse the goals "No casualty" and "Secure System". The goal "Robust and reliable system" addresses more technical requirements; it will be studied later on.

"Secure System" has two subgoals: "Privacy preserved" (which does not seem relevant for the elevator problem) and "No intrusion" which we'll detail hereinafter.

3.2.2.7. No casualty

Figure 14 refines "No casualty" for a service-oriented system by applying a Milestone-driven refinement tactics, that is, we consider safety issues at the beginning of a service to passengers, safety issues during service execution, and safety issues at the end of it.

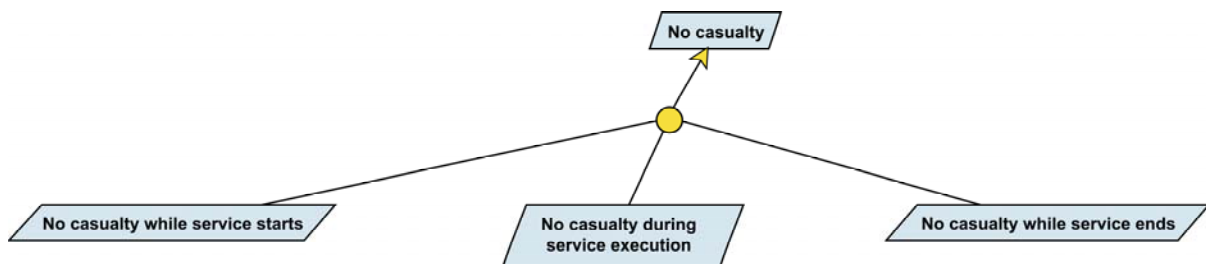


Figure 14. "No casualty" - Generic Pattern

Figure 15 shows how this pattern has been used for the elevator case. Goals have then been refined according to information collected from domain knowledge and interviews.

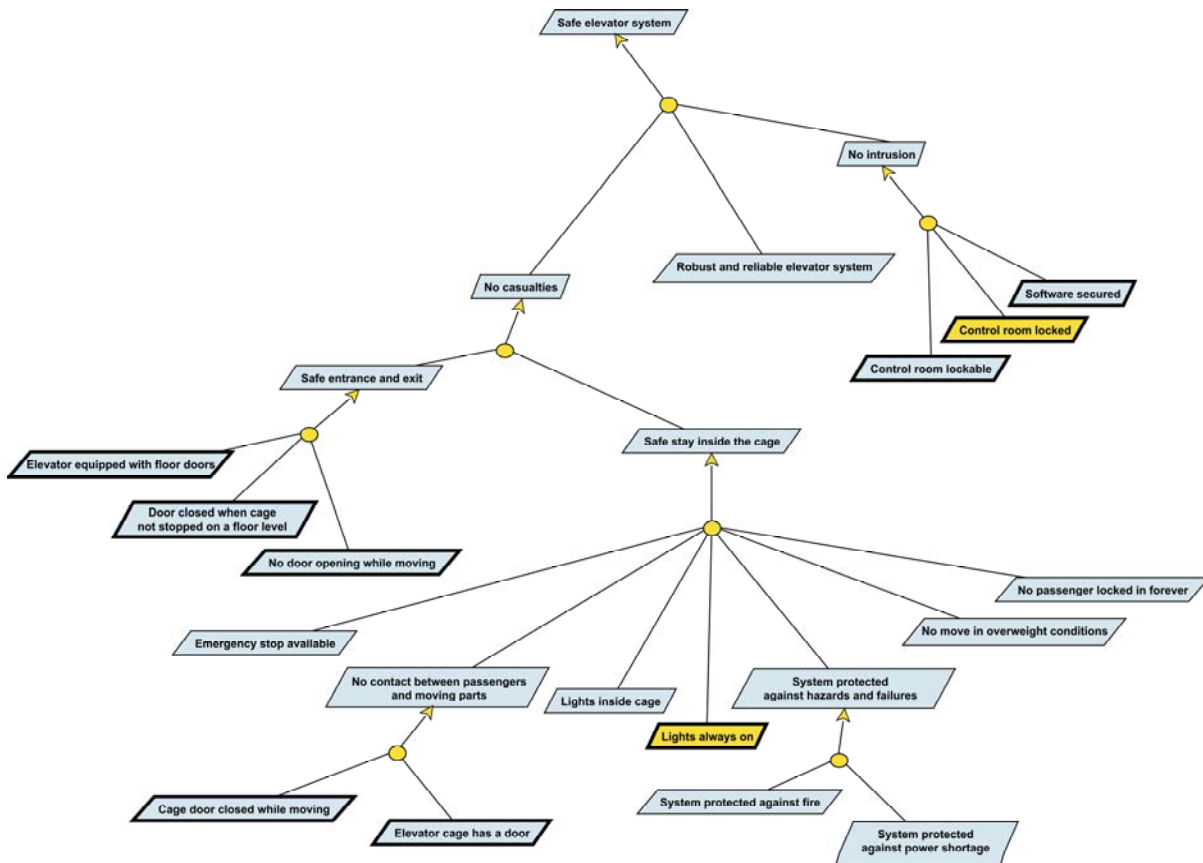


Figure 15. "Safe Elevator System"

Leaf goals are refined on the following diagrams:

- Figure 16 refines "Emergency stop available".
- Figure 17 refines "System protected against fire".
- Figure 18 refines "System protected against power failure".
- Figure 19 refines "No move in overweight conditions".

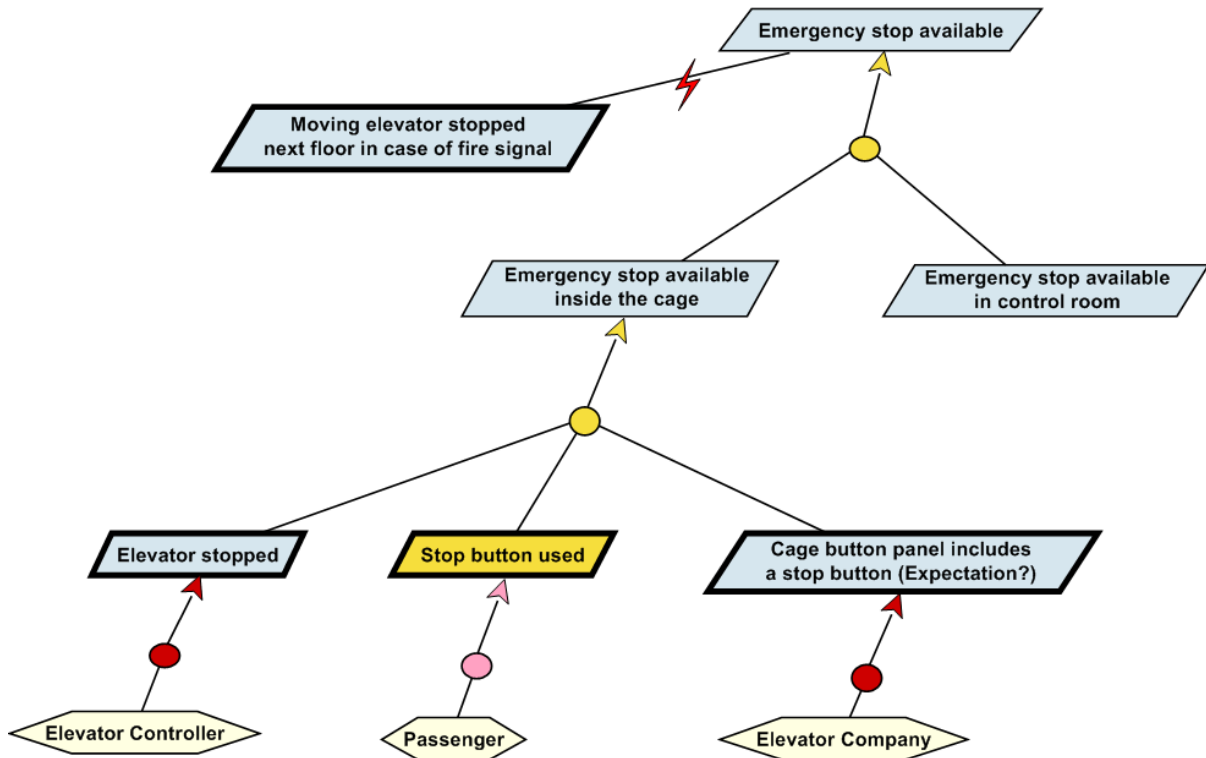


Figure 16. "Emergency Stop Available"

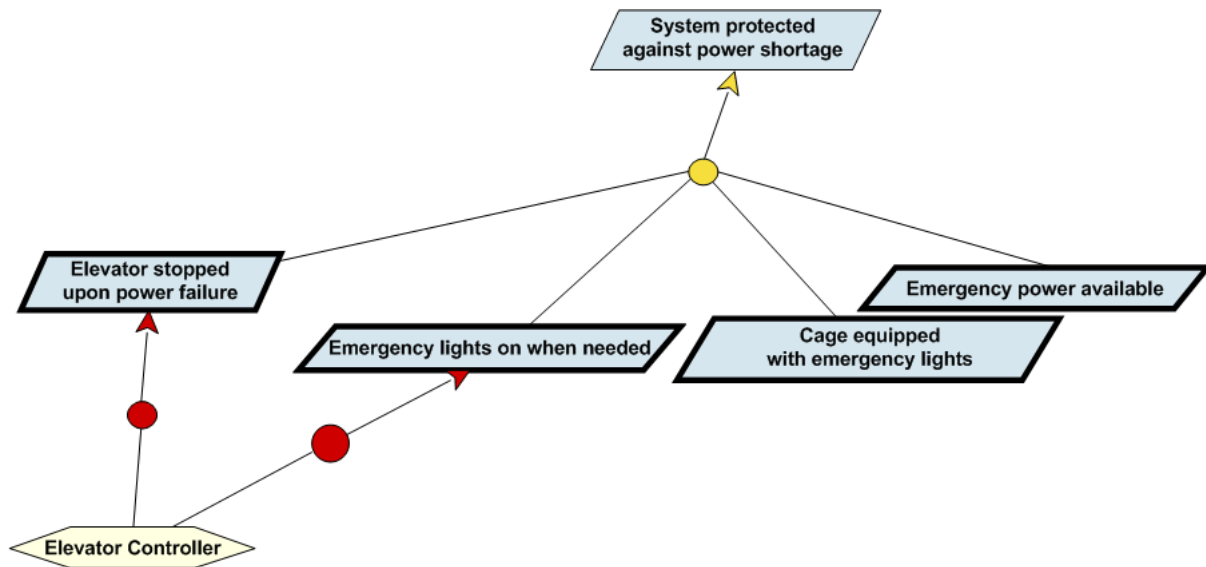


Figure 17. "System protected against power failure"

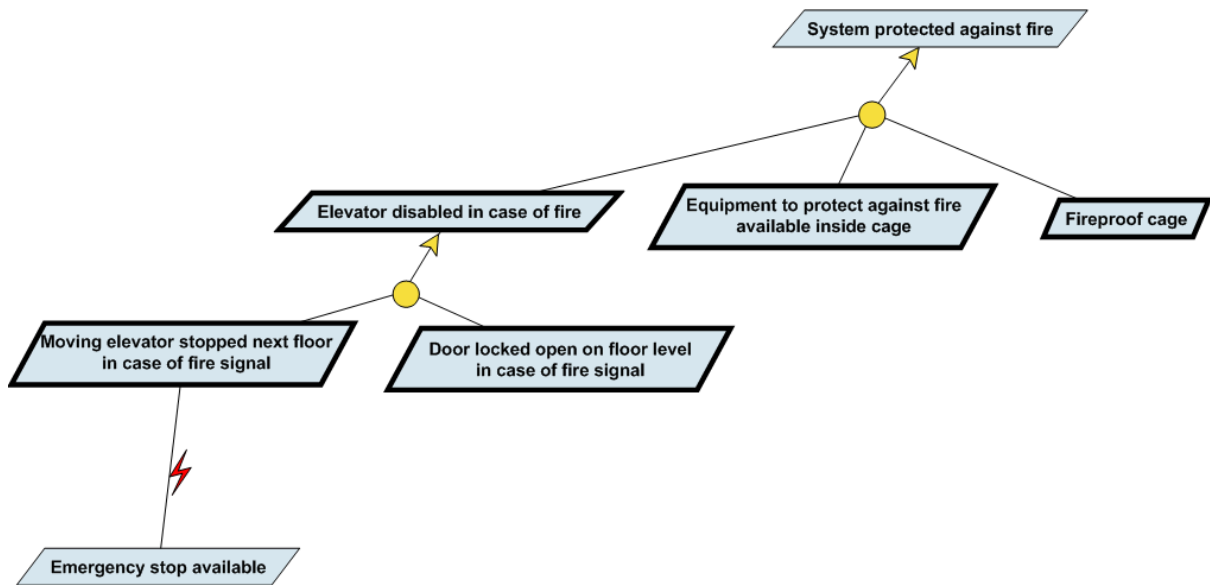


Figure 18. "System protected against fire"

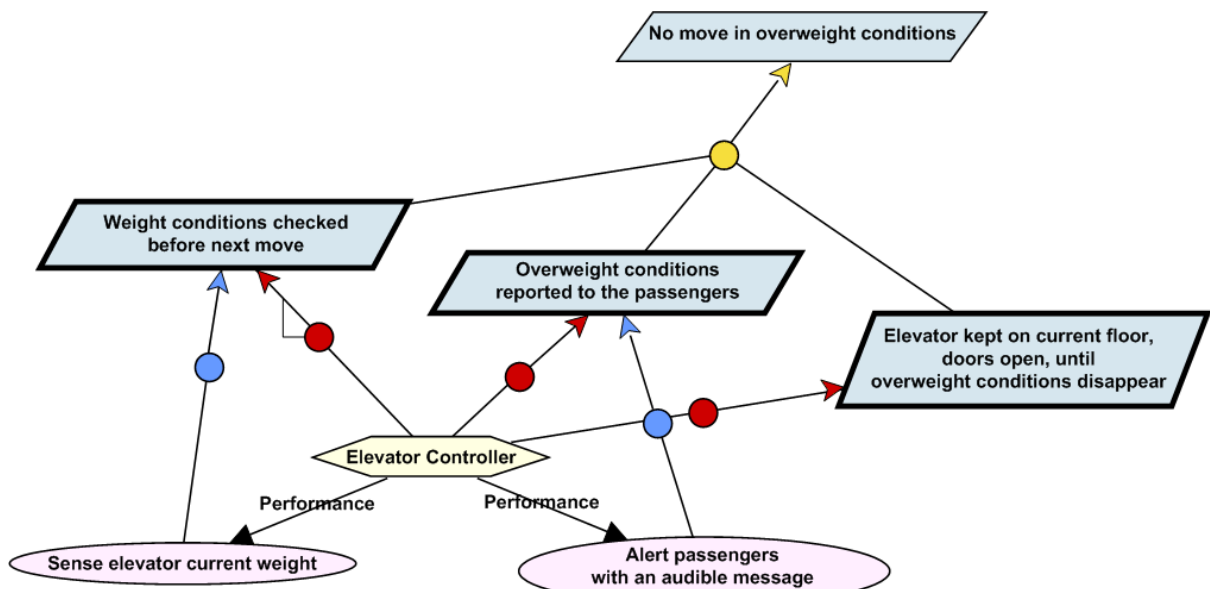


Figure 19. "No move in overweight conditions"

3.2.2.8. No intrusion

Figure 15 already shows how the No intrusion goal has been decomposed: requirements are introduced to restrain access to the room containing the hardware used for the system and also to have a software protection policy

3.3. Responsibility model

The responsibility model contains all the responsibility diagrams. A responsibility diagram describes for each agent, the requirements and expectations that he's responsible for, or that have been assigned to him.

To build a responsibility diagram, the analyst reviews the different requirements and

A KAOS Tutorial

expectations in the goal model and assigns an agent to each of them. For instance the diagram of Figure 18 (“System protected against fire”) has been updated as shown Figure 20.

- the *Elevator controller* is responsible for having the doors locked open when a fire alarm is set and for stopping all moving elevators as soon as they reach any floor
- the *Elevator company* that will install the elevator system is responsible for providing fireproof equipments (cage and equipment).

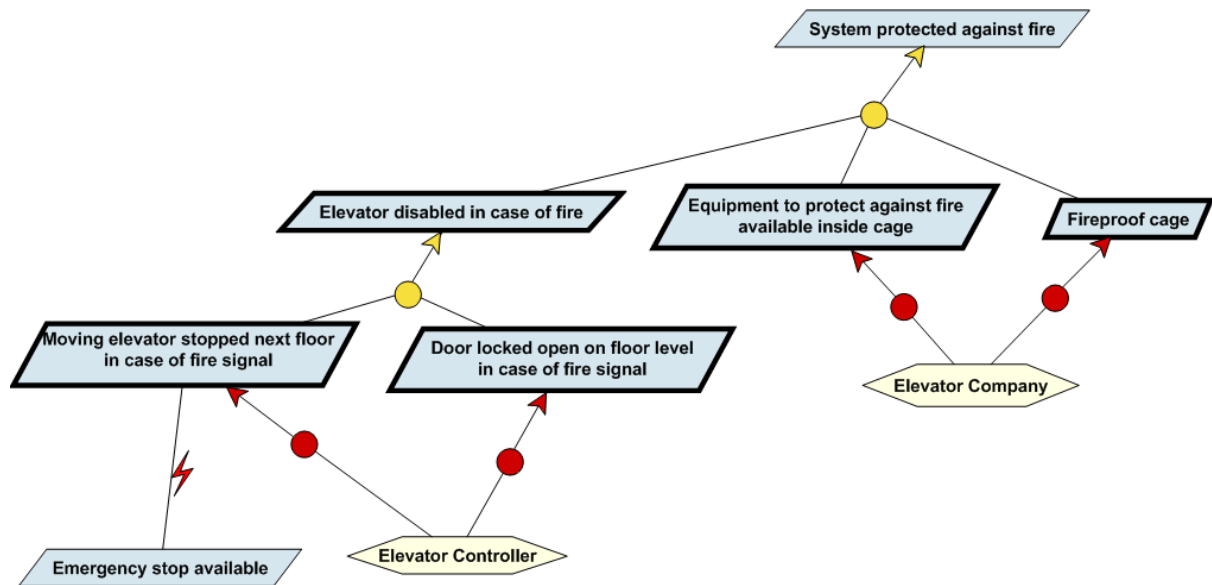


Figure 20. "System protected against fire (with responsibilities)"

After all requirements and expectations are assigned a responsible agent, a diagram is generated for each agent, listing all requirements and expectations that he's been assigned. Figure 21 and Figure 22 show such diagrams.

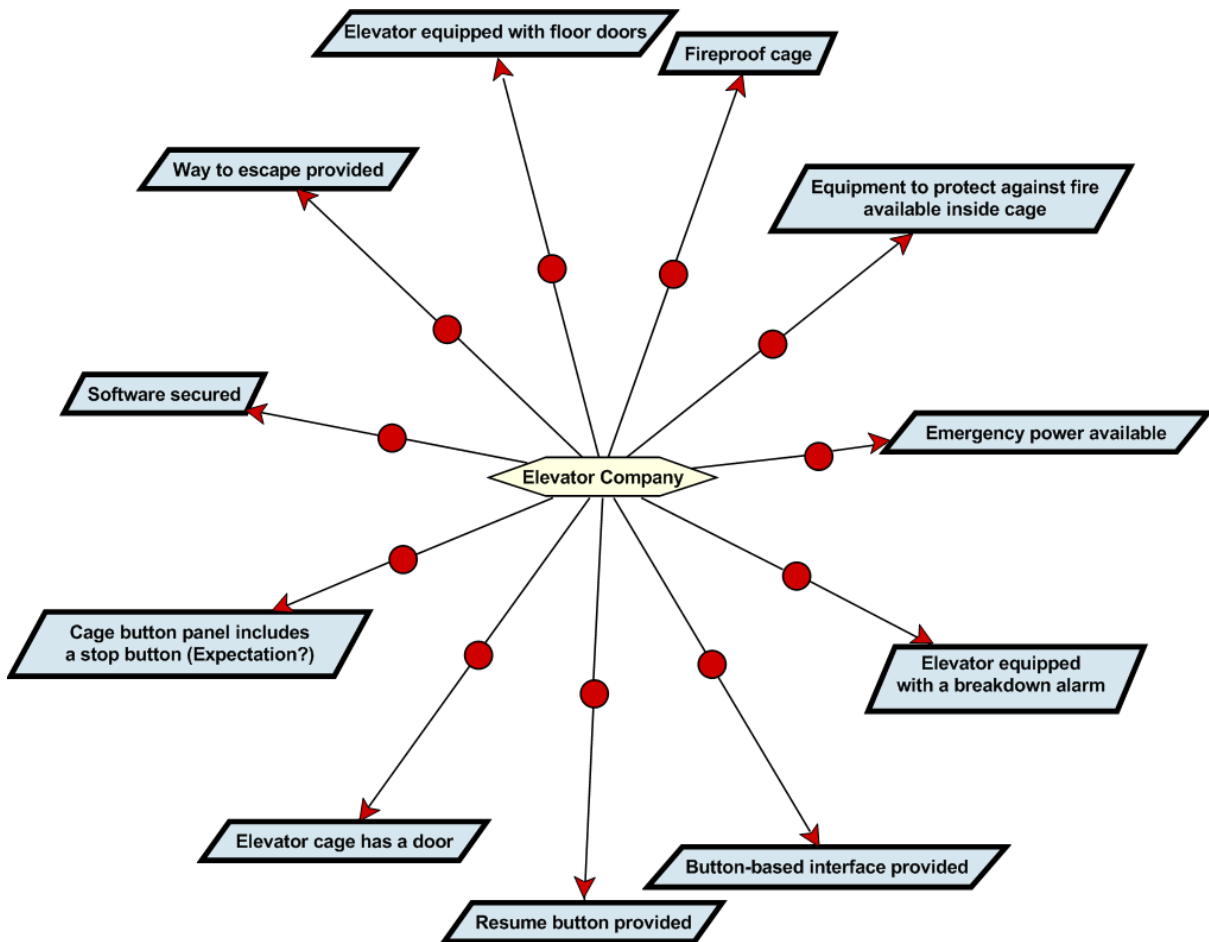


Figure 21. Responsibilities of the elevator company

By definition, the responsibility model is derived from the goal model. **Objectiver**, the tool supporting KAOS fully automates the production of all responsibility diagrams.

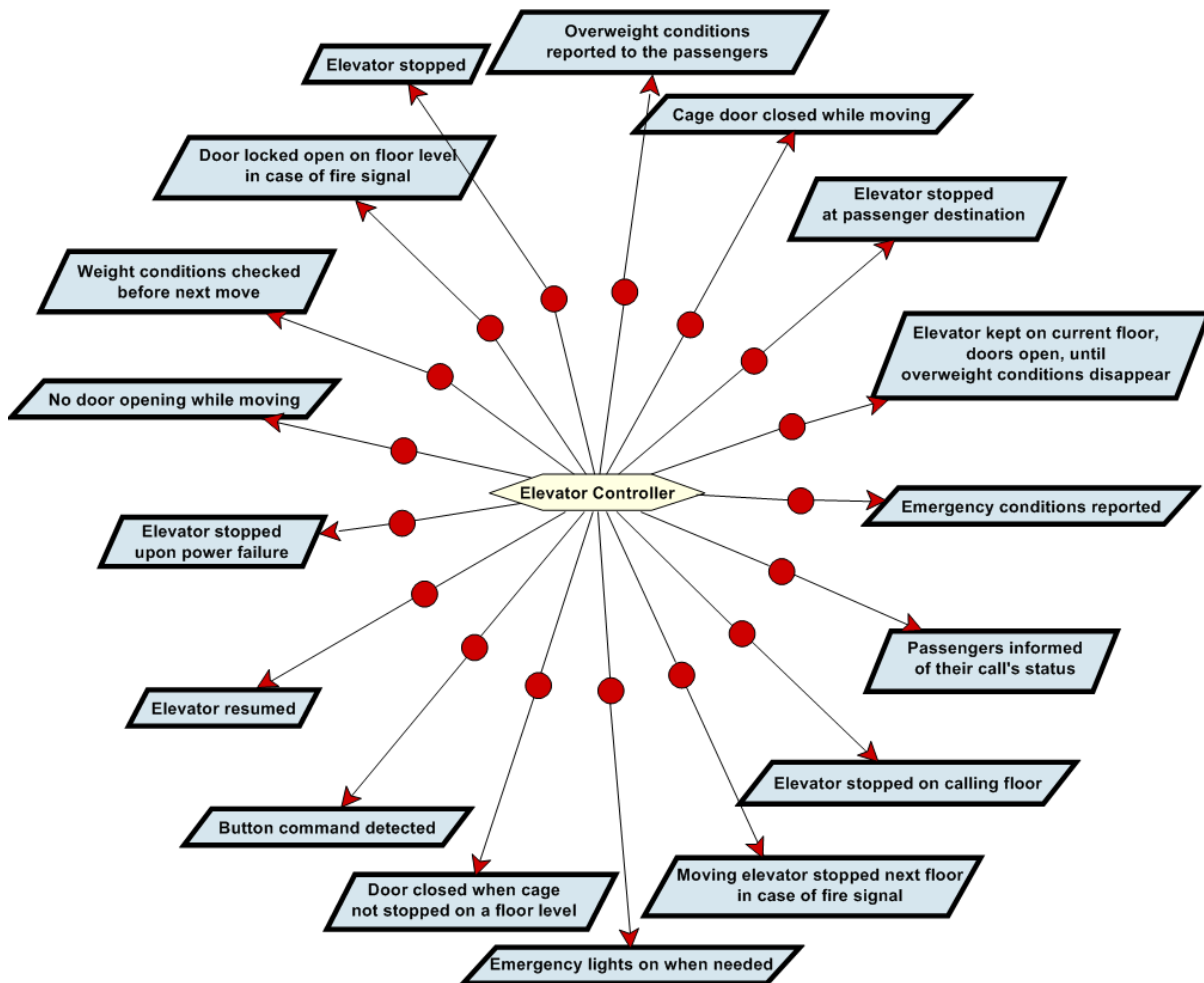


Figure 22. Responsibilities of the Elevator Controller

3.4. Object model

The object model is used to define and document the concepts of the application domain that are relevant with respect to the known requirements and to provide static constraints on the operational system that will satisfy the requirements. Part of the object model, you'll find objects pertaining to the stakeholders's domain and other objects introduced on purpose to express requirements or constraints on the operational system. Whatever type of the object, the stakeholders should understand what it means and why it was created in the model.

Three types of objects may coexist in the object model:

- **entities**: they represent **independent, passive** objects. For instance, *elevator doors*, *buttons*, etc... 'Independent' means that their descriptions needn't refer to other objects of the model. They may have attributes whose values define a set of states the entity can transition to. They are 'passive' means they can't perform operations.
- **agents**: represent **independent, active** objects. For instance, *elevator company*, *passenger*, and *elevator controller* are agents. They are active meaning they can perform operations. Operations usually imply state transitions on entities (for instance, the "CloseDoor" operation implies the following state transition on the entity "Door": status attribute changed from "Open" to "Close").
- **associations**: are **dependent, passive** objects. 'Dependent' because their descriptions

A KAOS Tutorial

refer to other objects. For instance, the *At* association links a *Cage* to a *Floor*. An instance of that association (say between Cage ‘c’ and Floor ‘f’) would hold if cage ‘c’ is currently located on floor ‘f’. They can have attributes whose values define the set of states the entity can transition to. They are passive so they can’t perform operations. But agents can make association instances change state by performing operations. For instance, the operation “*LeaveFloor*” implies the following transition: $At(c,f) \rightarrow \text{not } At(c,f)$.

Object identification is driven by the goal definition process. Most goals’s short and long definitions refer to domain objects worth being modelled and documented. All modelled objects shall have their own entries in the glossary section of the requirements document. During review of the object model, stakeholders will formally agree on a common vocabulary.

When new objects are identified while browsing through the goal model, the analyst shall define them in an object diagram and relate them to the existing concepts.

For instance, in Figure 23, we have defined “*Alarm bell*” as being a component of the “*Elevator System*”.

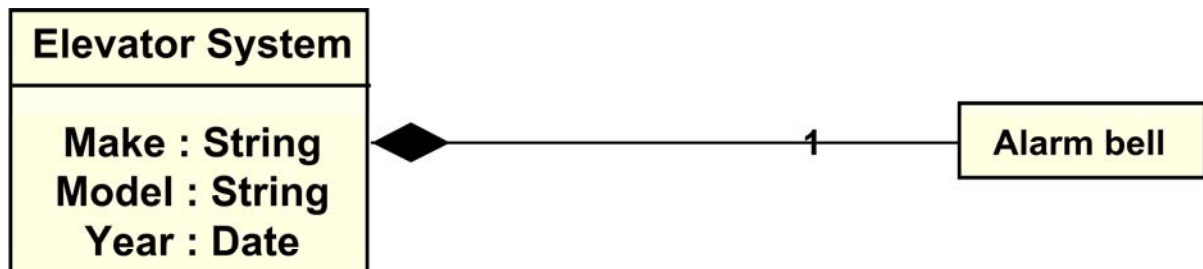


Figure 23. Alarm bell

Another way to identify new objects consists in looking at the requirements and discover the system components they are necessary for satisfying the requirements. Objects may be represented in goal diagrams; the **concerns** relationship is used to link a requirement to the objects that are needed for it to be satisfied (see an example on Figure 24). Identifying those objects shall further restrict the space of solutions that can be proposed by the future system provider.

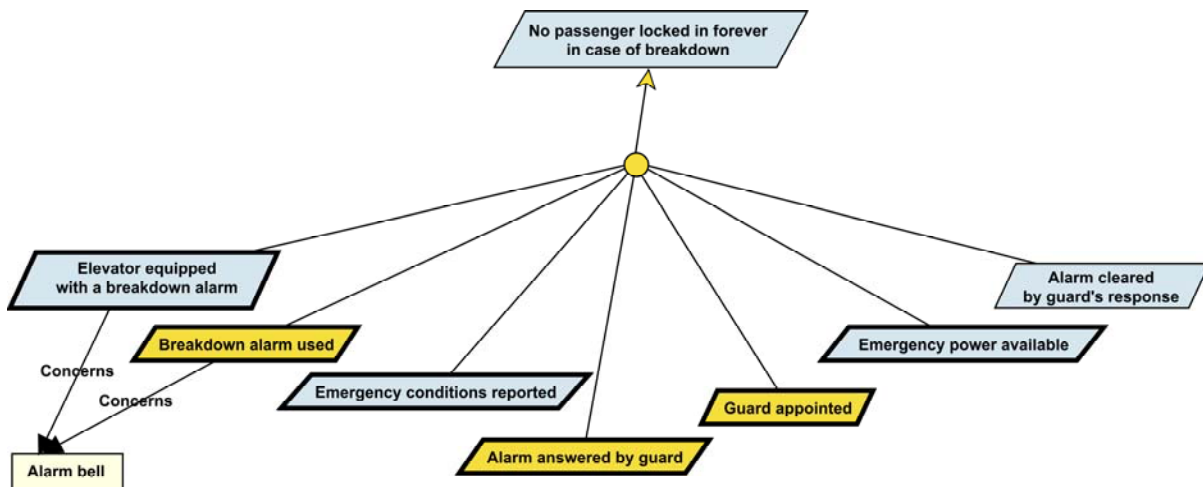


Figure 24. Concerns relationship: "Alarm bell"

A KAOS Tutorial

Figure 24 shows the components of an Elevator system. The figure should be read as follows:

The *Elevator System* is made of the following components:

- one or several *cage*
- at least 2 *floors*
- one alarm *bell*, which is (a specialization of) an *Alarm device*
- one *elevator controller*, located in a *control room*
- one *power supply*.

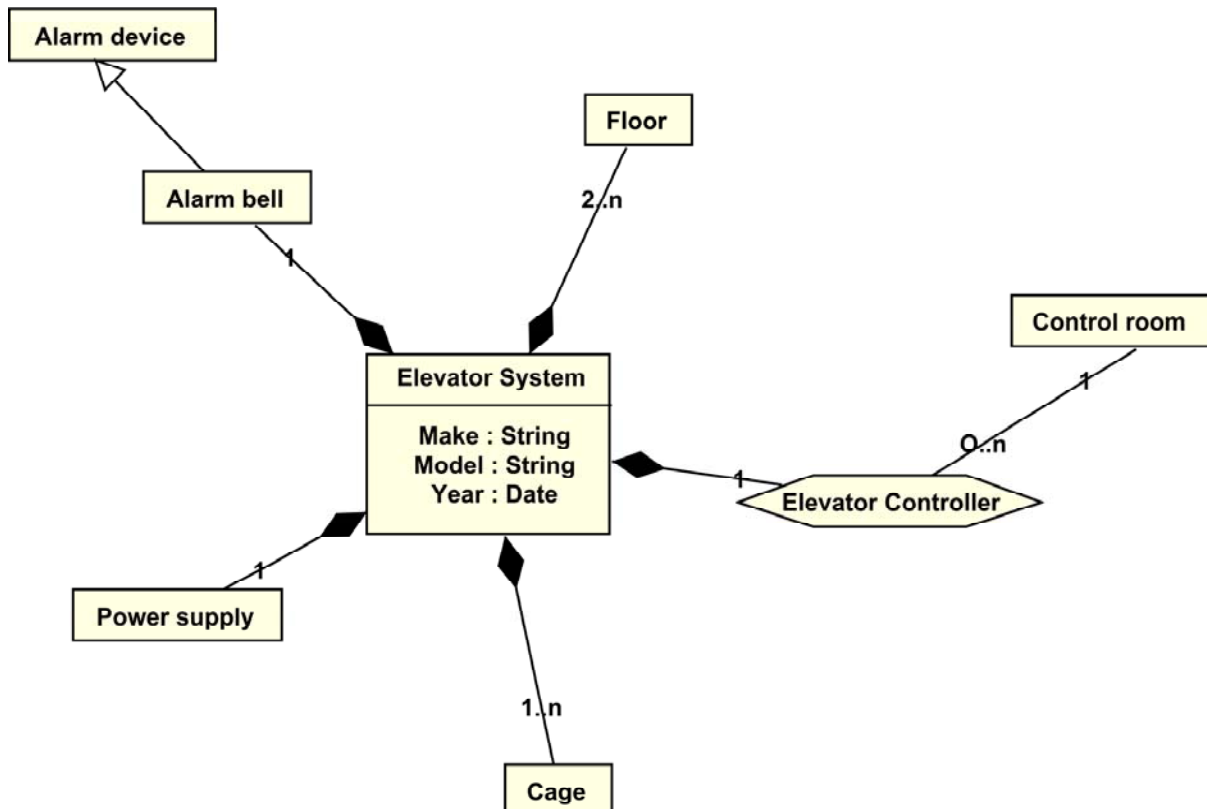


Figure 25. "Elevator System"

A cage is equipped as shown on Figure 26. Floors are all equipped as shown on Figure 27.

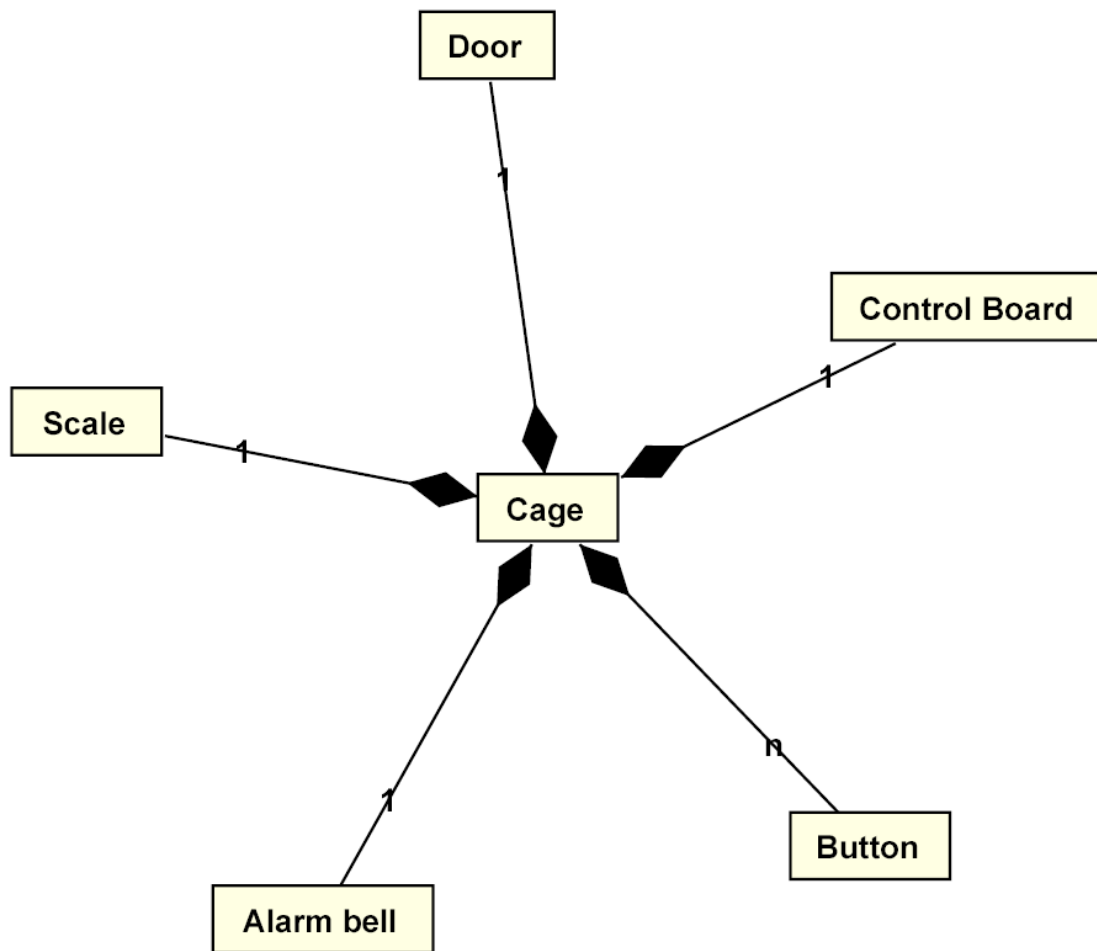


Figure 26. Object "Cage"

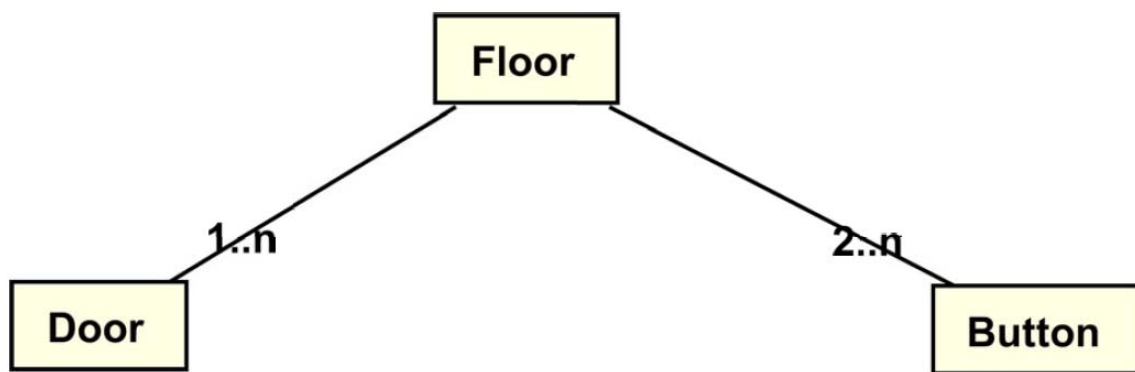


Figure 27. Object "Floor"

The KAOS object model is compliant with UML class diagrams in that KAOS entities correspond to UML classes; and KAOS associations correspond to UML binary association links or n-ary association classes. Inheritance is available to all types of objects (including associations). Objects can be qualified with attributes.

3.5. Operation model

The KAOS **operation model** describes all the behaviors that agents need to fulfill their requirements. Behaviors are expressed in terms of operations performed by agents. Operations work on objects (defined in the object model): they can create objects, trigger object state transitions and activate other operations (by sending an event).

Where do operations come from? There are two sources for identifying them :

- *Operations can directly be expressed by stakeholders during the interviews.* Stakeholders in charge of processes in the current system typically describe processes rather than goals during interviews. The analyst will then have to ask specific questions in order to identify the reasons behind the existing processes and hence unveil the goals that justify these processes.
- *Operations can be identified by looking at all the existing requirements.* They explain how requirements have to be realized.

A requirement can be operationalized by some object(s), by some agent behavior(s) or a combination of both:

- Requirements that describe static properties on the system are operationalized by objects. For instance, the requirement “*Elevator equipped with floor doors*” will be operationalized by an object: “*Floor door*”.
- Requirements that describe dynamic system properties are operationalized by operations.
- Requirements that describe properties with static and dynamic aspects are operationalized with objects and operations. For instance the expectation “*Stop button used*” will be operationalized with an operation “*Push button*” and the “*Button*” entity.

Figure 28 displays a process. **Operations** are represented as ovals. Concerned objects are connected to the operations by means of **Input** and **Output** links. **Events** are represented as those traffic signs that are used to indicate directions. Events can be external or produced by operations (they are then made an output of the operation). They may start (cause) or stop operations. For instance in the diagram, the “*Refresh*” event is produced by the “*Reschedule*” operation and it starts execution of operation “*Execute schedule*”.

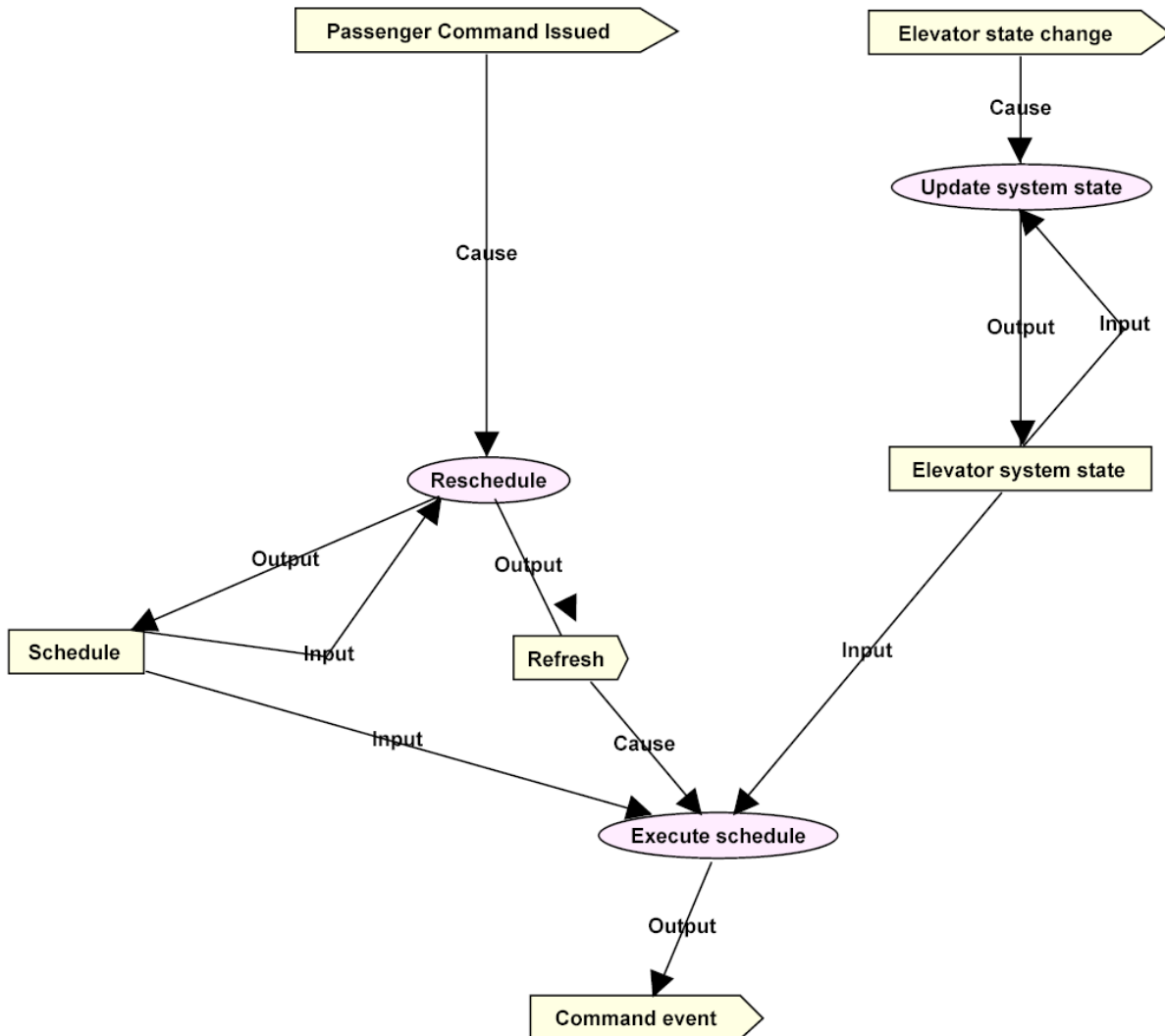


Figure 28. Process "Elevator control"

The diagram should be read as follows:

When a passenger pushes a button of the elevator system, the system refreshes the list of instructions (Reschedule) the elevator controller has to execute. The new schedule will be immediately in use by the elevator controller. To execute a schedule, the elevator controller needs to know the schedule and the latest elevator system state. The elevator system state is updated thanks to another specific event (Elevator state change).

Figure 29 shows a partial classification of the events that are used in the process. Events have been classified in three categories :

- passenger commands: translate passengers' operations like pushing a button ;
- elevator state change events: result from system observation: floor detection, sensors, etc.
- elevator system command events: issued by the elevator controller to operate the elevator system.

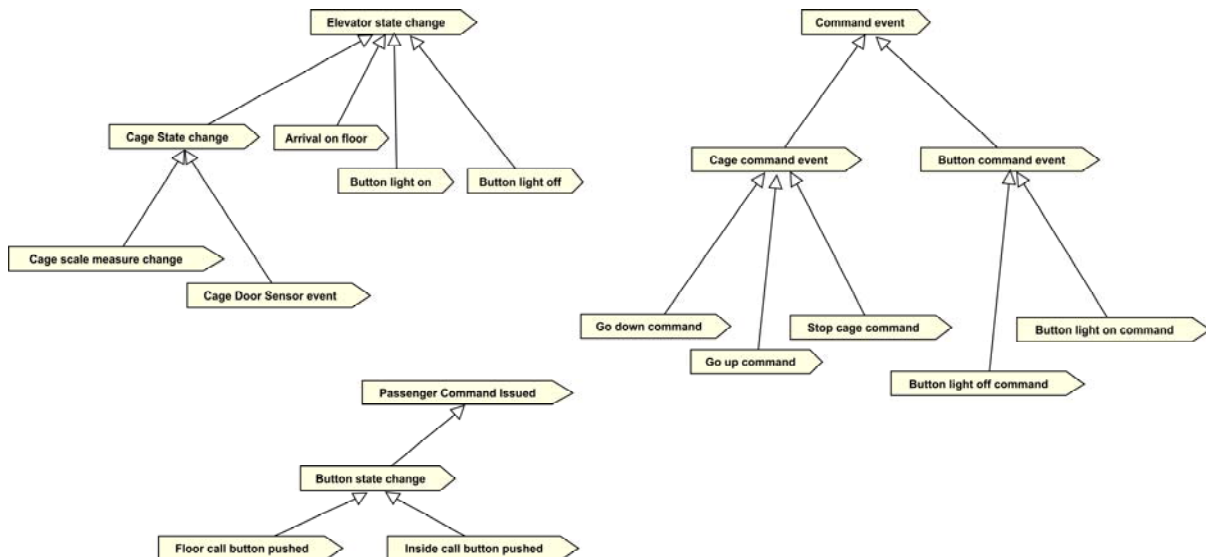


Figure 29. Object: State changes and Command events

A possible scenario for this process can be the following one : let's suppose the elevator is heading down to the ground floor and is currently on floor 35. On floor 15, a passenger calls the elevator to go down. Execution of the process can be traced as follows : the passenger command (event) “*Floor call button pushed*” is detected and triggers rescheduling by inserting two instructions in front of the schedule: one to turn the “*Floor call button light*” on (the one that was actually pushed by the passenger) ; second instruction is to have the elevator stop on floor 15. The new schedule then becomes effective and is taken into account by the elevator controller, which immediately sends an event to turn the light on (“*Button light on command*”). It will then wait to receive the event “*Arrival on floor*” for floor 15 before issuing a “*Stop cage command*” event.

Notice how a new concept was identified while we were defining this process: “*Schedule*”. This entity represents the sequence of commands the elevator controller has to execute. New concepts typically appear in processes while data and control flows are analysed.

A process diagram must fulfill the following completeness criteria :

Completeness criterion 3: *To be complete, a process diagram must specify*

- (i) *the agents who perform the operations*
- (ii) *the input and output data for each operation.*

Completeness criterion 4: *To be complete, a process diagram must specify when operations are to be executed.*

Operations can be triggered explicitly by an event (control flow) like the “*Passenger command issued*” event in Figure 28. If no event is specified, the operation will be triggered implicitly when all the data needed in input are available (data flow).

KAOS operations are used to **operationalize** (i.e. fulfill) requirements. Operations constrain the space of solutions that can be used by solution providers to design the system that shall meet the requirements. Requirements left unoperationalized are called “*open requirements*”; solution providers have the complete freedom to address the requirement as they want. The less open requirements are left in the model, the more precise the requirements document will be by eliminating possible implementations. In other words, KAOS requirements

operationalization bridges the gap between the *problem description* space (goals, requirements and domain objects) and the *solution description* space (operations representing agent behaviors and objects interacting with those operations). A good requirements document (and the underlying KAOS model) describes the problem as completely as possible, while the specification of a solution is limited only to what's really necessary.

Completeness criterion 5: *All operations are to be justified by the existence of some requirements (through the use of operationalization links).*

Figure 30 shows the operationalization of a requirement. The operationalization relationship is represented as a blue circle. It links the “Execute schedule” operation (Figure 28) to a requirements (from Figure 13). The diagram can be read as follows :

- the “Execute schedule” operation (which reads a “Schedule” in input) is executed in order to operationalize the requirement to have an elevator stopped on calling floor.
- the “Execute schedule” operation is performed by the “Elevator Controller” agent which is responsible for the requirement.

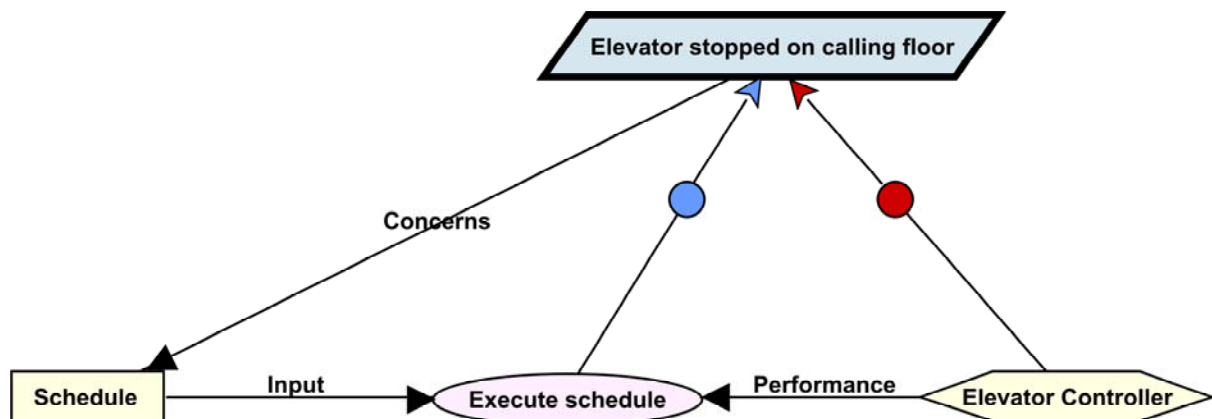


Figure 30. Operationalization: "Execute schedule"

Figure 30 represents the typical instance of a process pattern Kaos analysts have to reproduce in all projects: the pattern consists in an agent being responsible for a requirement; an operation performed by the agent; and an operationalization link between the operation and the requirement the agent is responsible for. A requirement is said to be “closed” when this triangular relationship **Responsibility-Operationalization-Performance** has been established.

Let's now turn our attention to the Reschedule operation. Why do we need that operation? When need we reorder existing schedules? Why not just add new commands at the end of the existing schedule? The following requirements identified on Figure 16 give a first answer:

- *Moving elevator stopped next floor in case of fire signal*
- *Elevator stopped (contributing to Emergency stop available inside the cage)*

Because of these cases, it's clear we need to modify the current schedule.

Another reason can be seen thanks to the scenario we described earlier, the reason being we need to minimize elevator moves and the average waiting time. These goals are not yet present in the goal model. They should refine the goal “Efficient elevator system” in Figure 5.

This particular situation gives a good example showing that the process of building a KAOS model is generally not linear: we can't proceed by first identifying all goals, then set all responsibilities, then define the domain vocabulary and finally specify the operations. In each

phase we can come with new enhancements to any of the 4 models.

The KAOS relationships (refinement, operationalization, responsibility, etc) provide **traceability** between all the pieces of the model. It also gives the analyst flexibility to explore the model (from operations to goals or reversely, for instance). Figure 31 outlines a first shot at what an efficient elevator system should be. The diagram has been elaborated based on the two goals given above.

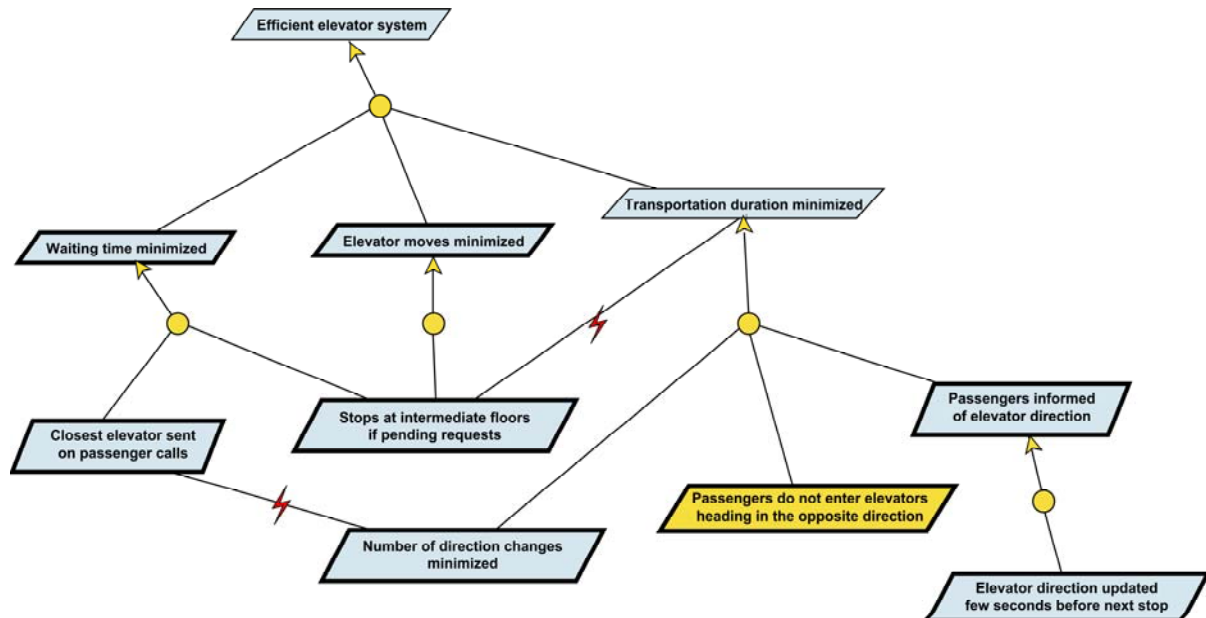


Figure 31. Goal: Efficient elevator system

Figure 32 displays the operationalization diagram for the *Reschedule* operation.

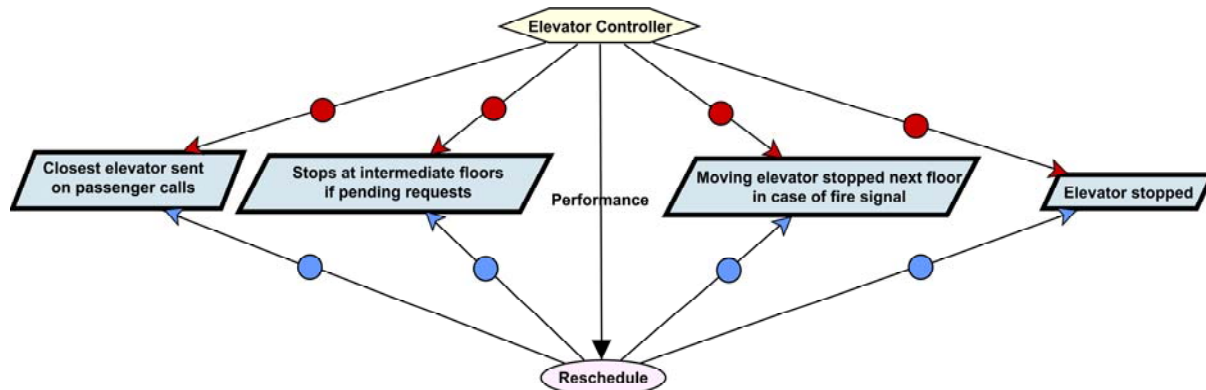


Figure 32. Operationalization: "Reschedule"

3.6. Dealing with obstacles

Obstacles are situations in which a goal, a requirement or an expectation is violated. In such cases, the obstacle is said to “obstruct” the goal, requirement or expectation.

Dealing with obstacles is very important for safety-critical systems: it allows analysts to identify and address exceptional circumstances at requirements engineering time (instead of at programming or running time) in order to produce for instance robust requirements or new requirements to avoid or reduce impacts of obstacles. The result will be a more reliable software.

A KAOS Tutorial

Obstacle analysis with KAOS is a goal-oriented activity. It begins with exploring the goal model and with negating each goal in turn. Let's remind what a refinement means: a goal G is refined into goals G_1, \dots, G_n if and only if G_1, \dots, G_n all together imply G , that is, they are a sufficient condition to achieve G . Therefore we know that if goal G is violated, it is because at least one of its subgoals is violated (if not, we would reach G).

Consider for instance the goal refinement displayed in Figure 3. This rationale allows us to consider that an elevator system will not be safe if at least one of the following conditions is met:

- casualty occur during system use
- some system component breaks down
- somebody succeeds in hacking into the system.

The same type of reasoning can be repeated by traversing the goal hierarchy. For instance, *Casualty occurring during system uses* can be refined into 3 possible obstacles according to Figure 14:

- casualty occur at service start
- casualty occur during service
- casualty occur at service end.

Each leaf obstacle is then reviewed with domain experts to study whether it is worth considering it in the obstacle analysis. It allows the analyst to prune the obstacle space and to focus on the most relevant obstacles.

Next, conditions for the obstacle to be raised are looked for. Typical ways used to identify obstacles start by considering component failures or people behaving in an unexpected way, maliciously or because of some people's capability limitation (disable people, children, ...).

Let's consider for instance the goal "*Passengers informed about elevator direction*" from Figure 31 and its subgoal "*Elevator direction updated few seconds before next stop*". An obstacle to the latter goal occurs when blind people want to use the elevator system as they cannot read the indications. Figure 33 shows how obstacles are represented in KAOS diagrams. Obstacles are linked to the goals they obstruct (**obstruction** link). Obstacles can be refined the same way we do with goals, but while goals are generally 'AND-refined', obstacles are most often 'OR-refined'.

Having identified the conditions for it to occur, the analyst can fix the obstacle in several ways.

A first way consists in defining new requirements that prevent the obstacle from occurring. In the above example, one can add a new requirement: "*Elevator direction announced by voice upon floor arrival*". Figure 34 shows the update to the goal model and the new requirements linked to the obstacle they resolve (**Resolution** link). If we carry on this way, other requirements can be identified for blind people. For instance: "*Floor level announced by voice upon floor arrival*" and "*Call buttons in Braille*".

An important source of obstacles concerns system (and components) reliability and robustness. Reliability and robustness requirements aim at preventing failures to occur. One can for instance introduce requirements in terms of Mean Time Before Failure (MTBF), or in terms of a maintenance program to replace components systematically at the right time, and so on.

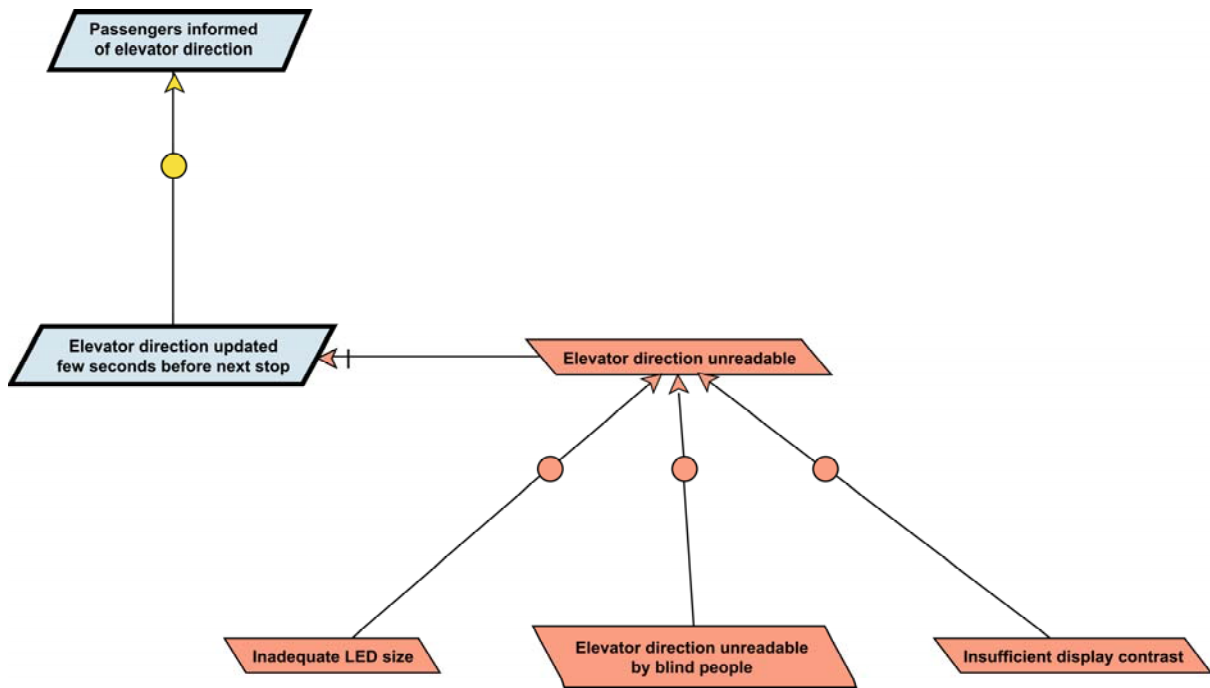


Figure 33. Obstacle: "Passengers informed of elevator direction"

A second way to fix an obstacle is to restore the obstructed goal once the obstacle occurs. For instance, if an elevator failure puts the system out of order, a restoration requirement can prescribe the maximum delay within which the system must be repaired.

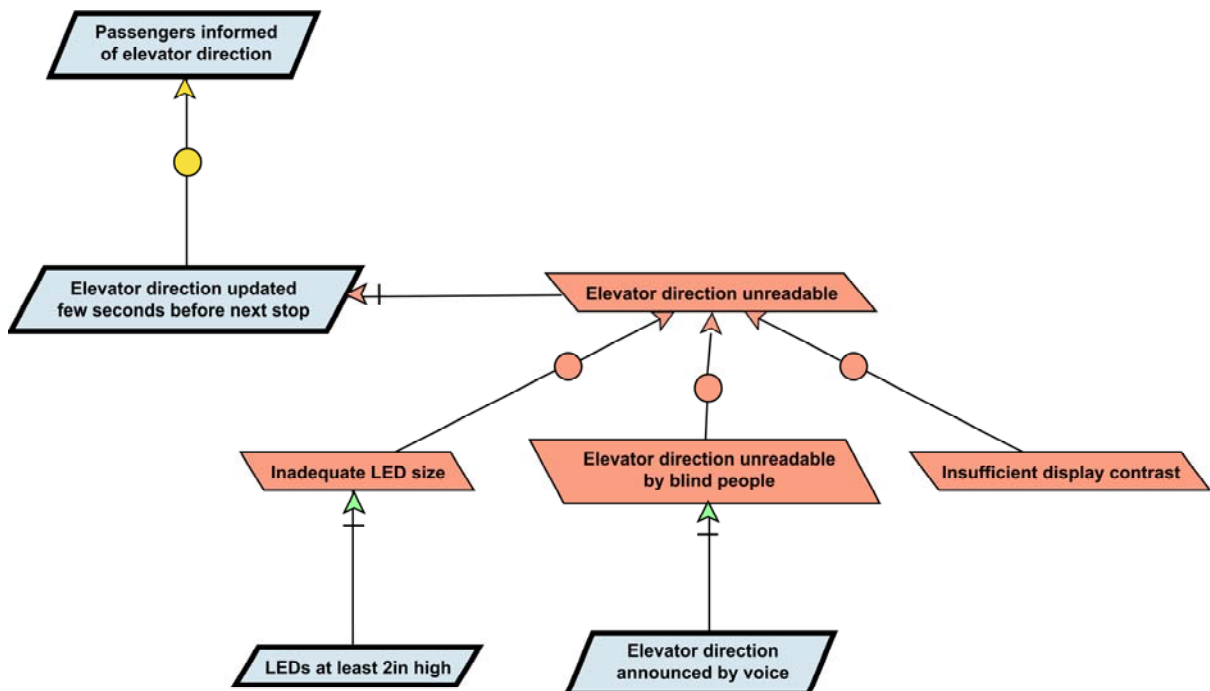


Figure 34. Obstacle resolution: "Passengers informed of elevator direction"

A third way to fix an obstacle consists in minimizing its effects. Suppose the building is equipped with two elevator cages, one serving floors 1 to 10, the other one floors 10 to 20. If one cage goes out of order, a requirement should prescribe that the system shall be self-reconfigured so that the other cage provides service to all floors (from 1 to 20). The service

would therefore not suffer from interruption.

The previous ways add new requirements. It is also possible to fix obstacles by modifying the KAOS model in different ways, for instance by substituting an agent with a more “capable” one, by weakening the obstructed goal so that the obstacle is no longer relevant or by substituting a goal to another one.

Figure 35 shows a possible refinement of the goal “*Robust and reliable elevator system*” taking the obstacle resolution discussion above into account. Let’s observe the alternative refinement of the *Worn-out components replaced in time*. It means that each of the two branches can be opted for separately to satisfy the goal. But KAOS does not force exclusion. So both alternatives can be used.

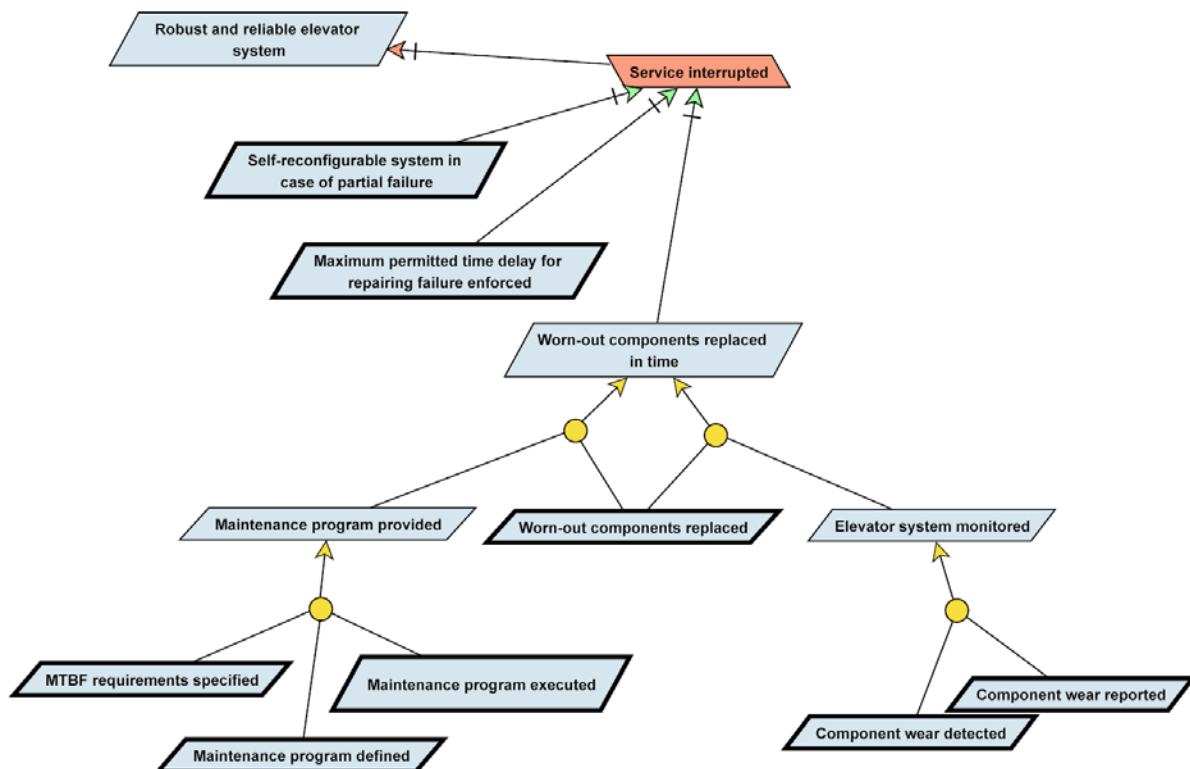


Figure 35. Obstacle resolution: Robust and reliable elevator system

3.7. Requirements document generation

The KAOS methodology focuses on requirements modelling and, as such, does not include any constraints for the requirements documents that could be derived from a KAOS model. However the reader having reached this step in this tutorial, should legitimately raise the question: “*Ok fine, how can I now use the information in the KAOS model and generate my requirements document?*”

This section describes a specific type of requirements documents that can be derived from KAOS models.

3.7.1. Requirements document structure

When working on industry-level projects, we, at Cediti, are usually generating requirements documents whose structure is borrowed from the *IEEE 830-1998* standard for software requirements specification. Here’s the IEEE 830 structure :

Table of contents

1. Introduction

1.1 Purpose

1.2 Scope

1.3 Definitions, acronyms, and abbreviations

1.4 References

1.5 Overview

2. Overall description

2.1 Product perspective

2.2 Product functions

2.3 User characteristics

2.4 Constraints

2.5 Assumptions and dependencies

2.6 Apportioning of requirements

3. Specific requirements

The following paragraphs briefly explain the meaning of each section in the IEEE template.

- **Purpose.** This section aims at delineating the purpose of the requirements document and at specifying the intended audience.
- **Scope.** This section explains what the software will, and, if necessary, will not do, describes the application of the software being specified, including relevant benefits, objectives, and goals.
- **Definitions, acronyms, and abbreviations.** This section provides the definitions of terms, acronyms, and abbreviations required to properly interpret the requirements document.
- **References.** This section provides a list of all documents referenced elsewhere in the requirements documents including title, date, publishing organization, ...
- **Overview.** This section describes what the rest of the requirements document contains and explains how the requirements document is organized.
- **Product perspective.** This section relates the software to be developed with its environment: description of interacting external components, identification of interfaces, and global architecture.
- **Product functions.** This section provides a summary of the major functions that the software will perform.
- **User characteristics.** This section describes those general characteristics of the intended users of the product including educational level, experience, and technical expertise.
- **Constraints.** This section aims at providing a general description of other items that will limit the developer's options (e.g., hardware limitations, development platform, etc.)
- **Assumptions and dependencies.** This section lists the factors (other than design constraints listed above) that affect the requirements stated in the requirements document.

- **Apportioning of requirements.** This section identifies requirements that may be delayed until future versions of the system.
- **Specific requirements.** This section contains all the software requirements to a level of detail sufficient to enable designers to design a system to satisfy those requirements, and testers to test that the system satisfies those requirements.

3.7.2. How to fill in the template from a KAOS model

The structure of the requirements document we use to produce is the following one

Table of contents

1. Introduction
 - 1.1 Document purpose
 - 1.2 System purpose
 - 1.3 Definitions, acronyms, and abbreviations
 - 1.4 References
 - 1.5 Overview
2. Overall description
 - 2.1 System perspective
 - 2.2 User requirements
 - 2.3 User characteristics
 - 2.4 Constraints
 - 2.5 Assumptions and dependencies
 - 2.6 Apportioning of requirements
3. System requirements

Here follows a description of each section and how it relates to the KAOS model (if any) and to the IEEE standard.

- **Document purpose.** Same as the Purpose section in the IEEE standard..
- **System purpose.** Same as the Scope section in the IEEE standard.
- **Definitions, acronyms, and abbreviations.** This section is derived from the KAOS object model. All concepts are extracted from the Object model by providing the concept name and the concept associated definition (if any). The list is sorted by concept names, alphabetically. If the diagrams, the object model consists of, have been organized in a tree structure, this structure can be used to create an isomorphic glossary structure.
- **References.** Same as in the IEEE standard.
- **Overview.** Same as in the IEEE standard.
- **System perspective.** Same as in the Product perspective in the IEEE standard.
- **User requirements.** This section is intended to replace the Product functions section of the IEEE standard. It aims at describing the problem set in the requirements document. It explores the KAOS goal model from top to bottom by adopting a pre-order traversal of the goal model. Each goal diagram corresponds to a subsection. Each subsection displays the diagram and a text explaining it. Cross-references are

inserted to refer to other diagrams or texts. Requirements are clearly distinguished in the text; each requirement is presented by giving its name, definition and unique identifying number. Requirements presented in this section are functional and non functional.

- **User characteristics.** Same as in the IEEE standard.
- **Constraints.** Same as in the IEEE standard.
- **Assumptions and dependencies.** This section lists the assumptions used in the KAOS goal model that have been labelled to appear in this section. It also lists the obstacles that the system is not expected to deal with.
- **Apportioning of requirements.** This section provides a table containing the list of all the requirements presented in the User requirements section sorted by priority level.
- **System requirements.** This section replaces the specific requirements section of the IEEE standard. It aims at describing the solution that is expected to solve the problem presented in the User requirements section. It contains the following sections :
 - **System architecture.** This subsection outlines a decomposition of the system by KAOS agents. For each agent, the list of the requirements he/she/it is responsible for is listed. The lists come from the KAOS responsibility model.
 - **Object model.** This subsection presents each diagram of the KAOS object model, explains each diagram structure, lists all concepts appearing in the diagram with their definition (the same as in the glossary). Concepts enumerated can be domain concepts or concepts introduced on purpose to operationalize requirements. In the latter case, the list of requirements, each concept contributes to operationalize, is mentioned in a table at the end of the section.
 - **Operation model.** This subsection presents each diagram of the KAOS operation model. It explains each diagram content. At the end of the section, a table shows the operations performed by each agent and the requirements these operations contribute to satisfy. Those requirements come from the KAOS operationalization relationships.

4. Discussion

4.1. Is it worth paying attention to a Kaos analysis?

A lot of requirements documents produced nowadays just describe solutions: the expected functions, processes and data structures. The example studied in Section 3 is far from being complete. However it should become clear to the reader that a requirements analysis with KAOS is much more than a limited description of the solution. An important focus is put on the problem itself. So the question that should now be raised: is it worth paying so much attention to the problem analysis first?

To answer the question, let's compare the kind of information provided by the *solution description* (see Figure 23 to Figure 29) with the one provided by the *problem description* (Figure 5, 7, 12, 13, 15-19, 31, 34). Are the information collected in the latter diagrams irrelevant for the requirements document ? Surely not ! They introduce abstract and fundamental properties that have to be fulfilled by the system to be. Imagine now a requirements document consisting only of description derived from Figure 23-29. One can reasonably expect that a development team will develop the *system right* with respect to that specification. But how may we guarantee that the system built is the *right system* if we discard the first part of the analysis which describes precisely what the users really need?

4.2. Traceability

A major benefit of KAOS resides in the fact that it provides continuum between the problem description and the expected solution description. This bi-directional traceability between problem and solution spaces is fundamental not only for the requirements analyst to be sure, the system to build will be the right one, but also for developers who need to understand the context and objectives to make correct architectural and design choices.

Moreover systems developed nowadays work in a quickly changing environment that requires lots of modifications. Limiting the available high-level information about the system to a high-level description of the required solution increases the risk of issuing, later on, change requirements that are inconsistent with the original vision. It is even more risky if, as it is often the case, those who maintain the system will not be those who developed the initial release.

Most companies consider nowadays that a requirements document is a one-shot document written to prepare calls for tenders or to be initially understood by internal development teams. As the system will require 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. If the approach followed to produce the requirements document relies only on the use of a word processor, modifying the requirements document quickly turns out to be a nightmare. As with KAOS, the requirements document is derived from a KAOS model, it becomes possible to modify the KAOS model and regenerate a consistent requirements document from it.

4.3. Completeness

Requirements documents elaborated with KAOS tend to be more complete. Completeness must be understood by looking at the five completeness criteria presented in this tutorial: a

complete KAOS model leaves no space for wishful thinking (a goal not refined), no space for requirements 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.

Completeness of a KAOS model clearly relies also on the completeness of the goal model. So how can we be sure that all the relevant goals have been collected and modelled? Completeness of the goal model relies on three points:

- it depends on the quality of the requirements definition phase. Interviewing stakeholders is a fundamental technique, which, if operated properly, leads to a good goal covering rate.
- it depends on the validation meetings during which stakeholders review consolidated goal diagrams. Conflicts are addressed and goals that have been forgotten during interviews receive a new chance to come up.
- it depends on refinement techniques that guarantee refinement completeness and consistency. Those techniques, based on generic refinement patterns, require formal notations to be used. However an informal use of those patterns as we have done it in this tutorial, is sufficient to deal with most types of non safety-critical applications.

4.4. No ambiguity

On the one hand, the completeness criteria evoked in Section 4.3 contributes to less ambiguity in requirements documents; we know who is responsible for what and who perform what.

On the other hand, the object model contains all the information needed to produce the requirements document glossary. The glossary validation forces all stakeholders who generally have different background, to agree on the domain and application relevant concepts. Standards for requirements document require the inclusion of a glossary. With KAOS, we can build the glossary progressively and we get for free a criterion for deciding which concept has to be defined in the glossary: in fact all those defined in the object model.

4.5. For which project size is it worth thinking of using Kaos ?

A frequently asked question is about the minimal project size for which a KAOS approach is relevant. Clearly, if a development project is estimated to take 20 man days, the probability of a positive return on investment is quite weak as building a requirements model is time consuming.

Most of our consulting cases concern projects for which a typical requirements analysis of 4 to 8 man months has been needed. The typical duration of the requirements analysis phase is 3 months and the budget needed for it represents about 10% of the total project cost.

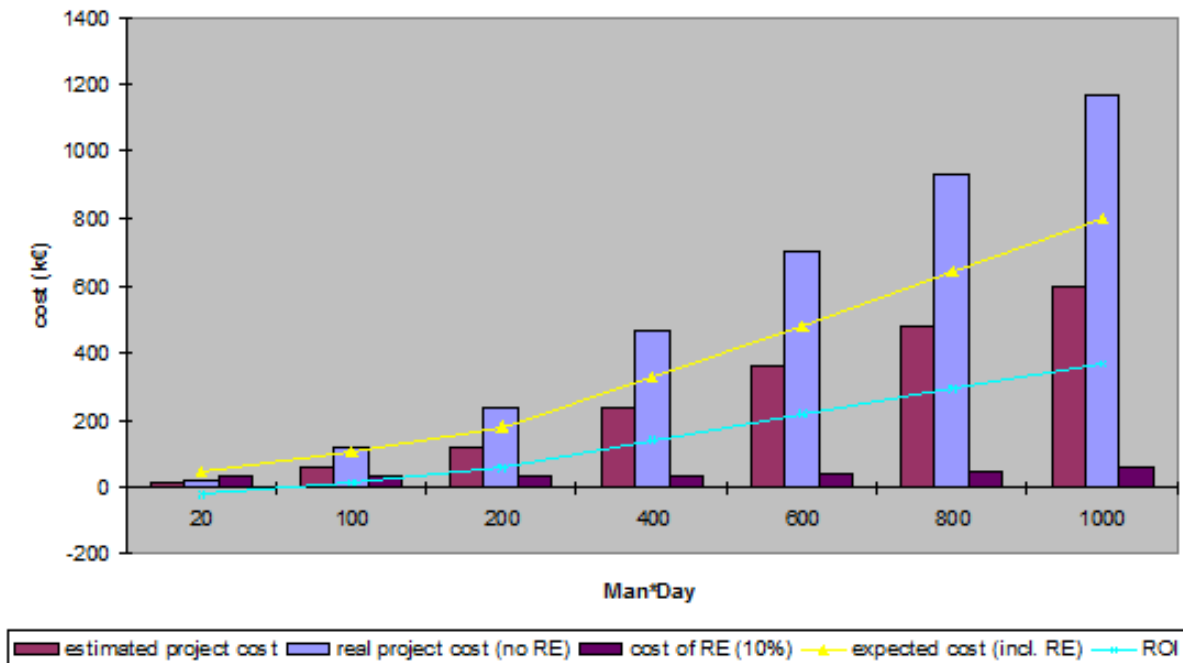
Figure 36 extrapolates the return on investment according to project size from our experience and from the following hypotheses :

- the cost of one developer is 0,6 k€ per day
- the cost of one analyst is 1 k€ per day
- about one development project over 2 experiments an cost overrun of about 189% (Standish group)
- about one project over 2 that fails, fails because of a requirements related problem (Standish group)

A KAOS Tutorial

- the cost of an ideal requirements analysis phase is estimated at 10% of the project cost with a minimum bound fixed to 30 k€.

The figure¹ shows that a KAOS-like requirements study is worth the effort as soon as the project man power is more than 100 man days. For medium-size and larger projects the cost reduction expected is about 30%.



Do we have to conclude that for projects less than 100 man days, a KAOS analysis is not relevant? In fact, the above estimation supposes that the requirements analysis is started from scratch with analysts having no specific domain background and doing no reuse of previous analysis schemes. Clearly if the company business consists in customizing a framework for its customers, it shall be interesting to develop a generic KAOS model once and specialize it many times during the gap analyses made to compare the user requirements with what the package provides.

¹ The *estimated project cost* line takes only developers' cost into account. The *real project cost (without RE)* line takes the overrun factor into account for one project over 2. The cost of a KAOS-like requirements study (*cost of RE*) is estimated to be 10% of the estimated project cost (which also corresponds to what we can reasonably expect customers agree to pay). The *expected cost (incl. RE)* represents the project cost including a KAOS-like requirements engineering phase and supposing that the requirements study will allow the project to avoid 75% of the overrun cost (we take here into account that one project over 2 that fails because of a requirements related problem and that those problems are the most costly to address as they are often discovered very late in the life cycle). The *ROI* line measures the difference between the real project cost with no KAOS-like requirements study and the one expected if a KAOS-like requirements study is made.

5. Conclusion

This tutorial has presented the ten key ideas underlying KAOS:

1. First build a requirements model
2. Justify your requirements by linking them to higher-level goals
3. Build a model of the whole system, not just the software part of it
4. Build a responsibility model
5. Build a consistent and complete glossary of all the problem-related terms you use to write the requirements
6. Describe how the agents need to behave in order to satisfy the requirements, they are responsible for.
7. Base the requirements document on the requirements model
8. Validate your requirements by first reviewing the model
9. Use a defensive approach to the building of a requirements model
10. Consider your requirements document as a reference that shall need updating during the project development life cycle
11. These ideas have been put to practice through the analysis of a case study.

This example and the discussion Section 4 explain all the benefits of using an approach like KAOS. Besides the production of undoubtedly better quality requirements documents, KAOS also has shown to be of a real help to analysts for thinking about the systems to come and a good tool to permit communication between the different project stakeholders during the whole system life cycle.

Nothing but that shows that the game is worth the candle!

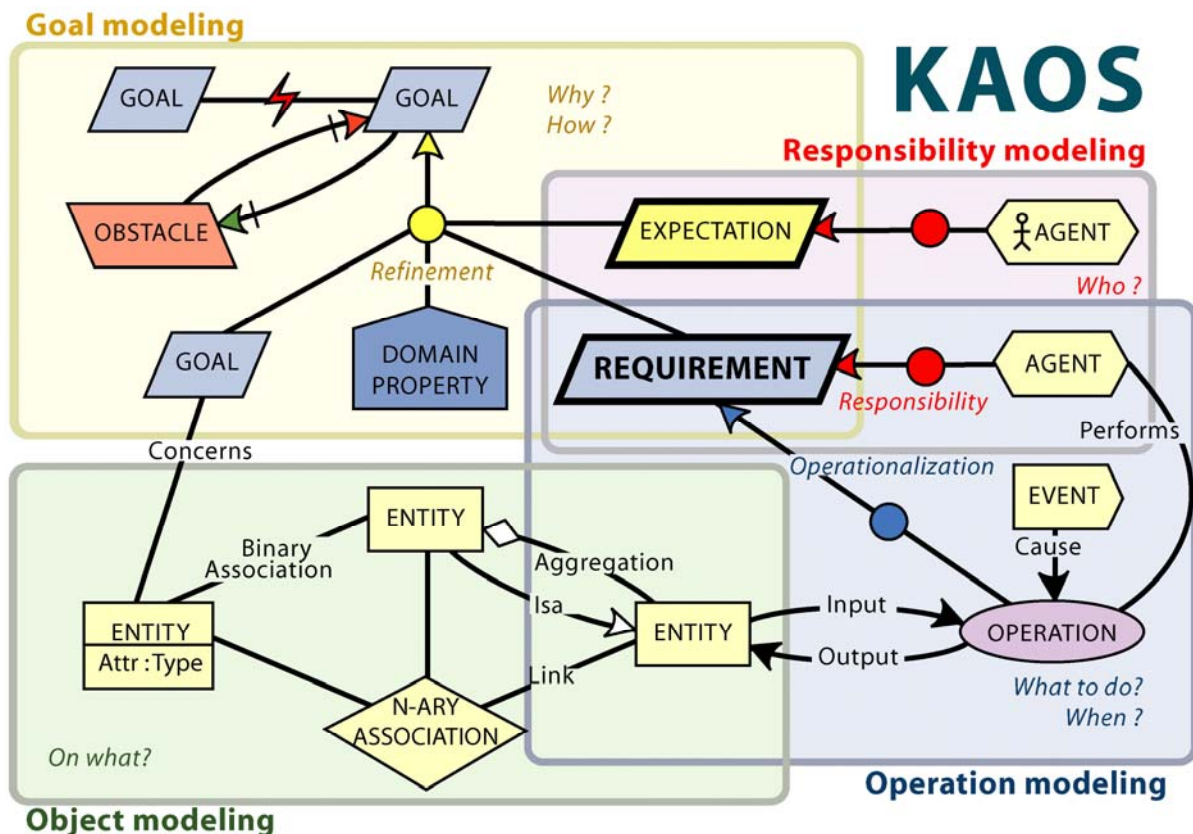
6. Methodology summary

6.1. KAOS meta-model

The KAOS/Objectiver methodology (www.objectiver.com) is a requirements engineering methodology that covers identification of the business requirements, of the requirements, of the responsible agents and, if needed, of the behaviors they need to conform in order to satisfy the requirements.

The methodology relies on the construction of a requirements model, the graphical part of which is represented by means of diagrams displayed in this document.

The following figure sums up the set of concepts and notations you may find in the methodology. Next a glossary explains each concept shortly in turn.



6.2. KAOS glossary

Agent

Active Object (=processor) performing operations* to achieve goals*. Agents can be the software being considered as a whole or parts of it. Agents can also come from the environment* of the software being studied; human agents are in the environment*.*

Association

Object, the definition of which relies on other objects linked by the association.*

Composite system

The software being studied and its environment.*

Conflict

Goals are conflicting if under some boundary condition the goals cannot be achieved altogether.*

Domain Property

Descriptive assertion about objects in the environment* of the software. It may be a domain invariant or a hypothesis. A domain invariant is a property known to hold in every state of some domain object, e.g., a physical law, regulation, ... A hypothesis is a property about some domain object supposed to hold.*

Entity

Autonomous object, that is, the definition of which does not rely on other objects.*

Environment

Part of the universe capable of interaction with the software being studied.

Event

Instantaneous object (that is, an object alive in one state only) which triggers operations* performed by agents*.*

Expectation

Goal assigned to an agent* in the environment*.*

Formal model

Model in which the concepts have been mathematically formalised. In Objectiver, the formal model is built on top of a semi-formal model. Only a part or the whole semi-formal model can be formalised. Objectiver uses first-order temporal logic with real-time extensions to formalise concepts.*

Goal

Prescriptive assertion capturing some objective to be met by cooperation of agents; it prescribes a set of desired behaviours. Requirements* and expectations* are goals.*

Model

Abstract representation of a composite system. An Objectiver model represents a composite system* by means of concepts of different types, mainly, objects*, desired or undesired properties (goals*, obstacles*), and behaviours (operations*).*

Object

Thing of interest in the composite system being modelled whose instances can be distinctly identified and may evolve from state to state. Agents, events, entities and associations are objects.*

Obstacle

Condition (other than a goal) whose satisfaction may prevent some goal(s) from being achieved; it defines a set of undesired behaviours.*

Operation

Specifies state transitions of objects that are input and/or output of the operation. Operations are performed by agents*.*

Operationalisation

Relationship linking a requirement to operations*. Holds when each execution of the operations (possibly constrained to that intent) will entail the requirement*. Makes the connection between expected properties (goals*) and behaviours (operations*)*



Refinement

Relationship linking a goal to other goals that are called its subgoals. Each subgoal contributes to the satisfaction of the goal* it refines. The conjunction of all the subgoals must be a sufficient condition entailing the goal* they refine.*



Requirement

Goal assigned to an agent* of the software being studied.*

Responsibility

Relationship between an agent and a requirement*. Holds when an agent* is assigned the responsibility of achieving the linked requirement*.*



Semi-formal model

Model in which the concepts are not mathematically formalised In Objectiver, every concept in the model receives a name, a type, a textual definition, values for attributes and a graphical representation.*