UCL
Université
catholique
de Louvain

Faculté des Sciences Appliquées

Département d'Ingénierie Informatique

Reasoning about Agents in Goal-Oriented Requirements Engineering

Emmanuel Letier

Thèse presentée en vue de
l'obtention du grade de
Docteur en Sciences Appliquées

22 Mai 2001

UCL

Université catholique de Louvain

Faculté des Sciences Appliquées

Département d'Ingénierie Informatique

Reasoning about Agents in Goal-Oriented Requirements Engineering

Emmanuel Letier

Jury: MM. A. De Herde, président
Y. Deville,
A. Finkelstein,
A. van Lamsweerde, promoteur,
T. Maibaum,
M. Sintzoff.

22 Mai 2001

Thèse presentée en vue de
l'obtention du grade de
Docteur en Sciences Appliquées

Abstract

The thesis proposes a number of techniques for elaborating requirements constructively from high-level goals. The techniques are based on the KAOS goal-oriented method for requirements engineering. This method consists in identifying goals and refining them into subgoals until the latter can be assigned as responsibilities of single agents such as humans, devices and software. Domain properties and assumptions about the software environment are also used during the goal refinement process. The method supports the exploration of alternative goal refinements and alternative responsibility assignments of goals to agents. It also supports the identification and resolution of conflicts between goals, and the identification and resolution of exceptional agent behaviors, called obstacles, that violate goals and assumptions produced during the goal refinement process.

The thesis enriches the KAOS framework through three kinds of techniques:

(a) techniques for identifying agents, goal refinements, and alternative responsibility assignments, and for deriving agent interfaces from such responsibility assignments;

(b) techniques for deriving operational requirements from goal specifications;

(c) techniques for generating obstacles to the satisfaction of idealized goals and assumptions, and for generating alternative obstacle resolutions.

The result is a coherent body of systematic techniques for requirements elaboration that are both theoretically well-founded (a formal model of agent is defined) and effective in practice (the techniques are validated on two real case studies of significant size: the London ambulance despatching system, and the Bay Area Rapid Transit train system).

Acknowledgments

Foremost, I thank my supervisor, Axel van Lamsweerde, for his meticulous feedback, helpful guidance, and precious experience. This work would not have been possible without his human, technical and financial support.

I thank the members of the jury, A. De Herde, Y. Deville, A. Finkelstein, T. Maibaum and M. Sintzoff for their valuable comments and words of encouragements.

I benefited from many discussions with my colleagues students, Laurent Willemet and Christophe Ponsard, who with me struggled with the KAOS language and method. Valuable experience about the use of KAOS in industrial settings was provided by people at CEDITI, in particular by Robert Darimont, Philippe Massonet and Emmanuelle Delor.

Sylvie Bourguignon, my wife, supported me throughout these years with all her patience and love. I also thank my parents, grandparents, parents-in-law and Laurette, brothers, brothers- and sisters-in-law, nephews and friends.

This research was supported by the “Communauté Francaise de Belgique” (FRISCO project, Action de Recherche Concertées Nr. 95/00-187 - Direction générale de la Recherche).

Contents

Chapter 1
Introduction

Requirement engineering (RE) is the very first step of the system development process. It is concerned with the identification of stakeholders' goals about the intended system; the specification of services and constraints that operationalize those goals; and the assignment of responsibilities for the resulting requirements to agents such as humans, devices and software [Lam2Kc].

Requirements engineering is now widely recognized to be among the most critical steps of system development. In order to implement a system that satisfies the stakeholders' needs, those needs must be clearly understood and adequately mapped to specifications of required software behaviour. Inadequate requirements engineering has been repeatedly pointed out to be a major source of problems in software development [Bel76, Sta95, ESI96]. The cost of correcting errors or misconceptions in requirements increases exponentially along the software life-cycle if such errors are not handled during the requirements engineering stage [Boe81]. It is thus essential that requirements engineering be done with great care and precision.

Goals play a prominent role in the requirements engineering process. Goals drive the elaboration of requirements to support them [Ros77, Dar91, Rub92]; they provide a completeness criterion for the requirements specification - the specification is complete if all stated goals are met by the specification [Yue87]; they provide a rationale for requirements - a requirement exists because of some underlying goal which provides a base for it [Dar91, Som97]; they are generally more stable than the requirements to achieve them [Ant94]. In short, requirements "implement" goals much the same way as programs implement design specifications.

Goals are to be achieved by the cooperation of various *agents*. Such agents may include software components that exist or are to be developed, external devices, and humans in the environment. The system being considered in the requirements engineering process is thus *composite* [Fea87]; it includes both the software-to-be and its environment.

Reasoning about agents is a critical aspect in the requirements elaboration process. It involves reasoning about alternative responsibility assignments of goals to agents [Fea87, Dar93]; reasoning about the agent's permissions and obligations to perform actions to achieve the goals [Ken93, Mai93]; reasoning about the required interactions between the different agents in terms of information they monitor, control and communicate [Fea87, Par95, Heim98]; reasoning about possibilities of agents' misbehaviors [Lam98]; and reasoning about agents' dependencies in an organizational environment [Yu93].

The elicitation of goals, their organization into a coherent structure, and their operationalization into requirements to be assigned to the various agents is at the core of requirements engineering. The thesis addresses two key concerns requirements engineers have to cope with during requirements elaboration: the exploration of alternative responsibility assignments of requirements to agents, and the handling of possible agents' misbehaviors.

Exploring Alternative Agents Responsibilities

A critical step of the requirements engineering process is the identification of the agents that should play a role in achieving the goals of the system; the definition of their responsibilities with respect to the goals to be achieved; and the identification of the quantities to be monitored and controlled by each agent in order to be able to achieve the goals it is assigned to [Par95, Zav97]. Alternative decisions about agent responsibilities, monitoring and control result in systems in which more or less functionality is automated and in which the interactions between the automated system and its environment may be quite different. Such decisions have a critical impact on the performance, cost and risks associated with the composite system. Responsibility assignment, however, is taken for granted by most specification techniques; no systematic support is available. As a result, alternative, perhaps superior, decisions about agent responsibilities, monitoring and control are not systematically explored; the rationale for such decisions is not made explicit for easier evolution.

Our general objective in this thesis is to provide systematic techniques for elaborating, from the high-level goals of the composite system, alternative agent assignments that satisfy those goals.

The exploration of alternative system proposals is at the heart of goal-oriented approaches to requirements engineering. Our work is based on an existing goal-oriented requirements elaboration method, called KAOS. The core of the method consists in identifying goals and refining them into subgoals until the latter can be assigned as responsibilities of single agents [Dar93].

Extensions to the KAOS language are proposed to model and reason about agent responsibilities, monitoring, and control. A *realizability* consistency rule is introduced to relate the responsibility of an agent for a goal to the agent's monitoring and control capabilities: the goal is realizable by an agent if it defines a relation between quantities monitored and controlled by the agent.

Realizability is seen to play a central role in the goal-oriented requirements elaboration process: violations of the realizability rule drives the identification of new agents and the refinement of goals into subgoals until the latter are realizable by individual agents.

We define a *taxonomy of realizability problems* to support the identification and classification of violations of realizability. Realizability problems include: lack of monitorability, lack of control, references to future, goal unsatisfiability, and time-unbounded achievement goals. This taxonomy is shown to be complete.

A library of *specification elaboration tactics* is then defined to guide the resolution of realizability problems. These tactics guide the elaboration of the requirements models by identifying new agents and by recursively refining goals into subgoals until the latter are realizable by single agents.

Once goals have been refined into subgoals that are realizable by single agents, the next step of the goal-oriented elaboration process consists in deriving the operations to be performed by the agents so as to satisfy the goals. *Formal operationalization patterns* are proposed to derive complete operational requirements from the formal definitions of realizable goals.

Handling Exceptional Agent Behaviours

Another important aspect of the software development process is the anticipation and handling of potential agent misbehaviors that might prevent goals from being achieved. The requirements elaboration process tends to produce requirements and assumptions about agent behaviors are often too ideal. Some of them are likely to be violated from time to time in the running system. The lack of anticipation of exceptional behaviors results in unrealistic, unachievable and/or incomplete requirements. As a result, the software developed from those requirements and assumptions will inevitably result in poor performance, sometimes with critical consequences on the environment.

Our objective here is to provide systematic techniques for reasoning about obstacles to the satisfaction of goals and assumptions elaborated in the requirements engineering process. A key principle is to tackle risks of unexpected agent behaviour as early as possible in the development process, that is, at the goal level. By reasoning about exceptions upfront in the development process, one is left with more freedom for resolving such exceptions, for instance, by identifying alternative system proposals in which the possibilities and consequences of agent misbehaviors are reduced.

The thesis describes systematic techniques for generating obstacles to the satisfaction of idealized goals and assumptions, and for generating alternative obstacle resolutions. The latter transform the goal model by deidealizing goals and assumptions or by generating new goals so as to avoid, reduce or tolerate the identified obstacles.

Contributions

The main contributions of the thesis are the following.

- A *formal model of agents* that provides the underlying semantic domain for a significant part of the KAOS language related to agents and their responsibilities. This model provides the basis for a precise definition of the *realizability* consistency rule that defines when an agent has sufficient monitoring and control capabilities to take responsibility for a goal.
- A *taxonomy of realizability problems* that allows one to identify unrealizable goals in a systematic way. This taxonomy is shown to be complete.
- A library of *agent-driven tactics* for recursively elaborating goal models and agents so as to resolve violations of the realizability rule.
- a library of *operationalization patterns* for deriving operational requirements from the formal definitions of realizable goals.
- a precise definition of the concept of *obstacle* to the satisfaction of goals, *obstacle identification techniques* for systematically generating obstacles from goals, and a library of *obstacle resolution tactics* for transforming the goal and agent models so as to resolve generated obstacles.
- An assessment of the proposed techniques on two real case studies of significant size: the LAS ambulance despatching system and the BART automated train control system.

Organization

The thesis is structured as follows.

Chapter 2 describes some background work on the modelling of goals and agents in RE.

Chapter 3 describes the KAOS goal-oriented specification language and method; it also introduces extensions to the language to model and reason about agent responsibilities, monitoring, and control.

Chapter 4 defines a formal model of agents that provides the underlying semantic domain for the KAOS language; it also gives a formal definition for the realizability consistency rule that relates agent responsibility for goals to its interfaces.

Chapter 5 describes a complete taxonomy of realizability problems used to identify unrealizable goals during the requirements elaboration process.

Chapter 6 defines agent-driven tactics for refining goals into subgoals and for identifying agents so as to resolve realizability problems.

Chapter 7 describes the use operationalization patterns for deriving operational requirements from realizable goals.

Chapter 8 is based on [Lam2Ka] and describes the handling of obstacles during the goal-driven requirements elaboration process.

Chapter 9 illustrates and assesses the techniques described in the thesis on the two forementioned cases studies.

Chapter 10 discusses related work.

Chapter 11 concludes and discusses further work.

Chapter 2
Goals and Agents in Requirements Modelling

This chapter briefly reviews some background works related to the modelling of goals and agents in requirements engineering. Some background on the handling of exceptional behaviours is introduced as well. Related work will be discussed in further detail in Chapter 10.

2.1. Foundations of RE

Before discussing particular requirements specification languages and methods, it is essential to understand the foundations of RE. Two important frameworks of requirements engineering have been proposed: the framework of Jackson and Zave [Jac95, Zav97], and the Four-Variable Model of Parnas [Par95].

The framework of Jackson and Zave [Jac95, Zav97] explains the precise nature of goals, requirements and domain properties, as well as the precise nature of the relationships among them[1]. In that framework, the term *machine* is used to denote the hardware/software to be developed. The purpose of the machine is to bring about some properties in the environment; therefore requirements engineering should only be concerned with modelling properties in the vocabulary of environment.

When modelling the environment, it is essential to distinguish between the quantities in the environment that are *monitored* by the machine, those that are *controlled* by the machine, and those in the environment outside the interface with the machine[2].

A further essential distinction is made between goals, domain properties and requirements:

- A *goal* is a desired property about quantities in the environment.
- A *domain property* is a property that naturally holds in the environment, as it would be without or in spite of the machine.
- A *requirement* is a special kind of goal that constrains the behavior of the machine. To be a requirement, a goal must satisfy the following three properties:

  (i) it is described entirely in terms of quantities monitored and controlled by the machine;

1. In order to keep a uniform terminology throughout the thesis, we changed the terminology used by Jackson and Zave. In their terminology, a goal is called a requirement, and a requirement is called a specification.
2. In the terminology of Zave and Jackson, monitored variables are defined as variables that are *shared* between the machine and its environment, and that are *controlled by the environment*; controlled variables correspond to *shared* variables between the machine and the environment that are *controlled by the machine*; environment variable outside the interface with the machine are unshared variables controlled by the environment.

(ii) it constrains only quantities that are controlled by the machine;

(iii) the controlled quantities are not defined in terms of future values of monitored quantities.

The main role of domain properties is to help bridge the gap from goals to requirements. More precisely, if G is the set of goals defined in terms of quantities in the environment, D the set of domain properties about the environment, and R the requirements on the machine, the following properties must hold:

1. the requirements, together with the domain properties must guarantee the satisfaction of the goals:

$$D, R \models G$$

2. the requirements must be consistent with the domain properties:

$$D, R \not\models \text{false}.$$

The Four-Variable Model of Parnas [Par95] is a similar model that introduces a further distinction between environmental quantities that are *monitored* and *controlled* by the system and the actual *inputs* and *outputs* of the software. The term *system* is used here to denote the software and its I/O devices.

This model, illustrated in Figure 1, emphasizes that requirements should be described as a relation between monitored and controlled variables. *Monitored variables* represent environmental quantities that the system measures. *Controlled variables* represent environmental quantities that the system controls. A black-box specification of required behaviour is given as two relations, REQ and NAT, from the monitored to the controlled variables. NAT defines natural constraints on the system behaviour, such as constraints imposed by physical laws and the system environment. REQ defines additional constraints on the system to be built as relations the system must maintain between the monitored and the controlled variables.

If the term machine used by Jackson and Zave is meant to represent the software and its I/O devices, the REQ relation of the Four-Variable Model is roughly equivalent to the requirements R in the previous framework. (A detailed comparison of these two frameworks can be found in [Gun2K].)

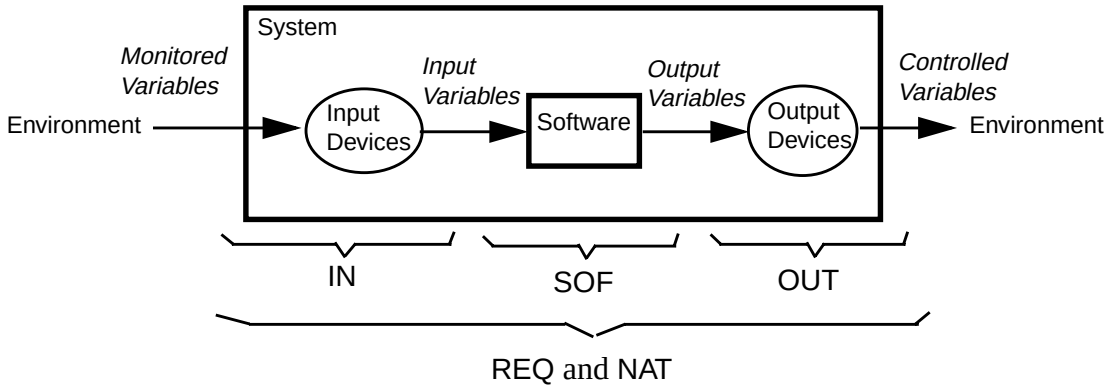


FIGURE 1. The Four-Variables Model

The Four-Variable Model introduces two additional sets of variables. *Inputs variables* represent the values actually stored in the input registers of the software system. *Output variables* represent the values of the output register of the software. A relation IN between monitored and input variables is used to describe the behaviour of the input devices. A relation OUT between output and controlled variables is used to describe the behaviour of the output devices.

The behaviour of the software is then described by a relation SOF between input and output variables. For the software to correctly implement the requirements, the relation SOF must satisfy:

$$\mathsf{IN} \wedge \mathsf{SOF} \wedge \mathsf{OUT} \wedge \mathsf{NAT} \Rightarrow \mathsf{REQ}.$$

Both frameworks insist on the well-known but often neglected importance of precisely defining the correspondence between mathematical variables and the physical quantities they denote.

The main advantage of these frameworks is that they provide *simple* models explaining the nature of requirements engineering. Simplicity is what makes these models so attractive and useful. Some aspects of requirements modelling have therefore been oversimplified, and some others have been ignored. In Chapter 4 of the thesis, we describe a more detailed, but more complicated, formal framework for requirements modelling.

A fundamental concern of requirements modelling is completely ignored in these frameworks: the engineering of requirements involves the exploration of *alternative* system proposals in which the boundary between the automated system and its environment may be quite different. The above models characterize only the result of this exploration; the boundary has been decided on and is frozen. Techniques supporting the exploration of alternative system proposals from goals is a central theme of the thesis.

2.2. Modelling Goals

The need to model WHY a system should be developed has been recognized since the early days of RE [Ros77]. However, most requirements modelling notations and techniques focus on the “late-phase” of the requirements engineering process [Yu97], during which initial statements of functional requirements are precisely reformulated and analyzed for ambiguity, incompleteness and inconsistency. Methods supporting this kind of analysis range from semi-formal (e.g. structured methods [Ros77], object-oriented methods [Rum91, Rum98]) to formal (e.g. history-based [Man92, Lamp94, Koy92], state-based [Pot91, Jon90, Abr96], or transition based [Har87, Heit96] -- see [Lam2Kb]).

Goal modelling is intended to address the “early-phase” of requirements engineering during which stakeholders goals are explored and alternative system proposals that satisfy the goals are investigated.

Two complementary frameworks for goal-oriented requirements engineering have emerged: the first one is KAOS [Dar93, Lam2Kc]; it is mostly concerned with the *generation* of alternative system designs from high-level goals defined in temporal logic. The second one is the NFR framework [Myl92, Chu2K]; it is mostly concerned with the *evaluation and selection* of alternatives with respect to qualitative ‘non-functional’ goals such as usability, performance, accuracy, security, etc.

In the *KAOS framework*, goals are related to subgoals through AND/OR refinement links. A goal is AND-refined into a set of subgoals when the satisfaction of all subgoals ensure the satisfaction of the parent goal. OR-refinement links relate a goal to an alternative set of AND-refinements; this means that satisfying one of the refinements is sufficient for satisfying the parent goal. Conflicts links between goals are also used to relate goals that cannot be satisfied together.

An optional formal assertion layer is used to formally specify goals in a real-time temporal logic [Man92, Koy92]. The formal definition of goals allows one to

- prove the completeness of goal-refinements, and identify overlooked goals and assumptions [Dar95, Dar96],
- identify conflicts between goals [Lam98b],
- generate obstacles from the definition of goals [Lam98a, Lam2Ka].

The *NFR framework* emphasizes the need to model and reason about non-functional requirements such as usability, performance, accuracy, security, etc. Such non-functional requirements provide criteria for evaluating and selecting among alternatives design decisions. The term “softgoal” is used to characterize goals that do not have a clear-cut criterion for their satisfaction. Weaker versions of goal refinement and goal conflict links are introduced to model softgoal dependencies. Instead of goal satisfaction, softgoal *satisficing* is introduced to express that lower-level softgoals are expected to achieve their parent softgoals within acceptable limits rather than absolutely. A labelling procedure has been defined to help determine the degree to which a softgoal is satisficed or denied by lower-level softgoals and requirements. The NFR framework is also composed of a rich catalogue of generic expert knowledge about softgoals and softgoal dependencies.

2.3. Modelling Agents

The concept of agent has also been introduced in several requirements specification languages and methods.

Formal agent-oriented languages such as structured MAL [Rya91], and AlbertII [Dub95] have been proposed to model and reason about systems made of interacting agents. These languages are targeted to the “late phase” of the requirement elaboration process. They do not support the process of identifying which agents should play a role in the system and what requirements each agent should be responsible for. In contrast, our work is concerned with the “early phase” of the requirement engineering process during which the goals of the system-to-be are identified and alternative operationalizations and responsibility assignments are still being investigated.

Our work has been significantly influenced by the paradigm of Composite System Design proposed in [Fea87] for the development of systems involving multiple interacting agents. This approach consists in separately specifying the global constraints on the behaviors of a multi-agent system and gradually deriving the local constraints on the behavior of individual agents. The notion of *responsibility* was identified there as playing a major role in describing the stages of decomposition of global constraints into local constraints on agent behaviors. Different responsibility assignments may lead to radically different designs. To support this kind of requirements elaboration process, [Fea87]

introduces a simple formal framework for modelling agents, agent interfaces, and agent responsibility for goals. An agent responsible for a goal is the only one required to restrict its behaviour so as to ensure the goal.

This framework has been defined further in [Fic92] where formal techniques are suggested for (i) identifying inconsistencies between operational specifications of individual agents and declarative specifications of global goals; and (ii) resolving the identified inconsistencies by transformations of the operational and declarative specifications. That paper describes a small core of specification elaboration operators capable of generating a whole range of composite system designs. These operators were discovered empirically by studying various existing composite systems and trying to rationally rederive their features.

The work reported in [Fea94] proposes another set of techniques based on the weakest precondition calculus [Dij76] and finite differencing techniques; the aim is to derive an operational specification of agents from global goals. Informal reasoning about lack of agent monitoring and control capabilities is used to guide the derivation process.

2.4. Exceptional Behaviours

Exceptional agent behaviours might obstruct the achievement of goals. These exceptional behaviours will be called obstacles.

Reasoning about exceptions during software development is of utmost importance to get high-quality software. There has been a lot of software engineering research to address this for the later stages of software architecturing or implementation, where the boundary between the software and its environment has been decided and cannot be reconsidered, and where the requirements specifications are postulated correct and complete [And81, Bor85, Per89, Cri91, Ros92, Jal94, Cri95, Aro98, Gar99]. In contrast, the techniques presented in the thesis work at the much earlier stage of requirements engineering, from goal formulations, so that more freedom is left concerning adequate ways of resolving goal violations -- like, e.g., considering alternative requirements or alternative agent assignments that result in different system proposals, in which more or less functionality is automated and in which the interaction between the software and its environment may be quite different.

Some work has been done at specification level though. The JSD method [Jack83] already recognized the need to anticipate and handle errors at that level. JSD provides techniques for handling inputs which are not valid for a given specification (such as meaningless inputs or inputs arriving in an unexpected order). Jackson also recognized that mistaken valid inputs cannot be handled by the proposed techniques, as they may require transformation of the whole specification, and that such errors should be taken into account in the earlier steps of the specification elaboration process. However, no techniques are provided there to anticipate and resolve such errors. The techniques described in Chapter 8 are intended to fill that void.

Systematic techniques have been defined for analyzing a formal model of the automated system for exceptional cases. The Z precondition calculus makes it possible to systematically identify exceptional cases in which an operation could be applied [Pot91]. The operation can then be made robust by separately specifying responses to those exceptional cases. The Z logical schema combination constructs are used to combine normal

and exceptional cases. With the same objective of making a specification complete against all possible inputs, automatic techniques have been defined to check whether a state machine model handles all possible sequence of inputs [Heim96, Heit96].

For safety-critical systems, hazard analysis techniques such as fault-tree analysis have been developed to identify and reason about the causes of hazards of a system [Lev95]. The primary purpose of such techniques is to analyze the causes of hazards, not to identify the hazards over which the analysis should start. A good understanding of the system is essential for identifying the top hazards to be analyzed, and for elaborating the fault trees.

Deontic logics are formalisms that enable one to specify and reason about normal and abnormal situations by means of modal operators such as permission and obligation [Mey93]. Such logics have been proposed for system specification, making it possible to specify what should happen if an abnormal situation occurs [Mai93, Ken93]. However such approaches do not provide any guidance for elaborating the requirements, in particular the requirements dealing with the abnormal situations. In contrast, the techniques discussed in Chapter 8 are based on goals which serve as a rationale for introducing new requirements to deal with the abnormal situations.

The concept of *obstacle* was first introduced in [Pot95] to describe situations that may block the fulfilment of goals. Obstacles are identified there by exploration of scenarios of interaction between software and human agents. This exploration is informal and based on heuristics. Some limited obstacle resolution strategies such as obstacle defence and obstacle mitigation are sketched there. Chapter 8 builds on that work by providing more systematic techniques for identifying and resolving obstacles to the satisfaction of goals.

Chapter 3
Goal-Oriented Requirements Engineering with KAOS

This chapter describes the KAOS goal-oriented specification language and method used in the dissertation. We first motivate our choice of a goal-oriented approach for exploring alternative responsibility assignments and for handling agent misbehaviours during requirement engineering. The KAOS goal-oriented specification language is presented in Section 3.2. The KAOS goal-driven method for elaborating requirements is then described in Section 3.3.

The chapter also introduces extensions of the KAOS language that are necessary to model and reason about agent responsibility, monitorability and control. The *agent interface model* is added to the KAOS language to model the monitoring and control capabilities of agents with respect to object attributes (Section 3.2.8). A *realizability* consistency rule that relates agent responsibility for goals to agent interfaces is introduced.

The realizability consistency rule plays a significant role in the goal-oriented requirement elaboration process: violation of that constraint drives the refinement of goals into subgoals and the identification of new agents (Section 3.3.5). The realizability consistency rule provides the basis for the formal techniques proposed in the thesis; techniques for identifying and resolving unrealizability are described in Chapters 5 and 6, respectively.

Much effort was also put into clarifying and simplifying various other aspects of the language. Portions of the KAOS language that significantly differ from or extend the original description of the language in [Dar93] are highlighted with a bar in the margin.

3.1. Why a Goal-Oriented Approach?

As mentioned in Chapter 2, goals play a prominent role in the requirement engineering process. In particular, they are well-suited to support the exploration of alternative designs involving multiple agents and the handling of agent misbehaviours.

The exploration of alternative system designs is at the core of requirements engineering [Fea87, Fick92, Dar93, Myl99]. Alternative ways of refining goals into subgoals, and alternative responsibility assignments of goals to agents yield different designs in which the boundary between the automated systems and its environment may be completely different. Goals are also used to identify the agents which should play some role in the system. The introduction of a new agent arises from the need to fulfil some system-wide goals.

Goals also provide the basis for identifying agent misbehaviours. The refinement of goals into subgoals involves making assumptions about agents in the environment. Goals and assumptions define the ideal behaviour of agents. Agent misbehaviours are then defined as exceptions which cause these goals and assumptions to be violated. Further-

more, handling agent misbehaviours at the goal level gives more freedom for resolving such exceptions, for instance by considering alternative goal refinements and responsibility assignments that yield different system designs.

3.2. The KAOS Goal-Oriented Requirement Specification Language

This section introduces the KAOS goal-oriented language used in the dissertation. Section 3.2.1 gives an overview of the language. Section 3.2.2 introduces the conceptual modelling aspects of the language and the three levels of modelling involved in requirement acquisition. Section 3.2.3 defines the temporal logic used in KAOS for specifying requirements formally. Sections 3.2.4 to 3.2.8 describe the various components of the language in more detail.

3.2.1 Overview

The KAOS language is a multiparadigm language. A KAOS model is composed of several submodels related through inter-model consistency rules (Figure 3.1).

The goal model is the driving model of the KAOS language. It declares the goals of the composite system. A *goal* defines an objective the composite system should meet, usually through the cooperation of multiple agents. An example of goal for a meeting scheduling problem is the goal Achieve[ConvenientMeetingHeld] requiring that every requested meeting is eventually held with the presence of all intended participant.

Goal-refinement links relate a goal to a set of subgoals. A set of subgoals refines a parent goal if the satisfaction of all subgoals is sufficient for satisfying the parent goal [Dar93, Dar95, Dar96]. As an example, the goal Achieve[ConvenientMeetingHeld] is refined in Figure 3.1 into the subgoals Achieve[PrtcptsCstrKnown], Achieve[ConvenientMeetingPlanned], and Achieve[PrtcptsInformed].

A parent goal may be refined by alternative sets of subgoals. For instance, the goal Achieve[PrtcptsCstrKnown] can alternatively be refined into the subgoals Achieve[PrtcptsCstrRequested] and Achieve[RequestedCstrProvided] or into the subgoals Maintain[ElectronicAgendaUpToDate] and Achieve[PrtcpsCstrKnownFromAgenda]. Note that alternative goal refinements may yield alternative system design.

In addition to goal refinements, *conflicts* between goals can also be captured [Lam98a].

The goal model has a two-layer structure. An outer semantic net layer [Bra85] is used for declaring goals and goal links. An inner textual layer is used for defining goals. Goals are defined in natural language and may optionally be defined formally in a real-time temporal logic formalism similar to [Koy92] - see Section 3.2.3 hereafter.

The object model declares the objects of interest in the application domain. An object is classified as *entity, relationship, event* or *agent* dependent on whether the object is autonomous, subordinate, instantaneous or active. The object's characteristics are declared as *attributes* and relationship links to other objects. Inheritance between objects is declared through *ISA* relationships.

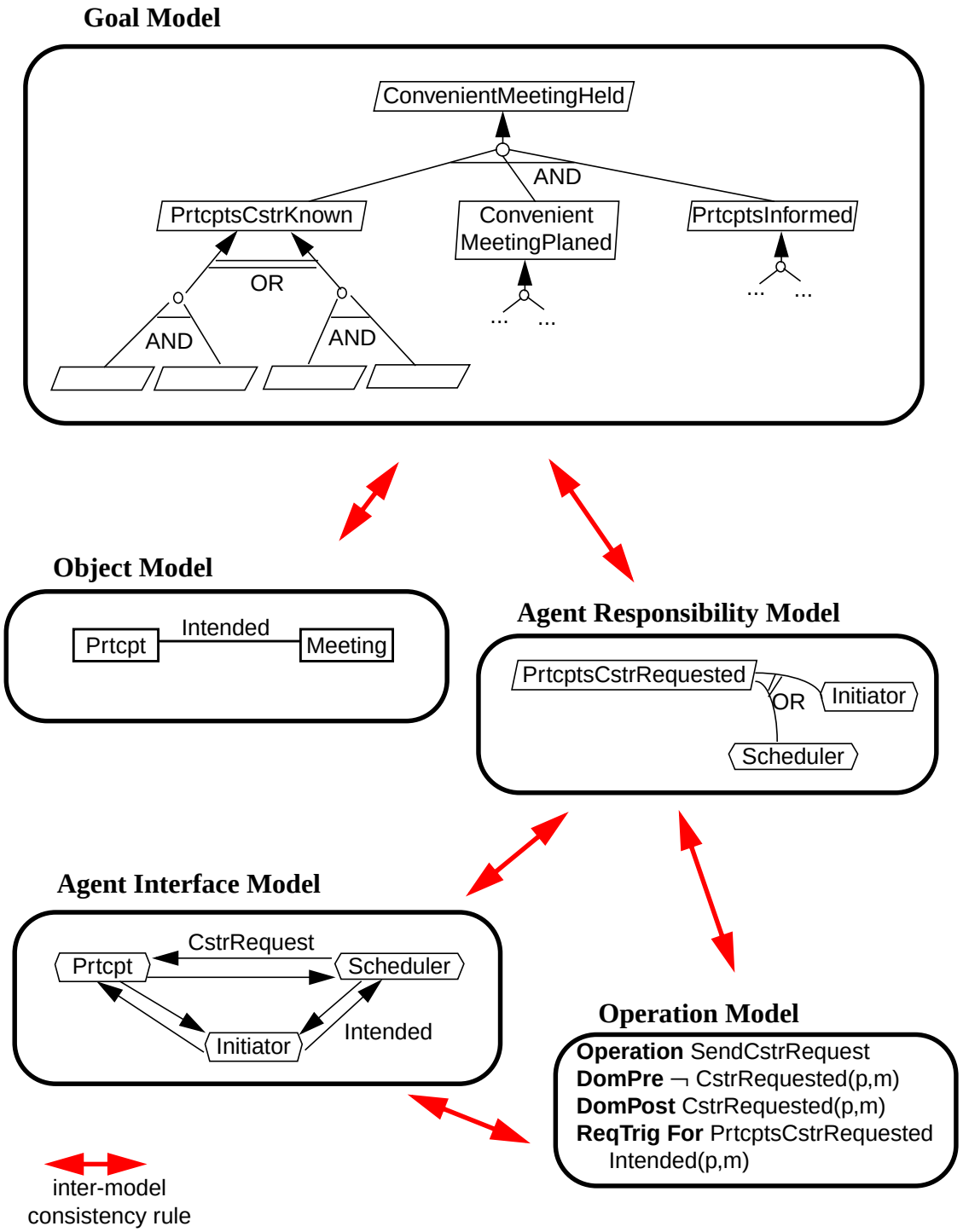


FIGURE 3.1. Overview of the KAOS models

There are inter-model rules between the object model and the goal model. On one side, the object model declares the vocabulary to be used in the definition of goals; on the other side, these definitions bound the vocabulary to be declared in the object model.

The agent responsibility model declares *responsibility* assignments of goals to agents. Agents include software components that exist or are to be developed, external devices, and humans in the environment. Responsibility assignment provides a criterion for stopping the goal refinement process. A goal assigned as the responsibility of a single agent is not refined any further. Alternative responsibility assignments are captured through OR responsibility links. For instance, the goal Achieve[PrtcptsCstrRequested] could be assigned the Scheduler software agent *or* to the Initiator agent. The meaning of a responsibility assignment is that the agent responsible for a goal is the only one required to restrict its behaviour so as to ensure that goal [Fea87].

The operation model defines the state transitions in the application domain. Operation are defined through *domain pre- and post-conditions*. For instance, the domain pre- and post-conditions of the SendCstrRequest in Figure 3.1 minimally capture what any sending of a constraint request is about in the application domain.

Further requirements on operations are necessary to ensure the goals assigned to individual agents. Such requirements are specified through *required pre-, trigger, and post- conditions*. They are related to the goal they ensure through *operationalization* links. For instance, the required trigger condition on the SendCstrRequest operation requires that a constraint request must be sent to a participant when the participant is intended to the meeting. This required trigger condition operationalizes the goal Achieve[PrtcptsCstrRequested].

Each operation is also related to the agent that can initiate it through a *performance* link.

The agent interface model declares which objects are monitored and controlled by each agent. In the agent interface model of Figure 3.1, the Scheduler agent controls the CstrRequested relationship, and monitors the Intended relationship. Alternative agent interface models can be captured through OR monitorability and control links.

There is an inter-model consistency rule between the agent interface and the corresponding responsibility assignment of a goal to an agent. Roughly speaking, *a goal can be assigned as the responsibility of an agent only if the goal is stated in terms of objects that are monitorable and controllable by the agent*. As will be seen later, this consistency rule, called the *realizability* rule, is a central concept of the thesis.

The KAOS language has a two-layer structure: an outer conceptual modelling layer for declaring concepts (such as goals, objects, agents, etc) and links between concepts (such as goal refinements, responsibility assignments of goals to agents, etc.); and an inner formal assertion layer for formally defining concepts. Section 3.2.2 further introduces some background material on conceptual modelling. Section 3.2.3 introduces the formal language used at the inner layer of the KAOS language. The various models of the KAOS language are then described in further detail in Sections 3.2.4 to 3.2.8.

3.2.2 Conceptual Modelling

We first introduce the three levels of modeling involved in requirements acquisition; then we define more precisely conceptual modeling constructs relevant to the definition of the KAOS language.

3.2.2.1 The Meta, Domain, and Instance Levels

The KAOS approach to requirements engineering involves three different levels of modelling: the meta-level, the domain-level, and the instance level (Figure 3.2).

The KAOS language is defined through a conceptual meta-model. The ***meta-model*** provides domain-independent abstractions in terms of which domain-specific concepts are acquired. The meta-model is composed of *meta-concepts* (such as Goal, Agent, Relationship, Operation, etc.); *meta-relationships* (such as Refinement between Goals, Responsibility between Agent and Goal, Monitoring between Agent and Object, etc.); *meta-attributes* of meta-concepts or meta-relationships (e.g. Definition of Goal, Priority of Goal, DomPre- and DomPost conditions of Operation, etc.); and *meta-constraints* on meta-concepts and meta-relationships (e.g. “the vocabulary used in the definition of a goal must be defined in the object model”).

The ***domain model*** is composed of domain-specific instances of meta-concepts and meta-relationships. In Figure 2, the Meeting concept is an instance of the Entity meta-concept, the goal Achieve[PrtcptsCstrRequested] is an instance of the Goal meta-concept, the operation SendCstrRequest is an instance of the Operation meta-concept, etc. The components of a requirements model are thus acquired as domain-specific instances of meta-concepts, linked by instances of meta-relationships, and characterized by instances of meta-attributes. These components must furthermore satisfy the meta-constraints of the meta-model.

The ***instance model*** refers to specific instances of domain-level concepts. A specific meeting is an instance of the domain-level concept of Meeting.

Role of the meta-model

In the context of requirements elaboration, the KAOS meta-model fulfills two important roles.

1. The KAOS meta-model determines the content and structure of the requirement documents.

> Concepts of the meta-model such as Goal, Operation, Agent, etc. define what information must be included in the requirements model. Links between concepts (such as goal refinement links) provide structure to the requirements model.

2. The KAOS meta-model drives the requirements elaboration process.

> The components of a requirements model are acquired as domain-specific instances of meta-concepts. Meta-relationships between concepts guide the acquisition of related concepts. For instance, the Refinement meta-relationship guides the acquisition of subgoals of a given goal.

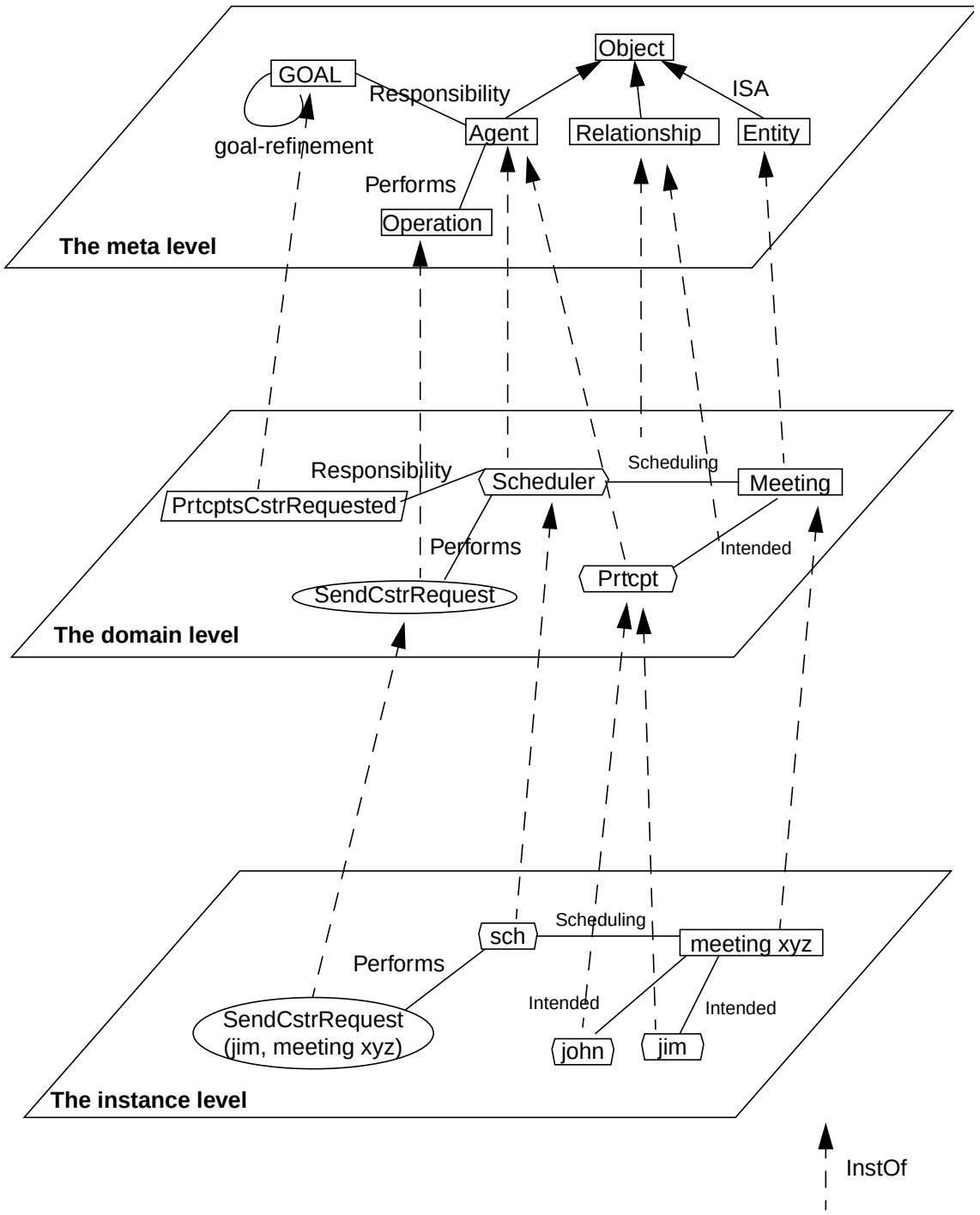


FIGURE 3.2. The meta, domain and instance levels

Meta-constraints play an important role during the requirement acquisition process. The violation of a meta-constraint drives the elaboration of the model so as to resolve the violation. For example, a violation of the meta-constraint linking goal definitions to the object model drives the identification of objects and attributes to be declared in the object model.

As will be seen later, one of the key ideas of the thesis corresponds to an application of this general principle; it consists in refining goals and identifying agents by systematically identifying and resolving violations of the realizability meta-constraint that relates goals and agent interfaces.

From the point of view of tools, the KAOS meta-model determines the structure of the requirement database in which the requirements model is gradually elaborated. The various components of the meta-model also yield criteria for measuring conceptual similarity when requirements are acquired by analogy with other systems [Mas97].

3.2.2.2 Characterizing meta-model components

As mentioned above, the meta-model is made of meta-concepts, meta-relationships linking meta-concepts, and meta-attributes attached to meta-concepts and meta-relationships. What is meant by meta-concept, meta-relationship and meta-attribute is now made more precise. AND/OR meta-relationships are also introduced to support the declaration of alternative requirements options at the domain level, and an Isa relation between meta-concepts is defined to support their specialization.

1. Meta-Concepts, meta-relationships and meta-attributes

A *meta-concept* C denotes a set of concept instances. The fact that a concept instance c belongs to a concept C is noted InstOf(c, C). Goal and Object are examples of meta-concepts of the KAOS meta-model. Specific domain-level goals and objects are respectively instances of the Goal and Object meta-concepts.

A *meta-relationship* R is a mathematical relation between n concepts C1,..., Cn; i.e.

$$R \subseteq C1 \times ... \times Cn$$

An instance of a relationship R is a tuple <c1,..., cn> with each $ci \in Ci$. We use the notation R(c1,..., cn) to denote that the tuple <c1,..., cn> is an instance of R. As an example, Concern is a meta-relationship linking the meta-concepts of Goal and Object. An instance of the Concern meta-relationship is a pair <G, Obj> where G is an instance of Goal and Obj an instance of Object. It declares that the object Obj is referenced in the definition of the goal G.

The cardinality of a concept Ci involved in a relationship R is a pair <mincard, maxcard> where mincard and maxcard denote the minimum and maximum number of instances of R in which every instance of Ci may participate, respectively.

A *meta-attribute* Att of a meta-concept or meta-relationship CR is defined as a function

$$Att: CR \rightarrow D$$

where D is called the domain of values of the attribute. For instance, the Goal meta-concept has the meta-attributes name and definition. The domain of values of the definition attribute of the Goal meta-concept is the set of natural language statements. The formal

definition of a goal is also a meta-attribute of the Goal meta-concept. Its domain of values is the set of logical formulas in the formal language used at the inner layer of the KAOS language.

2. And/Or meta-relationships

And/Or meta-relationships are introduced to support the declaration of alternative requirements fragments at the domain level. Since a goal can be refined into several alternative combinations of conjoined subgoals, the goal refinement meta-relationship is an AND/OR meta-relationship.

An *And/Or meta-relationship* R over concepts C and D is a compound binary relationship defined as follows [Dar93]:

R = AndR o OrR("o" denotes relation composition)

with $AndR \subseteq C \times AltR$, and $OrR \subseteq AltR \times D$, where AltR is a meta-concept that denotes the set of possible alternatives for the relationship R. An instance of R is a pair <c, {d1,..., dn}> such that there exists an alternative $alt \in AltR$ for which <c, alt> $\in$ AndR and <alt, di> $\in$ OrR for i =1..n.

And/Or meta-relationships have a Selected attribute with "yes" and "no" as possible values to record which alternatives are effectively selected.

And/Or meta-relationships are also subject to the following constraint:

for every $alt \in AltR$ there is exactly one $c \in C$ such that <c, alt> $\in$ AndR.

(In other words, the cardinality of AltR in the relation AndR is (1:1).)

For the Refinement And/Or meta-relationship, the constraint requires that each alternative refinement has exactly one parent goal.

An *Or meta-relationship* between concepts C and D is an And/Or relationship that satisfies the following additional constraint:

for every $alt \in AltR$ there is exactly one $d \in D$ such that <alt, D> $\in$ OrR

(In other words, the cardinality of AltR in the relation OrR is (1:1).)

As will be seen below, Responsibility is an example of Or meta-relationship between Agent and Goal; every alternative responsibility assignments links exactly one agent and one goal. (Several responsibility links may be used to assign several goals to the same agent.)

3. Meta-concepts specialization

Meta-concept specialization is captured through the binary ISA relation between concepts. This relation is defined by

Isa(C1, C2) iff every instance of C1 is also an instance of C2.

A consequence of meta-concept specialization is that attributes and relationships defined on C2 are also defined on C1.

3.2.3 Formal Specification of Timed Assertions

At the inner layer of the language, formal assertions can be attached to domain-level concepts. These assertions are values for the formal definition attribute of a goal; the pre, trigger, and post condition attributes of operations, etc. This section provides an introduction to the formal logic used in this dissertation.

Time plays a critical role in requirement engineering. Goals define sets of temporal behaviours of the system. There are many competing formalisms for expressing timed properties. The objective of the KAOS project has not been to define yet another logic, but to choose among existing formal languages. The choice for such a language resulted from a trade-off between the expressive power of the language and its formal analysis capabilities.

The logic used in KAOS is typed first-order real-time logic. Its real-time constructs are inspired by [Koy92]. It consists of the traditional temporal operators [Man92], together with additional real-time operators for specifying properties involving real-time deadlines. A key feature of the logic is its ability to model real-time properties concisely without referring explicitly to a time variable.

The temporal logic of [Koy92] has been extended with operators for specifying real-time properties referring to the absolute time of the system, and for specifying real-time properties in which the real-time bounds are allowed to change over time.

A limitation of the logic currently used in KAOS is that the time domain is assumed to be discrete. This makes it difficult to accurately capture and reason about properties involving time derivatives and integrals of time-continuous variables.

Note that the outer layer of the KAOS language is independent from the choice of formalisms at the inner layer of the language. The outer layer of the language can be given a precise semantics that is independent from the inner layer of the language. As a consequence, one can learn and use the outer layer of the language in a precise way without knowledge of the formal layer of the language. One can also more easily change or adapt the formalism used at the inner layer of the language, for instance, by replacing the temporal logic based on a discrete temporal domain by a temporal logic based on a continuous time domain.

We now introduce the temporal operators used in the dissertation. These operators are classified into operators for specifying qualitative temporal properties, relative real-time properties, absolute real-time properties, and flexible real-time properties.

3.2.3.1 Qualitative Temporal Properties

Qualitative temporal properties are properties about sequences of states. Examples of such properties are:

“every request is eventually satisfied”

“every state satisfies some state invariant I”

Qualitative temporal properties are specified using the following classical operators of temporal logics [Man92]:

◊ (some time in the future)	♦ (some time in the past)
❑ (always in the future)	■ (always in the past)
W (always in the future *unless*)	*B* (always in the past *back to*)
U (always in the future *until*)	*S* (always in the past *since*)

The semantics of such operators is defined over a linear temporal structure. An history is a function

h: N → State,

where N is a totally ordered set of time points, and State is the set of possible global states of the system. (Formally, State is the set of interpretation functions for the state variables of the model -- see Section 4.2.1.) In the sequel we take N to be the set of natural numbers.

The notation

(h, i) |= P

is used to express that assertion P is satisfied at time i of history h. An assertion is said to be satisfied by an history h iff it is satisfied at the initial time of the history, that is,

h |= P iff (h, 0) |= P.

The semantics of the above temporal operators is defined as follows [Man92]:

Future operators:

(h, i) |= ◊ P	iff	(h, j) |= P for some j ≥ i
(h, i) |= ❑ P	iff	(h, j) |= P for all j ≥ i
(h, i) |= P *U* Q	iff	there exists a j ≥ i such that (h, j) |= Q and for every k, i ≤k < j, (h, k)|= P
(h, i) |= P *W* Q	iff	(h, i) |= P *U* Q or (h, i)|= ❑ P

Past operators:

(h, i) |= ♦ P	iff	(h, j) |= P for some j ≤ i
(h, i) |= ■ P	iff	(h, j) |= P for all j ≤ i
(h, i) |= P *S* Q	iff	there exists a j ≤ i such that (h, j) |= Q and for every k, j < k ≤ i, (h, k)|= P
(h, i) |= P *B* Q	iff	(h, i) |= P *S* Q or (h, i) |= ■ P

The logic also uses the classical logical connectives ∧ (and), ∨ (or), ¬ (not), → (implies), ↔ (equivalent), and the usual quantifiers ∀ (for all), ∃ (exists).

We also use the following standard notations for entailment and strong equivalence:

P ⇒ Q	iff	❑ (P → Q)
P ⇔ Q	iff	❑ (P ↔ Q)

Note that there is an implicit ❑-operator in every entailment.

Since the temporal domain is assumed to be discrete, the following operators can also be introduced:

❍ (in the next state)●(in the previous state)

The semantics of these operators is defined as follows:

(h, i) |= ❍ P iff (h, i+1)|= P
(h, i) |= ● P iff (h, i-1)|= P and i>0

We also introduce the SCR-like notation @ P to denote that an assertion P has just become true:

@ P iff ● ¬ P ∧ P

The next and previous operators are also defined on *terms* of the language, i.e.,

VAL(h, i)(❍ T) = VAL(h, i +1)(T)
VAL(h, t)(● T) = VAL(h, i -1)(T) if i>0, otherwise it may have any value.

where T is a term of the language, and VAL(h,i)(T) is the valuation function of term T at time i of history h.

3.2.3.2 Relative Real-Time Properties

In requirement engineering, one often needs to specify real-time properties. Examples of such properties are:

"every request should be satisfied within an hour"

"every borrowed book copy should be returned within 2 weeks"

Relative real-time properties are properties referring to real-time delays between system states. In order to specify such properties, bounded versions of the above temporal operators are introduced in the style advocated by [Koy92]. Examples of such operators are:

$\Diamond_{\leq d}$ (some time in the future within deadline *d*)
$\Box_{\leq d}$ (always in the future up to deadline *d*)

To define such operators, the temporal structure N is enriched with a metric domain D and a temporal distance function

dist: $N \times N \rightarrow D$

which has all desired properties of a metric [Koy92]. We will take

D: {d | there exists a natural *n* such that $d = n \times \delta$},
where δ denotes some chosen time unit
dist(i, j): $| j - i | \times \delta$

The ❍-operator then yields the nearest subsequent time position according to the time unit.

The semantics of the real-time operators is then defined accordingly, e.g.,

Future operators:

$(h, i) \models \Diamond_{\leq d} P$ iff $(h, j) \models P$ for some $j \geq i$ with $dist(i, j) \leq d$
$(h, i) \models \Box_{\leq d} P$ iff $(h, j) \models P$ for all $j \geq i$ such that $dist(i, j) \leq d$
$(h, i) \models P\ U_{\leq d}\ Q$ iff $(h, j) \models Q$ for some $j \geq i$ with $dist(i, j) \leq d$ and for every k, $i \leq k < j$, $(h, k) \models P$
$(h, i) \models P\ W_{\leq d}\ Q$ iff $(h, i) \models P\ U_{\leq d}\ Q$ or $(h, i) \models \Box_{\leq d} P$

Past operators:

$(h, i) \models \blacklozenge_{\leq d} P$ iff $(h, j) \models P$ for some $j \leq i$ with $dist(j, i) \leq d$
$(h, i) \models \blacksquare_{\leq d} P$ iff $(h, j) \models P$ for all $j \leq i$ with $dist(j, i) \leq d$
$(h, i) \models P\ S_{\leq d}\ Q$ iff $(h, j) \models Q$ for some $j \geq i$ with $dist(j, i) \leq d$ and for every k, $i \leq k < j$, $(h, k) \models P$
$(h, i) \models P\ B_{\leq d}\ Q$ iff $(h, i) \models P\ S_{\leq d}\ Q$ or $(h, i) \models \blacksquare_{\leq d} P$

3.2.3.3 Absolute Real-Time Properties

Some real-time properties refer to the absolute time of the system, as in the following example

"every borrowed book copy should be returned by the end of the year for inventory."

In order to capture such properties, we introduce bounded versions of qualitative temporal operators in which the real-time bounds refer to the *absolute time* of the system, e.g.,

$\Diamond_{\leq T}$ (some time in the future before time T)
$\Box_{\leq T}$ (always in the future up to time T)

To define such operators, the temporal structure N is enriched with a function

$time: N \rightarrow Time$

which assign to each time point the current time at that point. We define such function as follows:

$time(i) =_{def} time(0) + dist(0, i)$

The time at the initial time point, i.e. time(0), is arbitrarily chosen. The semantics of absolute real-time operators is then defined as follows, e.g.,

$(h, i) \models \Diamond_{\leq T} P$ iff $(h, j) \models P$ for some $j \geq i$ such that $time(j) \leq T$
$(h, i) \models \Box_{\leq T} P$ iff $(h, j) \models P$ for all $j \geq i$ such that $time(j) \leq T$

3.2.3.4 Flexible real-time Properties

Flexible real-time properties are properties involving deadlines and delays whose values are not necessarily fixed and may be time-dependent, as in the following example:

"When a reviewer accepts to review a paper, the review of the paper should be available to the associate editor in charge of the paper within some delay which has been agreed between the associate editor and the reviewer."

In this last example, the delay for returning the review is a variable whose value may change over time; it could be extended or shortened.

Flexible real-time properties are the most frequent real-time properties in requirement engineering. Most often, the real-time bound is a time-dependent variable. (Note that the operators defined in the previous sections only allowed for constant real-time bounds.)

In order to capture such properties, we introduce bounded versions of qualitative temporal operators in which the relative or absolute real-time bounds refer to variables whose values may change over time, e.g.,

$\Diamond_{\leq d}$ P: P holds before delay d has expired,
where d is a time-dependent variable of type D

$\Diamond_{\leq T}$ P: P holds before deadline T is passed,
where T is a time-dependent variable of type Time

These operators are defined over the temporal structure previously introduced.

The formal definition of such operators requires some attention. We first consider the operator $\Diamond_{\leq T}$ P where T is a time-dependent *deadline*.

The operator is formally defined as follows:

$(h, i) \models \Diamond_{\leq T} P$ iff there exists $j \geq i$ such that $(h, j) \models P$
and for all k such that $i \leq k \leq j$, $time(k) \leq VAL_{(h,k)}(T)$

where $VAL_{(h,k)}(T)$ is the value of the time-dependent variable T at the time position k of history h.

The definition asserts that there is a future time point j at which P holds and that form the current time point i up to j, the current time is less that the current value for the deadline. As a consequence, if the deadline is extended before the initial deadline is passed, the assertion is satisfied provided P holds before the extended deadline. However, if the deadline is extended after the initial deadline is passed, the assertion is violated if P did not hold before the first deadline.

This operator is similar to the bounded obligation operator in [Ken93]. It is defined here for declarative temporal assertions, whereas the bounded obligation operator is defined for a logic of operations.

The operator $\Diamond_{\leq d}$ P when d is a time-dependent *delay* is similarly defined as follows.

$(h, i) \models \Diamond_{\leq d} P$ iff there exists $j \geq i$ such that $(h, j) \models P$
and for all k such that $i \leq k \leq j$, $time(k) - time(i) \leq VAL_{(h,k)}(d)$

The definition of the past temporal operators is much simpler. The deadline is evaluated in the current state of interpretation; this corresponds to the intuition that when talking about the past, the deadline or delay mentioned is by default the deadline as currently defined. As an example, the operator $\blacksquare_{\leq d}$ when d is a time-dependent delay is formally defined as follows:

$(h, i) \models \blacksquare_{\leq d} P$ iff $(h, j) \models P$ forall $j < i$ such that $time(i) - time(j) \leq VAL_{(h,i)}(d)$.

3.2.4 The Goal Model

Having introduced the conceptual modelling framework and the temporal assertion framework, we now detail the various submodels that together define a KAOS model.

3.2.4.1 Defining Goals

As mentioned before, a goal defines an objective the composite system should meet usually through the cooperation of multiple agents. For example, a goal in a meeting scheduling problem would be that each requested meeting is eventually held with the presence of all intended participants. This ideal goal might be captured by the following specification fragment.

Goal Achieve[ConvenientMeetingHeld]
Definition each requested meeting is eventually being held with the presence of all intended participants.
FormalDef $\forall$ m: Meeting:
m.Requested
$\Rightarrow \Diamond$
m.Holds$\wedge$ ($\forall$ p: Participant): Intended(p,m) $\rightarrow$ Participates(p,m)

Each goal has a *name*, a natural language *definition*, and an optional *formal definition*. The above goal is named Achieve[ConvenientMeetingHeld] (the Achieve verb is a keyword that will be explained below).

A goal defines a set of admissible histories in the composite system. Intuitively, an history is a temporal sequence of states of the system. Each goal is satisfied by some histories and falsified by some other histories. The notation

h |= G

is used to express that history h satisfies the goal G.

The definition of a goal is a natural language statement describing the set of histories satisfying the goal. The formal definition of a goal is a temporal logic formula describing the same set of histories. (It is the specifier's responsibility to ensure that the natural language and formal definitions of a goal describe the same property.)

3.2.4.2 Classifying Goals

A *goal taxonomy* is used to guide the acquisition and definition of goals. Goals are classified according to their pattern and category.

The *pattern* of a goal is based on the temporal behaviour required by the goal. The KAOS language distinguishes the following four goal patterns:

Achieve goals: goals requiring that some property eventually holds
Cease goals: goals requiring that some property eventually stops to hold
Maintain goals: goals requiring that some property always holds
Avoid goals: goals requiring that some property never holds

Goal patterns provide a lightweight way of declaring the temporal behaviour of a goal without writing formal goal definitions.

The pattern of a goal can also be used to guide the specification of the formal definition of the goal: goal patterns constrain the formal definition of goals [Dar98]. Goal patterns and corresponding temporal formula templates include the following:

Achieve:	$P \Rightarrow \Diamond Q$,	$P \Rightarrow \Diamond_{\leq d} Q$,	$P \Rightarrow \bigcirc Q$
Cease:	$P \Rightarrow \Diamond \neg Q$,	$P \Rightarrow \Diamond_{\leq d} \neg Q$,	$P \Rightarrow \bigcirc \neg Q$
Maintain:	$P \Rightarrow Q$,	$P \Rightarrow \Box Q$,	$P \Rightarrow Q \; W \; R$
Avoid:	$P \Rightarrow \neg Q$,	$P \Rightarrow \Box \neg Q$,	$P \Rightarrow \neg Q \; W \; R$

Goal *categories* provide a further classification of goals that can be used to guide the acquisition, definition and refinement of goals. The idea is to reuse common goal and goal-refinement techniques found in various application area. As an example, the goal Achieve[ConvenientMeetingPlanned] is an instance of a satisfaction goal because it is concerned with satisfying an agent wish. The category of the goal is declared as follows.

Goal Achieve[ConvenientMeetingHeld]
  InstOf SatisfactionGoal
  ...

Goal categories include the following:

- *Satisfaction goals* are Achieve goals concerned with satisfying agent wishes.
- *Safety goals* are Maintain goals concerned with avoiding hazardous states
- *Security goals* are Maintain goals concerned with avoiding threats to the system
- *Information goals* are Achieve goals concerned with making an agent informed about some states in its environment.
- *Accuracy goals* are Maintain goals concerning the accuracy of the beliefs of an agent about its environment.

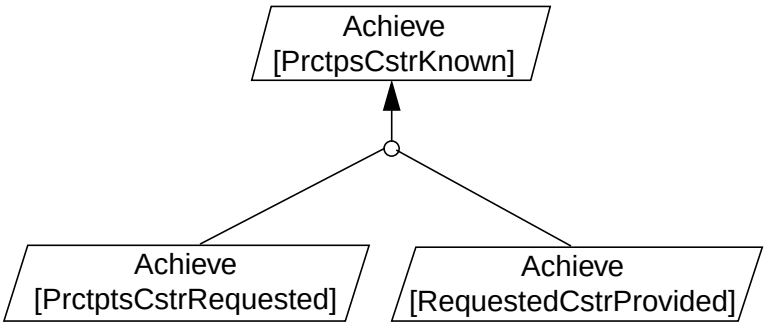
Goal categories are organized into a specialization hierarchy. For example, the category of security goal is specialized into subcategories such as confidentiality goals, availability goals, authentication goals, etc. according to standard classification in the security domain [Amo94]. The declaration of the goal category is optional.

3.2.4.3 Domain Properties

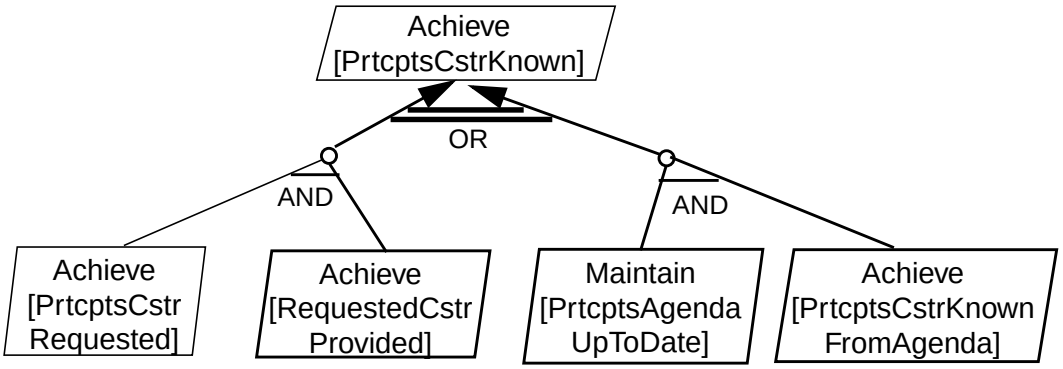
As will be seen below, domain properties play a critical role when refining goals into subgoals. A domain property is a property that is naturally true about the composite system. Physical laws are typical examples of domain properties. An example of domain property for the meeting scheduling problem is the fact that a participant cannot participate simultaneously in two different meetings. Domain properties are declared as domain invariants attached to objects in the object model (see Section 3.3.2.5).

3.2.4.4 Goal Refinement

Goals are related to subgoals through goal refinement links. As an example, the figure below shows that the goal Achieve[PrtcptsCstrKnown] is refined into the two conjoined subgoals Achieve[PrtcptsCstrRequested] and Achieve[RequestedCstrProvided].



A goal can be refined into several *alternative* combinations of subgoals. The figure below shows an alternative refinement for the goal Achieve[PrtcpsCstrKnown], in which participants constraints are retrieved from electronic agendas of participants.



In case of alternative goal refinements, a Selected attribute attached to goal refinement links is used to capture which alternative refinements are actually chosen. (The selection of OR-refinement links is not exclusive; more than one alternative refinement could be selected.)

Semantics of goal refinements

Formally, a set of goals $\{G_1, ..., G_n\}$ refines a goal G in a domain theory Dom if the following conditions hold [Dar95]:

1. $G_1, ..., G_n , Dom \models G$ (completeness)
2. $\bigwedge_{j \neq i} G_j , Dom \not\models G$ for each $i \in [1..n]$ (minimality)
3. $G_1, ..., G_n , Dom \not\models false$ (consistency)

The first condition requires that the satisfaction of the subgoals together with the satisfaction of domain properties in Dom is sufficient for satisfying the parent goal. The second condition requires that if a subgoal is left out of the refinement, the remaining subgoals are not sufficient for satisfying the parent goal. The third condition requires that the conjunction of the subgoals is logically consistent with the domain theory.

The formal definition of goals allows one to verify formally the completeness, minimality and consistency of goal refinements.

Refinement tactics and patterns

Goal refinement tactics and formal refinement patterns have been defined to help the analyst produce complete, minimal and consistent goal refinements [Dar95, Dar96].

Formal goal refinement patterns are generic goal refinements between abstract goal formulations. They are proved correct once and for all. Reusing a refinement pattern entails reusing its proof. Formal refinement patterns are classified according to *refinement tactics*. The latter help selecting goal refinement patterns according to semantic criteria.

Two important goal refinement tactics are worth pointing out: the *milestone-driven* and the *case-driven* goal refinement tactics (see [Dar95, Dar96] for further details).

The *milestone-driven* refinement tactic refines an Achieve goal of the form $C \Rightarrow \Diamond T$ by introducing an intermediate milestone M for reaching a state satisfying the target condition T from a state satisfying the condition C. A Typical goal refinement pattern associated with this tactic is shown in Figure 3.3.

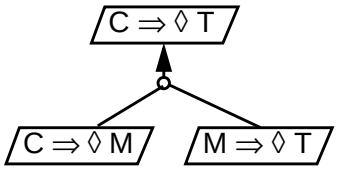


FIGURE 3.3. Milestone-driven goal refinement pattern

As an example of using the pattern in Figure 3, consider the goal Achieve[PrtcptsCstrKnown] defined as follows:

Intended(p,m) ⇒ ◊ CstrKnown(p,m)

The pattern above can be used to refine the goal with the following instantiations:

C : Intended(p,m)
T : CstrKnown(p,m)
M : CstrRequested(p,m)

That is, the goal is refined by introducing the intermediate milestone CstrRequested(p,m). The following subgoals are thereby obtained:

Goal Achieve[PrtcptsCstrRequested]
FormalDef ∀m: Meeting, p: Prtcpt
Intended(p,m) ⇒ ◊ CstrRequested(p,m)

Goal Achieve[RequestedCstrProvided]
FormalDef ∀m: Meeting, p: Prtcpt
CstrRequested(p,m) ⇒ ◊ CstrKnown(p,m)

The *case-driven* goal refinement tactic refines a goal by splitting it into cases. A typical goal refinement pattern associated with this tactic is shown in Figure 3.4.

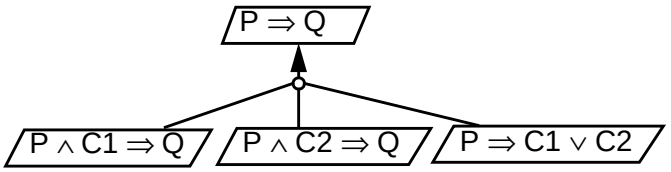


FIGURE 3.4. A case-driven goal refinement pattern

As an example of application of the case-driven goal refinement tactic, consider the goal Achieve[PrtcptsCstrRequested]. If different constraint requests have to be sent out according to the importance of the participant, one can refine the goal by instantiating the case-driven refinement pattern in Figure 3.4 as follows:

P : Intended(p,m)
Q : ◊ CstrRequested(p,m)
C1: Intended[p,m].importance = 'High'
C2: Intended[p,m].importance = 'Low'.

The following subgoals are thereby obtained:

Goal Achieve[ImprtPrtcptsCstrRequested]
FormalDef ∀m: Meeting, p: Prtcpt
Intended(p,m) ∧ Intended[p,m].importance = 'High' ⇒ ◊ CstrRequested(p,m)

Goal Achieve[OtherPrtcptsCstrRequested]
FormalDef ∀m: Meeting, p: Prtcpt
Intended(p,m) ∧ Intended[p,m].importance = 'Low' ⇒ ◊ CstrRequested(p,m)

(Note that these subgoals form an AND-refinement of the parent goal, not an Or-refinement.)

The third assertion resulting from the instantiation of the pattern, that is,

Intended(p,m)
⇒ Intended[p,m].importance = 'High' ∨ Intended[p,m].importance = 'Low'

is a domain property stating that the importance of an intended participant is either high or low.

3.2.4.5 Goal Conflicts

Conflicts between goals can be recorded through instances of the Conflict meta-relationship. Intuitively, a set of goals are conflicting if they cannot be realized together.

As an example, consider a system to control a pump inside a mine [Jos96]. The goals

Maintain[PumpOnWhenHighWater] and Maintain[PumpOffWhenCriticalMethane]

can be formally defined as follows

s.WaterLevel > 'HighWater' ∧ HasPump(s,p) ⇒ p.Status = 'On'

s.MethaneLevel > 'Critical' ∧ HasPump(s,p) ⇒ p.Status = 'Off'

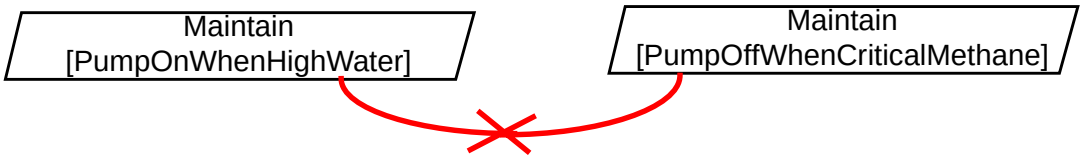


FIGURE 3.5. Goal conflict: an example

These goals cannot always be realized together. Whenever the water level is above the high water level and the methane level is critical, the first goal requires the pump to be on whereas the second one requires the pump to be off. A binary conflict between the two goals can be represented graphically as suggested in Figure 3.5.

Note that the two goals are not logically inconsistent. However, they become inconsistent whenever the following condition holds:

$\exists$ s: Sump, p: Pump
$\Diamond$ s.WaterLevel > 'HighWater' $\wedge$ s.MethaneLevel > 'CriticalMethane' $\wedge$ HasPump(s,p)

That is, the two goals are inconsistent when the water level is above high and the methane level is critical. This assertion is called the boundary condition of the effective conflict. Such potential conflicts are called divergences.

Formally, a set of goals {G1, ..., Gn} is divergent if there exists a *boundary condition* B such that the following conditions hold [Lam98]:

1. $\{B, G_1, ..., G_n, Dom\} \models false$ (logical inconsistency)
2. for every i: $\{B, \bigwedge_{j \neq i} G_j, Dom\} \not\models false$ (minimality)

Further details about goal conflicts and divergences can be found in [Lam98b].

3.2.4.6 Soft Goals and Optimization Goals

In addition to goals that define a set of admissible histories, we also consider soft goals and optimization goals.

Softgoals are goals that do not have a clear-cut criterion for their satisfaction [Myl92, Myl99]. Instead of goal satisfaction, goal *satisficing* is introduced to express that lower-level goals or requirements are expected to contribute to the satisfaction of the goal within acceptable limits rather than absolutely. Soft goals are generally high-level goals that need to be refined into more precise goals. Examples of softgoals are high-level 'non-functional' goals such as usability, flexibility, maintainability, etc.

Optimization goals are softgoals that enable one to compare alternative system designs through minimize/maximize operators. An example of an optimization goal in the meeting scheduling problem is to minimize the interaction among participants to a meeting. Such a goal might be captured by the following specification fragment.

Goal Minimize[ParticipantInteraction]
Definition minimize the number of meeting-related messages sent to participants

Alternative designs for the meeting scheduling problem can be evaluated against such a goal. The alternative in which participants' constraints are known through electronic agendas better contributes to minimizing interaction among participants than the alternative in which the participants' constraints are explicitly requested.

Softgoals and optimization goals can be And/Or refined like any other KAOS goals, conflicts between softgoals/optimization goals can also be captured [Myl92, Myl99]. An important research issue concerns the precise definition of optimization goals, reasoning techniques about soft goals and optimization goals, and the role of such goals in selecting among alternative goal refinements.

In graphical representations of goal models, softgoals and optimization goals are depicted by cloud-like symbols.

3.2.5 The Object Model

3.2.5.1 Objects

An object instance is a “thing” that can be distinctly identified. A domain-level object describes a set of such instances that share some common characteristics. A specific meeting is an example of an object instance. The domain-level object Meeting is a conceptual abstraction characterizing all meeting instances.

For each domain-level object Obj, there is a built-in set member(Obj) that denotes the set of objects instances that are currently members of the domain-level object[1]. The notation InstOf(o,Obj) is also used to denote that the object instance o is currently a member of the domain-level object Obj, i.e.,

InstOf(o,Obj) iff $o \in$ member(Obj)

Each domain-level object has a name and a definition. The *name* of the object is used to identify the object. The *definition* of an object is a natural language statement that should provide a precise interpretation for the set member(Obj), so that one can tell whether or not a particular object instance is currently an instance of the domain-level object.

The set of instances that are currently members of an object may change over time. A given person could be an instance of Student at some time, and could no longer be an instance of that object at some later time.

Note also that objects are not necessarily disjoint. An object instance may be a member of several objects simultaneously. As an example, a given person could simultaneously be an instance of the two different objects Participant and Initiator.

The KAOS object meta-model distinguishes among different kinds of objects. An Object is an Entity, Relationship, Event or Agent, depending on whether the object is autonomous, subordinate, instantaneous or active. Table 3.1 summarizes the different specializations of the Object meta-concept and lists a few examples of objects from the meeting scheduling problem.

Objects are always declared as being of one of the specializations of the Object concept. These specializations are further described below.

1. *The set* member(Obj) *replaces the “Exists” attribute of the original KAOS language described in [Dar93].*

	Definition	Examples
ENTITY	autonomous object	Meeting
AGENT	active object	Initiator, Scheduler, Participant
EVENT	instantaneous object	MeetingRequest
RELATIONSHIP	subordinate object	Intended (links Participant and Meeting) Scheduling (links Scheduler and Meeting)

TABLE 3.1. Specializations of the Object meta-concept

3.2.5.2 Entities

An entity is an autonomous object. The Meeting object is an example of an entity. It might be declared in KAOS as follows

Entity Meeting
Definition An instance of Meeting is any actual meeting that has already been held or any wished meeting (no matter whether it will actually be held or not).

As a specialization of Object, the meta-concept of Entity inherits all features of object; each entity is identified by a *name* and has a *definition* that is used to define which instances are currently a member of the domain-level entity.

3.2.5.3 Events

An event instance is an instantaneous object. An example of event in the meeting scheduling problem is the event MeetingRequest, which may be specified as follows.

Event MeetingRequest
Definition An instance of MeetingRequest occurs when an initiator requests a new meeting to be scheduled.

As any specialization of object, each domain-level event is identified by a *name* and has a *definition* which define its event instances.

An event instance is said to occur at the current time if it is currently an instance of some domain-level event, i.e. we introduce the following notation:

ev.occurs iffInstOf(ev,E) for some domain-level event E.

The instantaneous nature of event is captured by the following built-in domain property which says that an event instance occurs only once in any given history:

forall h: History, ev: EventInstance
if (h, t) |= ev.occurs then for all t' ≠ t (h, t') |≠ ev.occurs

In temporal logic, this property is defined by:

ev.occurs ⇒ ❍ ❑ ¬ ev.occurs ∧ ● ■ ¬ ev.occurs

3.2.5.4 Agents

Agents are active objects, that is, they are capable of performing operations. Such agents may be software agents, hardware devices, or humans. The scheduler is an example of software agent in the meeting scheduling problem. This agent may be declared as follows.

Agent Scheduler
Definition A scheduler is a software agent assisting the meeting initiator for meeting planning.

As a specialization of object, agents are declared by a name and a definition. The particular (meta-) features of agents are that they can perform operations, monitor and control objects, and take responsibility for goals. Such features are described in the sections describing the operation model, the agent interface model, and the agent responsibility model.

3.2.5.5 Relationships

A relationship is a mathematical relation between n objects. For example, the Intended relationship links Participant and Meeting. It is specified as follows.

Relationship Intended
Links Participant **role** is_intended_to **card** 0:N
Meeting **role** has_participant **card** 0:N
Definition the participant is intended to participate at the meeting.

As a specialization of Object, the concept of Relationship has a *name* and a *definition*.

Each relationship also has an ordered set of *links* to specific entities, events or agents. If a relationship Rel has the set of links $\{Obj_1, Obj_2, ..., Obj_n\}$, the set of current instances of the relationship is given by

$$member(Rel) \subseteq \{<ob_1, ..., ob_n> \mid InstOf(ob_i, Obj_i) \text{ for } i \in [1..n]\}$$

The instances of the relationship are those tuples $<ob_1,..., ob_n>$ of object instances which are linked by the relationship. Note that the Obj_i in the above definition may not be distinct. For example, the relationship Brother links the object Person to itself.

The following notation is introduced to denote that a tuple of object instances is an instance of the relationship Rel:

$$Rel(ob_1,..., ob_n) \text{ iff } <ob_1,...., ob_n> \in member(Rel)$$

Each link of a relationship to an object has a role and has cardinality constraints.

The *role* of an object in a relationship is the function that it performs in the relationship.

The *cardinality* of an object in a relationship link is a pair (mincard, maxcard) that denotes the minimum and maximum number of instances of the relationship in which a given object instance can be involved simultaneously, respectively.

Cardinality constraints in the object model can only be used to state domain properties. If a cardinality constraint is a goal rather than a domain property, it must be stated explicitly in the goal model. For instance, consider a train control system, and the property that two trains should never be at the same time on the same track segment. This property defines a goal rather than a domain property. It should not be declared as a cardinality constraint in the object model.

3.2.5.6 Attributes

Objects may have attributes. The date and date range of a meeting are examples of attributes of the Meeting entity. Such attributes are declared in KAOS as follows.

ENTITY Meeting
Definition ...
Has
date_range: SetOf[Date]
the set of dates within which the meeting should take place
date: Date
the planned date of the meeting. If the meeting is not planned, the date is undefined.

Each attribute has a *name*, a *range* of values, and a *definition*. The above attributes are named date_range and date; their range of values are SetOf[Date] and Date, respectively.

The semantics of an attribute Attr declared for an object O with a range D is a function

Attr: member(O) $\rightarrow$ D.

The function defined by an attribute is not necessarily total. In the example above, the date attribute of the Meeting entity is a partial function. A meeting that is not yet planned does not have a date.

The definition of an attribute is a natural language description of the meaning of the attribute.

Note that relationships may also have attributes. As an example, the importance of a participant for a meeting can be declared as follows:

Relationship Intended
Links Participant **role** is_intended_to **card** 0:N
Meeting **role** has_participant **card** 0:N
Definition Intended(p,m) iff participant p is intended to participate at meeting m.
Has
importance: {High, Low}

3.2.5.7 Specialization

Inheritance between objects can be declared through instances of the IsA meta-relationship. As an example, the Participant agent can be declared as a specialization of the Person agent.

The semantics of IsA links between objects is membership inclusion, i.e.

If IsA(Obj1, Obj2) then member(Obj1) $\subseteq$ member(Obj2)

Inheritance of object features is a consequence of this definition. That is, attributes declared for Obj2 are defined on instances of Obj1 as well; and relationships linking Obj2 link instances of Obj1 as well.

3.2.5.8 Invariants

Domain properties that are naturally true about objects can be declared as domain invariants.

As an example, consider the Meeting entity specified as follows:

ENTITY Meeting
Definition ...
Has
date: Date
the planned date of the meeting. If the meeting is not planned, the date is undefined.
planned: Bool
a meeting is planned if there exists a date for the meeting
DomInvar
m.planned ⇔ (∃ d: Date) m.date = d

The above invariant formally defines what a planned meeting is in the domain.

Cardinality constraints are short-cuts for specifying a restricted class of domain invariants.

3.2.5.9 Consistency rules between the object and goal model

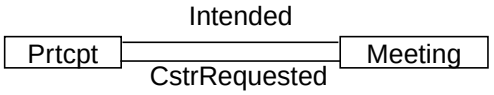
As mentioned before, there is a meta-constraint relating the goal and object models:

Every vocabulary element used in the formal definition of goals must be declared in the object model.

Consider for instance the goal Achieve[PrtcptsCstrRequested] defined above as follows:

∀m: Meeting, p: Prtcpt
Intended(p,m) ⇒ ◊ CstrRequested(p,m)

The formal definition references the sorts Prtcpt and Meeting, and the predicates Intended and CstrRequested. In order to satisfy the meta-constraint between the goal and object models, the following portion of the object model is derived from the formal definition of the goal:



In the KAOS meta-model, a Concern meta-relationship linking Goal to Object is used to keep track of the objects concerned by the goal.

3.2.6 The Agent Responsibility Model

The agent responsibility model is used to declare responsibility assignments of goals to agents. Responsibility assignments provide a criterion for stopping the goal refinement process. A goal assigned as the responsibility of a single agent must not be refined further.

3.2.6.1 Responsibility Links

Responsibility is an OR meta-relationship that links agents to goals. Responsibility links declare potential responsibility assignment of goals to agents.

Examples of responsibility assignment are shown in Figure 3.6. In the figure, the goal Achieve[PrtcptsCstrRequested] is potentially assigned to the Initiator agent *or* to the Scheduler agent. The goal Achieve[RequestedCstrProvided] is potentially assigned as the responsibility of the Participant agent.

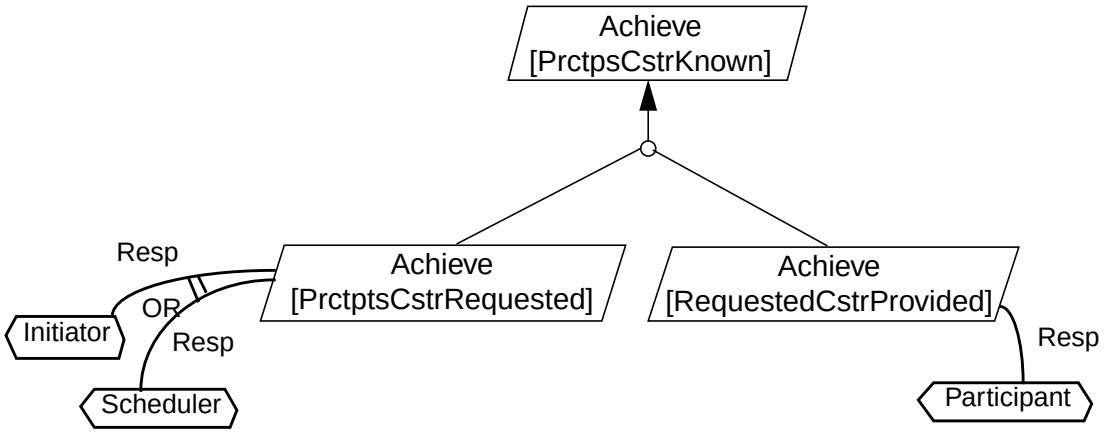


FIGURE 3.6. Responsibility assignments

Responsibility links have a *Selected* attribute used to indicate which alternative responsibility assignments are actually chosen.

A goal effectively assigned to an agent in the software-to-be is then called a Requirement, whereas a goal effectively assigned to an agent in the environment is called an Assumption.

3.2.6.2 Instance declarations

At the outer-level, responsibility links declare the agent class responsible for the goal. Instance declarations attached to responsibility links specify more precisely which instance of the agent class is responsible for the goal instantiated to specific object instances. The following example illustrates the need for such instance declarations. Consider an air traffic system, and the goal Maintain[PlaneInRegionOnCourse] whose definition is given by

$\forall$ pl: Plane, atc: AirTrafficController
InRegion(pl, atc) $\Rightarrow$ pl.OnCourse

Suppose that the goal can be assigned as the responsibility of the air traffic controller agent. Such responsibility assignment could be declared as follows:

Responsibility [AirTrafficController, PlaneInRegionOnCourse]
Instance Declaration ∀ atc: AirTrafficController, pl: Plane
Responsibility [atc, InRegion(pl, atc) ⇒ pl.OnCourse]

The outer level declaration declares that agents that are instances of AirTrafficController are responsible for maintaining planes on course. The instance declaration defines more precisely which instance of air traffic controller is responsible for which plane, that is, each instance of air traffic controller is responsible for maintaining on course only those planes which are in the region it controls.

The *Responsibility* operator relates a term denoting an agent instance to a temporal formula. (This does not introduce second-order into the language.) It is a time-independent declaration of responsibility assignment at the instance level. It can only be universally quantified, i.e. instance declarations attached to responsibility links must have the following form:

(∀ ag: Agent, ob_1: Obj_1, ..., ob_n: Obj_n): *Responsibility*(ag, G(ag, ob_1, ..., ob_n)).

(The *Responsibility* operator can only be used to define instance-level responsibility assignments attached to Responsibility links; it cannot be used in the formal definition of goals.)

Note that even though the *Responsibility* operator is time-independent, it enables one to declare time-dependent responsibility assignments. In the example above, an air traffic controller is responsible for maintaining a plane on course while the plane is in its region. When the plane leaves its region, the air traffic controller is no longer responsible for maintaining that plane on course.

3.2.6.3 Semantics of Responsibility

The (intuitive) meaning of a responsibility assignment of a goal to an agent is that the agent responsible for a goal is the only one required to restrict its behaviour so as to ensure the goal [Fea87].

As an example, consider the goal Achieve[PrtcptsCstrRequested] assigned to the Scheduler agent. This declaration of responsibility means that the Scheduler must restrict its behaviour so as to ensure this goal, no matter what the behaviours of other agents are.

The meaning of responsibility assignments is made explicit through *operationalization links* that relate goals to requirements on operations and through the *responsibility meta-constraint* on such links. Intuitively, a set of requirements on the application of operations operationalize a goal if the temporal behaviours produced by such requirements is included in the set of histories admitted by the goal; the responsibility meta-constraint requires that only operations assigned to the agent responsible for a goal are constrained so as to operationalize that goal. Operationalization links and the responsibility meta-constraint are further discussed in Sections 3.2.7.5 and 3.2.7.6.

3.2.7 The Operation Model

An *operation* is an input-output relation over objects; operation applications define state transitions. Operations are characterized by pre-, post-, and trigger conditions. A distinction is made between *domain* pre/postconditions, which capture the elementary state transitions defined by operation applications in the domain, and *required* pre/trigger/postconditions, which capture additional strengthenings to ensure that the goals are met.

As an example, the operation PlanMeeting in the meeting scheduling problem may be defined as follows.

Operation PlanMeeting
Input Meeting {**arg** m}
Intended
Constraint
Output Meeting /{Planned, Date}
DomPre ¬ m. Planned
DomPost m.Planned
ReqPostFor Maintain[ConvenientMeetingDate]
(∀ p: Participant): ● Intended(p,m) → m.Date ∉ ● Constraint[p,m].exclset

The operation above is named PlanMeeting. Its domain pre and post condition capture elementary state transitions corresponding to the operation, namely, from a state where the meeting is not planned to a state where the meeting is planned. The operation has a required post condition which states that the meeting date must be outside the exclusion set of all intended participants. This required post condition operationalizes the goal Maintain[ConvenientMeetingDate]. The Input/Output clauses provide signature information (in particular, the output of PlanMeeting is restricted to two specific attributes of the Meeting object).

3.2.7.1 Domain Pre/Post and Required Pre/Trigger/Post Conditions

In addition to its name, each operation is characterized by the following features[1]:

- A *domain precondition* characterizing the states before any application of the operation;
- A *domain postcondition* defining a relation between states before and after applications of the operation;
- *Required preconditions* defining those states in which the operation is allowed to be applied;
- *Required trigger conditions* defining those states in which the operation is obliged to be immediately applied provided the domain precondition is true;
- R*equired post conditions* defining additional conditions that applications of the operation must satisfy.

Required pre, trigger and post conditions are related to goals through *operationalization* links further described below (Section 3.2.7.5).

---

1. This description of the operation model is slightly different from the one in [Dar93] where required pre-, trigger, and postconditions are defined as attributes of a meta-relationship relating operations to goals.

Operations denote atomic state transitions. In case where one needs to model an activity that lasts over several states, it is necessary to model one atomic operation that starts the activity, one atomic operation that ends the activity (or several operations if there are different ways in which the activity can terminate), and one additional state variable to indicate that the activity currently holds.

Several operations can occur concurrently. This *non*-interleaving semantics is mostly motivated by the semantics of trigger conditions as immediate obligations. With an interleaving semantics, an operation model would be inconsistent when the trigger conditions of two (or more) operations are true at the same time.

A formal semantics for the operation model is defined in chapter 7.

Note the difference between *required precondition* and *required trigger condition*. Required preconditions define permissions to apply the operation. If the operation is applied, the required preconditions of the operation must be true. Required trigger conditions define immediate obligations to perform the operation. If a trigger condition is true, the operation must be applied (provided that the domain precondition is true).

A meta-constraint of the language is that required trigger conditions must imply required preconditions, that is,

$$\mathsf{ReqTrig} \Rightarrow \mathsf{ReqPre}.$$

Violations of that constraint yield inconsistent operation models. If there is a state in which a required trigger condition is true and a required precondition is false, the operation must be applied immediately whereas it may not be applied at the same time.

3.2.7.2 Inputs and Outputs

Operations are related to objects through Input/Output links. An object is among the inputs (resp. outputs) of an operation if it is among the sorts making up the domain (resp. codomain) of the relation defined by the operation.

For the PlanMeeting operation above, the objects Meeting, Intended and Constraint are declared as Input of the operation, whereas the object Meeting is declared as Output of the operation.

The Input and Output meta-relationships have optional *Argument* and *Result* meta-attributes, respectively, to declare instances of variables referenced in the assertions attached to the corresponding operation.

Input and Output links may also declare more precisely which object attributes make up the domain and codomain of the operation. In the example above, the Planned and Date attributes of the Meeting entity are declared as outputs of the operation. Attributes declared as outputs of an operation are the only ones whose value can be changed by an application of the operation.

3.2.7.3 Initial Conditions

One can also define initial conditions that must be satisfied at the initial state of every history. Initial conditions can be declared as domain properties or as goals. Goals constraining the initial state of the system are also to be assigned as the responsibility of single agents.

3.2.7.4 Performance Links

An operation is related to the agent that can initiate it through a Performance link. Performing is an OR meta-relationship linking agents to operations.

As an example, the operation PlanMeeting could be alternatively assigned to the Scheduler agent or to the Initiator agent.

As an OR meta-relationship, it has a Selected attribute used to denote which agent is effectively assigned to which operation in the selected alternative.

Instance declarations attached to Performance links are used to declare specific assignments of operations to specific agents at the instance level. For example, the assignment of the operation PlanMeeting to the Scheduler agent can be declared as follows:

Performance [Scheduler, PlanMeeting]
InstDecl ∀ sch: Scheduler, m: Meeting
Scheduling(sch, m) ⇒ *Performance*(sch, PlanMeeting(m))

The *Performance* operator relates an agent instance to an operation instance (i.e. an operation with its arguments and results instantiated to instances of input and output objects). Instance declarations for Performance links have the form:

R(ag, ob1, ..., obn) ⇒ *Performance*(ag, op(ob1, ..., obn)

where R is a domain-specific predicate relating a term denoting an agent instance to instantiation of the arguments and result of the operation.

"Unique performer" meta-constraint

In some given alternative, an operation is associated with exactly one agent. In the metamodel, this is captured by defining the cardinality constraint of the Operation concept in the Performance relationship to be (1:1). At the instance level, the predicate R must satisfy the following constraint:

R(ag, ob1, ..., obn) ∧ R(ag', ob1, ..., obn) ⇒ ag = ag'

3.2.7.5 Operationalization Links

Required pre-, trigger, and postconditions are related to goals through operationalization links.

The Operationalization meta-relationship is an AND/OR relationship between goals and required pre, trigger, and post conditions of operations. Intuitively, a set of required pre, trigger, and post conditions operationalizes a goal if satisfying the required conditions on operations guarantees that the goal is satisfied. A formal semantics of operationalization links is defined in Chapter 7.

As an example, the above required post condition on the operation PlanMeeting is introduced to ensure the goal Maintain[ConvenientMeetingDate].

As another example, consider a mine pump control system and the goal Maintain[Pump SwitchOnWhenHighWaterDetected]. This goal defined as follows:

∀ c: PumpController
● c.HighWaterSignal = 'On' ⇒ c.PumpSwitch = 'On'

An operationalization of that goal is given by the following required trigger and preconditions on the operations SwitchPumpOn and SwitchPumpOff:

Operation SwitchPumpOn
Input PumpController {**arg** c}/ HighWaterSignal
Output PumpController {**res** c} / PumpSwitch
DomPre c.PumpSwitch = 'Off'
DomPost c.PumpSwitch = 'On'
ReqTrigFor Maintain[PumpSwitchOnWhenHighWaterDetected]
c.HighWaterSignal = 'On'

Operation SwitchPumpOff
Input PumpController {**arg** c}/ HighWaterSignal
Output PumpController {**res** c} / PumpSwitch
DomPre c.PumpSwitch = 'On'
DomPost c.PumpSwitch = 'Off'
ReqPreFor Maintain[PumpSwitchOnWhenHighWaterDetected]
¬ c.HighWaterSignal = 'On'

The required trigger condition on the SwitchPumpOn operation requires that the pump *must be* switched on when the HighWaterSignal is On; the required precondition on the SwitchPumpOff operation requires that the pump *may be* switched off only if the HighWaterSignal is not On.

The formal definition of operationalization links defined in Chapter 7 can be used to show that those requirements constitute a complete and minimal operationalization of the goal.

3.2.7.6 The responsibility meta-constraint

As mentioned in Section 3.3.2.6.3, the semantics of assigning responsibility for a goal to an agent is that only the agent responsible for the goal should restrict its behaviour so as to ensure the goal [Fea87]. This is captured in the meta-model by the following responsibility meta-constraint on operationalization links. It requires that the goal be operationalized by strengthening only the operations performed by the responsible agent:

if Responsibility(Ag, G) and Operationalization({P_1, ..., P_n}, G)
then Performance(Ag, Operation(P_i)) for all n =[1..n]

where Operation(P_i) denotes the operation constrained by the requirement P_i.

As an example, if the goal Maintain[ConvenientMeetingDate] is assigned as the responsibility of the Scheduler agent, the operation PlanMeeting whose required postcondition operationalize the goal must be assigned to the same Scheduler agent.

Similarly, if the goal Maintain[PumpSwitchOnWhenHighWaterDetected] is assigned as the responsibility of the PumpController agent, the operations SwitchPumpOn and SwitchPumpOff whose required trigger and pre conditions operationalize that goal must be performed by the PumpController agent.

3.2.8 The Agent Interface Model

The agent interface model is used to declare the quantities monitored and controlled by each agent. Declaring which quantities are monitored and controlled by which agents is important as the requirements assigned to an agent must be defined in terms of quantities monitored and controlled by the agent [Fea87, Par95, Jac95, Zav97]. In the KAOS framework, such quantities correspond to object attributes and links.

3.2.8.1 Monitoring and Control Links

Agent interfaces are declared through Monitoring and Control links between agents and objects' attributes.

The meaning of a Monitoring link between an agent and an object attribute is that the agent directly monitors ("reads") the value of the attribute.

As an example, the Scheduler software agent does not directly monitor the actual date constraints of the intended participants of a meeting; but it monitors the messages about constraints that it receives from participants.

The meaning of a Control link between an agent and an object attribute is that the agent directly controls ("writes") the value of the attribute. In other words, an agent controls an attribute if it is capable of controlling state transitions for that attribute. We also consider that attributes controlled by an agent are observable by that agent as well.

As an example, the Scheduler agent does not control the presence of participants at a meeting, but it could be declared to control the planned date of meetings.

In addition to monitored and controlled variables, we also consider variables that are internal to an agent. An internal attribute of an agent is an attribute controlled by that agent and monitorable by no other agents. (The motivation for having internal variables in addition to interface variables is discussed in Section 6.5 of Chapter 6.)

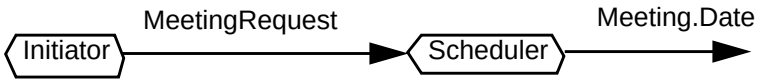
To simplify the description of the model, we use the syntactic convention that Monitoring (resp. Control) links can also be declared between an agent and an object when the agent monitors (resp. controls) all attributes of the object.

For instance, the declarations below state that the Scheduler agent monitors the MeetingRequest event controlled by the Initiator agent; and the Scheduler agent controls the date of the Meeting object.

Monitors [Scheduler, MeetingRequest]
Control [Initiator, MeetingRequest]
Control [Scheduler, Meeting.Date]

Graphically, agent interfaces are represented by context diagrams similar to the ones used in Structured Analysis [Dem78]. Monitoring and Control links are represented by arrows labelled with attributes (or objects). An arrow leaving an agent means that the agent controls the attributes attached to the arrow; an arrow pointing to an agent means that the agent monitors the attributes attached to the arrow. Internal variables are not represented on the graphical view of the model.

The Monitoring and Control links in the example above will be represented graphically as follows.



Monitoring and Control are Or meta-relationships. Alternative agent interface models can be represented. In the graphical view, alternative agent interface models are represented by separate, alternative agent interface diagrams.

3.2.8.2 Instance declarations

Instances declarations attached to Monitoring and Control links enable one to describe more precisely which agent instance monitors or controls the values of the attribute for which object instance.

For example, the instance declaration attached to the following Control link declares that the scheduler instance sch scheduling the meeting m controls the date of that meting.

Control [Scheduler, Meeting.Date]
InstanceDeclaration ($\forall$ sch: Scheduler, m: Meeting):
Scheduling(sch,m) $\Rightarrow$ *Ctrl*(sch, m.Date)

The *Mon* and *Ctrl* operators relate an agent instance to an attribute of an object instance. Instance declarations for Monitoring and Control links have the form

R(ag, o) $\Rightarrow$ *Mon*(ag, o.Attr)
R(ag, o) $\Rightarrow$ *Ctrl*(ag, o.Attr)

where R is a domain-specific predicate between ag and o.

3.2.8.3 The "unique control" meta-constraint

The Control meta-relationship must satisfy the *unique control constraint* requiring that in each alternative an object's attribute is controlled by at most one agent.

At the outer-level of the language, the unique control constraint is defined as follows:

if Control(ag, Ob.Attr) then there is no ag' $\neq$ ag such that Control(ag', Ob.Attr).

At instance-level, the unique control constraint requires that the predicate R involved in an instance declaration of the form

R(ag, o) $\Rightarrow$ *Ctrl*(ag, o.Attr)

satisfies the following condition:

R(ag, o) $\Rightarrow \neg$ ($\exists$ ag') ag' $\neq$ ag $\wedge$ R(ag', o).

The unique control constraint is used to avoid interference problems between concurrent executions of agents.

The unique control constraint is an important and necessary feature of the language. It is necessary to be able to assign goals as the responsibility of single agents. As an example, consider a goal requiring that a predicate P never holds:

❑ $\neg$ P

If the predicate P is controlled both by the agents ag1 and ag2, none of these agents can alone guarantee the goal: if the goal is assigned as the responsibility of the unique agent ag1, nothing prevents the other agent from making the predicate P true, thereby violating the goal.

The unique control constraint sometimes requires a single variable controlled by two agents to be split into two variables, each controlled by a single agent.

3.2.8.4 The input/output meta-constraint

In a given alternative, the Monitoring and Control links of an agent are related to the Input and Output links of the operations performed by the agent. This is captured by the following *input/output meta-constraint* of the language:

(i) if Input(Ob.Attr, Op) and Performance(Ag, Op) then Monitoring(Ag, Ob.Attr) or Control(ag, Ob.Attr)
(ii) if Output(Ob.Attr, Op) and Performance(Ag, Op) then Control(Ag, Ob.Attr)

These constraints capture the facts that every attribute which is part of the input (resp. output) of the operation must be monitored (resp. controlled) by the agent performing the operation.

3.2.8.5 The realizability meta-constraint

The monitoring and control capabilities of agents constrain the possible responsibility assignments of goals to agents: for a goal to be assignable to an agent, it must be defined in terms of variables monitored and controlled by the agent [Fea87, Par95, Jac95, Zav97].

As an example, consider the meeting scheduling problem and the ideal goal Achieve[ConvenientMeetingHeld] requiring that requested meetings are eventually held with the presence of all intended participants. That goal is clearly not assignable to the Scheduler software agent, because it constrains the actual occurrence of meetings and the presence of participants at meetings; these are not controllable by the Scheduler agent.

As another example, consider the goal Achieve[ConvenientMeetingPlanned] requiring to find a date for the meeting that satisfies the constraints of the participants. This goal is also not assignable by the Scheduler software agent because the scheduler has no monitoring capabilities on the actual date constraints for participants to the meeting.

Consider on the other hand the goal Achieve[PrtcptsCstrRequested] requiring that a constraint request is sent to every participant intended for the meeting. That goal is assignable to the Scheduler agent provided it controls the sending of constraint requests to participants and it monitors which participants are intended for the meeting.

The *realizability meta-constraint* captures this relation between agent responsibilities and agent interfaces. A goal is said to be *realizable* by an agent if an operationalization of the goal that satisfies the responsibility and input/output meta-constraints can be found[1]. The realizability meta-constraint requires that a goal assigned to an agent is realizable by that agent. A formal definition of this meta-constraint is given in Chapter 4.

The realizability meta-constraint plays a significant role in the goal refinement process. It defines what is meant for a goal to be assignable to an agent, and therefore provides a precise criterion for stopping the goal refinement process. Violations of that constraint drive the elaboration of the goal and agent models; a goal that is not realizable by a single agent has to be further refined so as to eventually reach subgoals that are realizable by single agents. The role of this meta-constraint in the goal refinement process is further described in Section 3.3.5 below.

3.3. The Goal-Oriented Requirements Elaboration Method

This section discusses the KAOS goal-oriented method. It first gives a brief overview of the method, then further describes the various steps of the method and illustrates their application on a small case study, the mine pump control system. The important role of the realizability meta-constraint in this goal-oriented process is then discussed in Section 3.3.5.

3.3.1 Overview

The KAOS goal-oriented requirements elaboration method consists in deriving the requirements for a future system from high-level goals. It consists in the following activities (Figure 3.7).

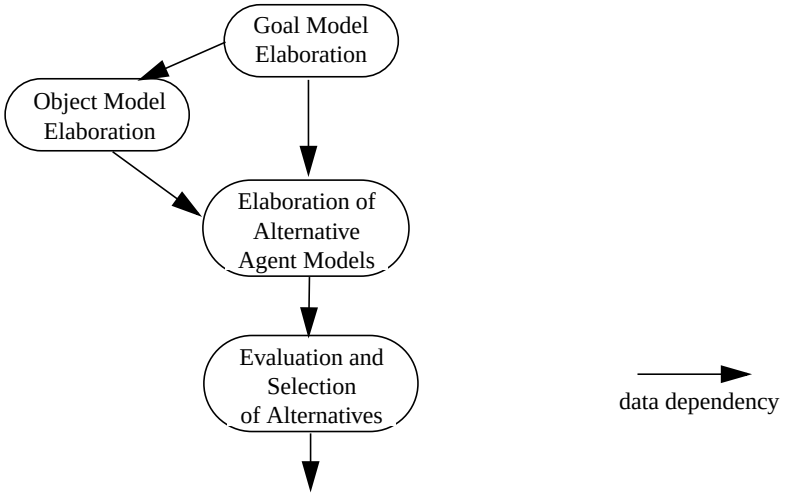


FIGURE 3.7. The goal-driven requirement elaboration process

The activities are ordered by data dependencies; they are running concurrently, with much intertwining between them. In particular, there is significant intertwining between the elaboration of the goal model and the elaboration of alternative agent models. This intertwining is further discussed in Section 3.3.5.

---

1. This concept of realizable goal is the equivalent at the goal level of the concept of realizable specification at the design level: a specification is realizable if there exists a program that implements it [Aba89].

1. Elaborating the goal model and identifying objects. The first activity consists in identifying goals and elaborating the goal refinement graph. Preliminary goals are identified from various sources available (documentation, high-level objective statements, interviews, scenarios, and so forth). Further goals are identified by a combination of bottom-up and top-down processes. Subgoals are identified by asking HOW questions about the goals already identified; parent goals are identified by asking WHY questions. The identified goals are given formal definitions from which the objects concerned by the goals are identified.

In summary, the activities of elaborating a goal model and identifying the objects concerned by the goals consist of the following activities:

1.1 identifying preliminary goals
1.2 formalizing goals and identifying objects
1.3 eliciting further goals through WHY questions
1.4. eliciting further goals through HOW questions

2. Exploring Alternative Agent models. Alternative agent responsibilities, agent interfaces and operation models are gradually elaborated from the goal model. Potential responsibility assignments of goals to agents are identified during the goal refinement process. Agent interfaces are derived form responsibility assignments of goals to agents. Operations relevant to the goals and requirements on operations that guarantee the satisfaction of the goals are identified during the operationalization step. The following steps are thus performed for elaborating alternative agent models from goals:

2.1. Identifying potential responsibility assignments
2.2. Deriving agent interfaces from responsibility assignments
2.3. Operationalizing goals

3. Evaluation and selection of Alternatives. This activity consists in making choices among alternative goal refinements and responsibility assignments. Such choices are based on non-functional or softgoals [Myl92] such as reduce costs, minimize risks, avoid agents' overloading, maximize flexibility, etc. Relevant techniques for such an evaluation and selection process are described in [Fea91], [Rob90] and [Myl99].

3.3.2 The Mine Pump Example

The following problem statement for a mine pump is used as a running example to illustrate the KAOS goal-oriented requirements elaboration method. The problem statement, taken from [Jos96], is reproduced below.

> Water percolating into a mine is collected in a sump to be pumped out of the mine (see. Figure 3.8). The water level sensors *D* and *E* detect when water is above a high and below a low level, respectively. A pump controller switches the pump on when the water reaches the high water level and off when it goes below the low water level. If, due to a failure of the pump, the water cannot be pumped out, the mine must be evacuated within one hour.
>
> The mine has other sensors (*A, B, C*) to monitor the carbon monoxide, methane and airflow levels. An alarm must be raised and the operator informed within one second of any of these levels becoming critical so that the mine can be evacuated within one hour. To avoid the risk of explosion, the pump must be operated only when the methane level is below a critical level.

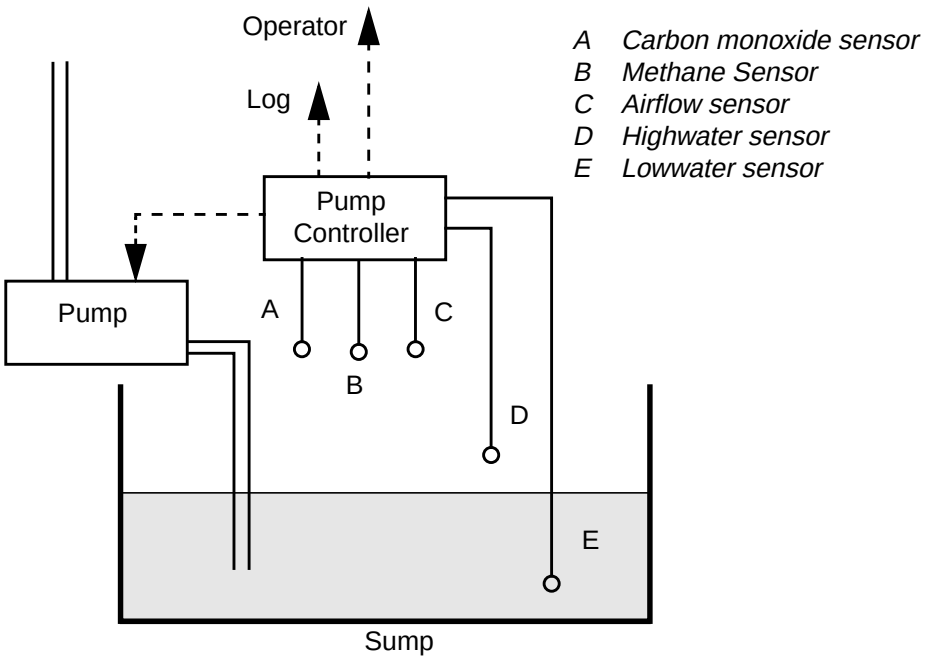


FIGURE 3.8. Mine pump and control system

Human operators can also control the operation of the pump, but within limits. An operator can switch the pump on or off if the water is between the low and high water levels. A special operator, the supervisor, can switch the pump on or off without this restriction. In all cases, the methane level must be below its critical level if the pump is to be operated.

Readings from all sensors, and a record of the operation of the pump, must be logged for later analysis.

3.3.3 Elaborating the goal and object models

3.3.3.1 Identifying preliminary goals

The first step of the elaboration method consists in identifying preliminary goals from various sources: interviews, analysis of available documentation to find out problematic issues with the existing system, objectives that are explicitly stated about the envisioned one, usage scenarios, operational choices whose rationale has to be elicited, etc.

Mine Pump Examples

Figure 3.9 gives the preliminary goals identified from the initial statement of the mine pump case-study. These goals correspond to properties explicitly stated in the initial problem statement. The goals are listed in the order in which they appear in the initial problem statement.

At this stage, elicited goals are given a name and a tentative natural language definition. The temporal pattern (Achieve/Maintain) of the goal is also identified. Goal names and definitions will gradually be made more precise as the specification evolves and more knowledge is gained about the system.

Goal Maintain[PumpOnWhenHighWater]
Definition The pump must be On when the water level in the sump is above the high water level.

Goal Maintain[PumpOffWhenLowWater]
Definition The pump must be Off when the water level in the sump is below the low water level.

Goal Achieve[MineEvacuatedWhenPumpFailure]
Definition If, due to a failure of the pump, the water cannot be pumped out, the mine must be evacuated within one hour.

Goal Achieve[AlarmRaisedWhenCriticalGazLevel]
Definition If the carbon monoxide level, methane level or airflow level becomes critical, an alarm must be raised and the operator must be informed within one second.
Refines Achieve[MineEvacuatedWhenCriticalGazLevel]

Goal Achieve[MineEvacuatedWhenCriticalGazLevel]
Definition If the carbon monoxide level, methane level or airflow level becomes critical, the mine must be evacuated within one hour.

Goal Maintain[PumpOffWhenCriticalMethane]
Definition The pump must be off when the methane level is above a critical level.
Refines Avoid[Explosion]

Goal Avoid[Explosion]
Definition No explosion should occur in the mine.

Goal Maintain[OperationsLogged]
Definition Readings from all sensors, and a record of the operation of the pump, must be logged for later analysis

FIGURE 3.9. Preliminary goals for the mine pump

Only a portion of the goal graph is elicited. Further goals will be identified by asking WHY and HOW questions.

3.3.3.2 Formalizing Goals and Identifying Objects

The next step in the specification elaboration process is to formalize goals and derive the objects concerned by the goals. This step consists of the following intertwined substeps:

(i) *Formalize goals*. The informal definitions of goals are translated into formal definitions. The formalization of goals is guided by the pattern and category of the goal. The pattern of a goal drives the choice of a pattern of temporal formula. The category of the goal is intended to further guide the formalization of the goal by providing generic formalizations of goals specific to the goal category.

(ii) *Derive objects from goal definition*. The definition of goals drives the identification of objects and attributes to be declared in the object model. Through application of the consistency rule between the goal model and object model, the vocabulary used in the formal definition of goals gives rise to the declaration of objects and attributes in the object model.

(iii) *Define objects and attributes*. The identified objects and attributes must be defined by relating them to the real-world quantities they denote [Par95, Zav97].

Mine Pump Examples

Consider the goal Maintain[PumpOnWhenHighWater]. The Maintain pattern of the goal suggests using one of the formal patterns for Maintain goals (see Section 3.2.4.2). Since the goal requires a property to hold globally, the formal definition of the goal has the general form

$$P \Rightarrow Q,$$

where P and Q are state formulae. The abstract definition pattern is then instantiated to yield the following definition:

$$\forall\ s: \text{Sump}, p: \text{Pump}:$$
$$s.\text{WaterLevel} \geq \text{`HighWater'} \wedge \text{HasPump}(s,p) \Rightarrow p.\text{Motor} = \text{`On'}$$

The vocabulary appearing in the definition of the goal gives rise to the declaration of objects and attributes shown in Figure 3.10. The formal definition references the sorts Sump and Pump, which drives the declaration of the corresponding entities in the object model. The HasPump predicate drives the declaration of the corresponding relationship between Sump and Pump. The sump's water level and the status of the pump motor are declared as attributes of the corresponding entities.

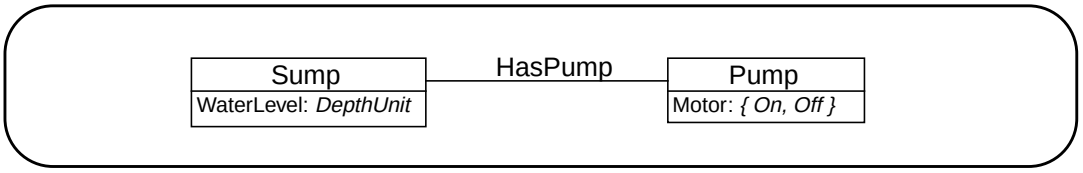


FIGURE 3.10. Preliminary object model derived from the goal Maintain[PumpOnWhenHighWater]

In addition to declaring the vocabulary relevant to the goals of the system, the object model should also include precise the interpretation for the vocabulary used. For instance, the entities Sump and Pump could be defined as follows:

Entity Sump
Definition Bottom portion of a mine, into which water percolating into the mine is collected in order to be pumped.
Has
WaterLevel: *DepthUnit*
the actual water level in the sump

Entity Pump
Definition Device used to pump water out of the mine.
Has
Motor: *{On, Off}*
the status of the motor of the pump

The formalization of the other goals in Figure 3.9 gives rise to further elaboration of the object model. Consider the goal Achieve[MineEvacuatedWhenPumpFailure]. The Achieve pattern of the goal suggests defining the goal with one of the formal definition patterns for Achieve goals. Since the goal requires a property to eventually hold within some real-time bound, the following pattern of goal definition is chosen:

$$P \Rightarrow \Diamond_{\leq d} Q$$

The goal is then formally defined as follows:

$$\forall p: Pump, m: Mine$$
$$p.Failure \wedge PumpInMine(p,m) \Rightarrow \Diamond_{\leq 1h} \neg (\exists miner: Miner): InsideMine(miner,m)$$

The initial portion of the object model is now enriched with the vocabulary used in this definition (Figure 11).

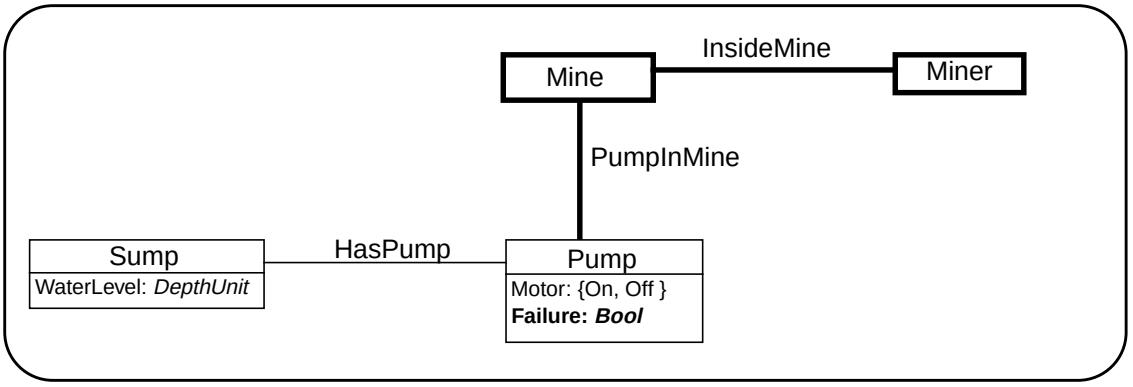


FIGURE 3.11. Object model derived form the goal Achieve[MineEvacuatedWhenPumpFailure]

The formalization of other goals gives rise to further elaboration of the object model. The formal definition of all the preliminary goals is given in Figure 3.12. The derived object model is shown in Fig. 3.13.

The activities of formalizing goals and deriving the corresponding fragments of the object model are performed continuously during the requirement elaboration process as new goals are being identified.

Goal Maintain[PumpOnWhenHighWater]
Definition The pump must be On when the water level in the sump is above the high water level.
FormalDef $\forall$ s: Sump, p: Pump:
s.WaterLevel $\geq$ 'HighWater' $\wedge$ HasPump(s,p) $\Rightarrow$ p.Motor = 'On'

Goal Maintain[PumpOffWhenLowWater]
Definition The pump must be Off when the water level in the sump is below the low water level.
FormalDef $\forall$ s: Sump, p: Pump:
s.WaterLevel $\leq$ 'LowWater' $\wedge$ HasPump(s,p) $\Rightarrow$ p.Motor = 'Off'

Goal Achieve[MineEvacuatedWhenPumpFailure]
Definition If, due to a failure of the pump, the water cannot be pumped out, the mine must be evacuated within one hour.
FormalDef $\forall$ p: Pump, m: Mine
p.Failure $\wedge$ PumpInMine(p,m) $\Rightarrow \Diamond_{\leq 1h} \neg$ ($\exists$ miner: Miner): InsideMine(miner,m)

Goal Achieve[AlarmRaisedWhenCriticalGazLevel]
Definition If the carbon monoxide level, methane level or airflow level becomes critical, an alarm must be raised and the operator must be informed within one second.
FormalDef $\forall$ m: Mine
(m.MethaneLevel $\geq$ 'CriticalMethane'
$\vee$ m.COLevel $\geq$ 'COCritical'
$\vee$ m.AirFlow $\leq$ 'CrticalAirflow')
$\Rightarrow \Diamond_{\leq 1sec}$
m.Alarm = 'On'
$\wedge$ ($\forall$ op: Operator): OperatingMine(op,m) $\rightarrow$ op.Informed

Goal Achieve[MineEvacuatedWhenCriticalGazLevel]
Definition If the carbon monoxide level, methane level or airflow level becomes critical, the mine must be evacuated within one hour.
FormalDef $\forall$ m: Mine
(m.MethaneLevel $\geq$ 'CriticalMethane'
$\vee$ m.COLevel $\geq$ 'COCritical'
$\vee$ m.AirFlow $\leq$ 'CrticalAirflow')
$\Rightarrow \Diamond_{\leq 1h} \neg$ ($\exists$ p: Miner): InsideMine(p,m)

Goal Maintain[PumpOffWhenCriticalMethane]
Definition The pump must be off when the methane level is above a critical level.
FormalDef $\forall$ s: Sump, p: Pump:
s.MethaneLevel $\leq$ 'CriticalMethane' $\wedge$ HasPump(s,p) $\Rightarrow$ p.Motor = 'Off'

Goal Avoid[Explosion]
Definition No explosion should occur in the mine.
FormalDef ($\forall$ m:Mine): $\Box \neg$ m.explosion

Goal Maintain[OperationsLogged]
Definition Readings from all sensors, and a record of the operation of the pump, must be logged for later analysis
FormalDef <not specified at this stage>

FIGURE 3.12. Formal definitions for the preliminary goals

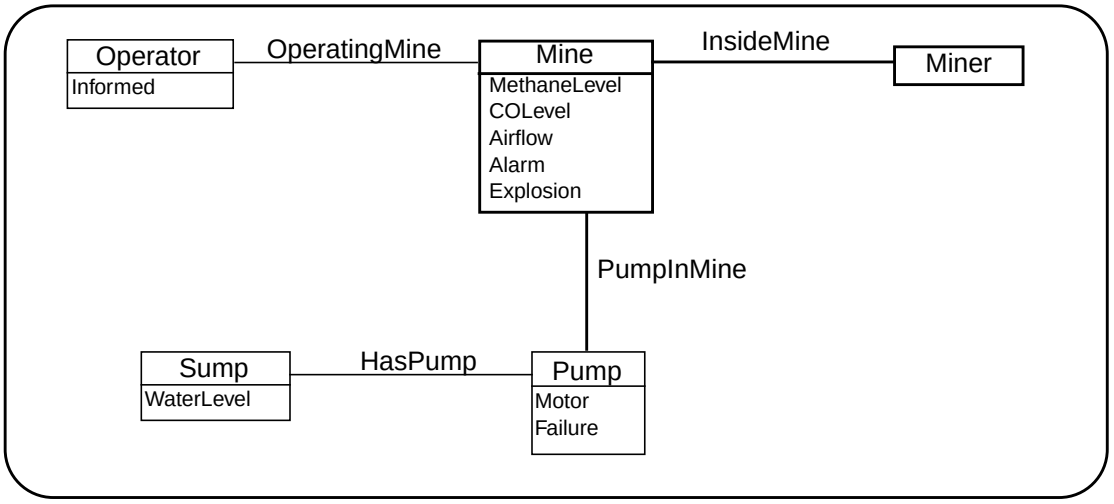


FIGURE 3.13. Object model derived from the preliminary goals

Living with ambiguities

Ambiguities are inevitable in the early stages of the requirement elaboration process. Two places where ambiguities arise are in the definition of objects/attributes and in the definition of goals.

At the early stages of the requirement elaboration process, it is not always clear what definitions should be given to objects and attributes. For instance, the Failure attribute of the Pump entity is not precisely defined in the initial problem statement. A pump failure could have at least one of the following different meanings:

- the motor of the pump refuses to go on;
- the motor of the pump is on but the pump is not pumping water;
- the motor of the pump refuses to go off;
- all or some of the above.

Failing to write a precise definition for objects and objects' attributes makes the model useless. The model cannot be criticized or validated, because it is impossible to know exactly what it means. However, such ambiguities need not be solved as soon as they arise. It is often preferable to solve them after further elaboration of the model, when more knowledge is gained about the system.

Another source of ambiguity arises in the translation of natural language definitions of goals into formal definitions. Natural language definitions are often ambiguous because of the inherent ambiguity of natural languages. They are also sometimes ambiguous on purpose when one does not want to commit oneself to a fixed description of what is exactly required, but rather leaves some freedom about what should be achieved. Again, such ambiguities need not be fully resolved in the very early stages of the requirement elaboration process; they should be resolved gradually as more knowledge is gained about the system. In order to do this, it is helpful to start with idealized formal definitions of goals.

Being Idealist

First-sketch goal definitions tend to be idealistic; it is generally impossible to write realistic goal definitions at the beginning of the requirement elaboration process. First-sketch goals will be impossible to achieve due to agent misbehaviour, limited agent capabilities, or conflicts with other goals. Consider for instance the goal Maintain[PumpOnWhenHighWater]. The goal is idealized because delays to switch on the pump may make it impossible for the pump to be on as soon as the water level is above "high", or because a critical methane level may prevent the pump for being operated, or yet because the pump may fail to start when commanded. Such exceptional cases cannot be all anticipated in the early phases of the requirement elaboration process; idealized goals definitions are thus inevitable.

It is actually *desirable* to start from idealized goal definitions. The reason is simple: premature compromises of what is ideally required prevents the identification of further goals and the exploration of alternatives, such as alternatives ways of deidealizing goals, or to make trade-offs between conflicting goals. Therefore, one should be idealist when writing first-sketch goal definitions.

This recommendation amounts to the traditional advice of specifying ideal behaviours first, and to consider exceptional cases later. For operational specifications, this recommendation makes the specification easier to read and write. At the goal level, specifying the ideal behaviour first is important not only to make the process of writing the specification simpler, but more importantly, to avoid premature, implicit, and probably not optimal compromises.

Systematic techniques for handling idealized goals are studied in Chapter 8 on obstacles analysis. Techniques for identifying and resolving conflicts between goals are described in [Lam98a].

3.3.3.3 Eliciting New Goals through WHY questions

Asking WHY questions enables one to identify higher-level goals that provide rationale for the initial goals. Identifying higher-level goals is important for understanding the initial goals, and for identifying other important subgoals that may have been overlooked in the first place.

There are no clear-cut criteria for knowing when to stop asking why questions [Zav97a]. The quest for higher-level goals should remain within the system's subject matter.

Mine Pump Examples

Asking WHY questions about the goal Maintain[PumpOffWhenLowWater] yields the goal graph shown in Figure 3.14. If the pump motor is on when no water flows into the pump, there is a risk of the pump being burned out. Thus, the goal refines the goal Avoid[PumpOnWhenEmpty] which in turn refines the goal Avoid[PumpBurnedOut]. The dots in the goal graph indicate that the refinements are not complete.

Companion subgoals can be formally elicited through formal refinement patterns [Dar96]. In the goal graph above, the new goal Avoid[PumpOnWhenEmpty] is formally defined as follows:

$p.Empty \Rightarrow p.Status = \text{'Off'}$

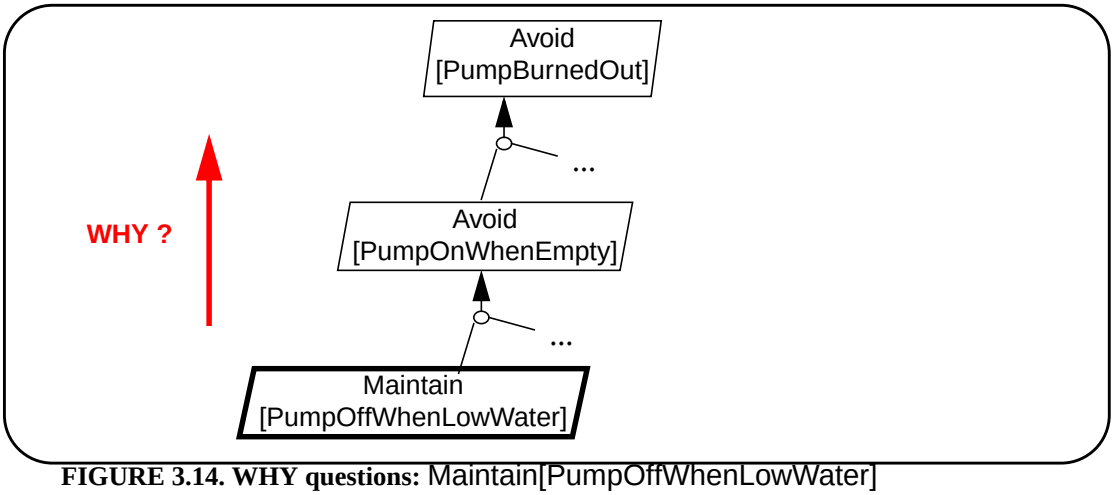
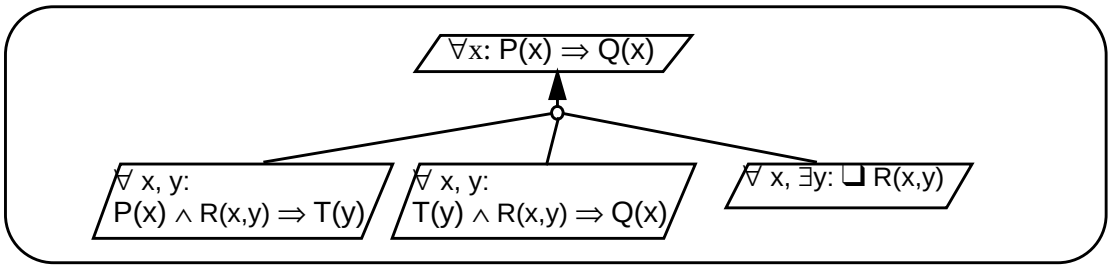


FIGURE 3.14. WHY questions: Maintain[PumpOffWhenLowWater]

The following formal goal refinement pattern can be used to identify companion subgoals [Dar95].



The formal definitions of the goals Maintain[PumpOffWhenLowWater] and Avoid[PumpOnWhenEmpty] matches that pattern with the following instantiations:

P(x): p.Empty Q(x): p.Motor = ‘Off’
T(y):s.WaterLevel ≤ ‘Low’R(x,y): HasPump(s,p)

The instantiation of the pattern yields the following new assertions:

(∀ p: Pump, s: Sump):
p.Empty ∧ HasPump(s,p) ⇒ s.WaterLevel ≤ ‘Low’

(∀ p: Pump, ∃ s: Sump): ❑ HasPump(s,p)

The second assertion is a domain property stating that every mine pump is related to a sump. The first assertion is a new goal constraining the value of the low water level. The following companion subgoal has thereby been elicited formally:

Goal Maintain[AppropriateLowWaterLevel]

Definition The low water level must be defined so that if the water level in the sump is above the low level, the water can flow into the pump.

FormalDef ∀ p: Pump, s: Sump
p.Empty ∧ HasPump(s,p) ⇒ s.WaterLevel ≤ ‘Low’

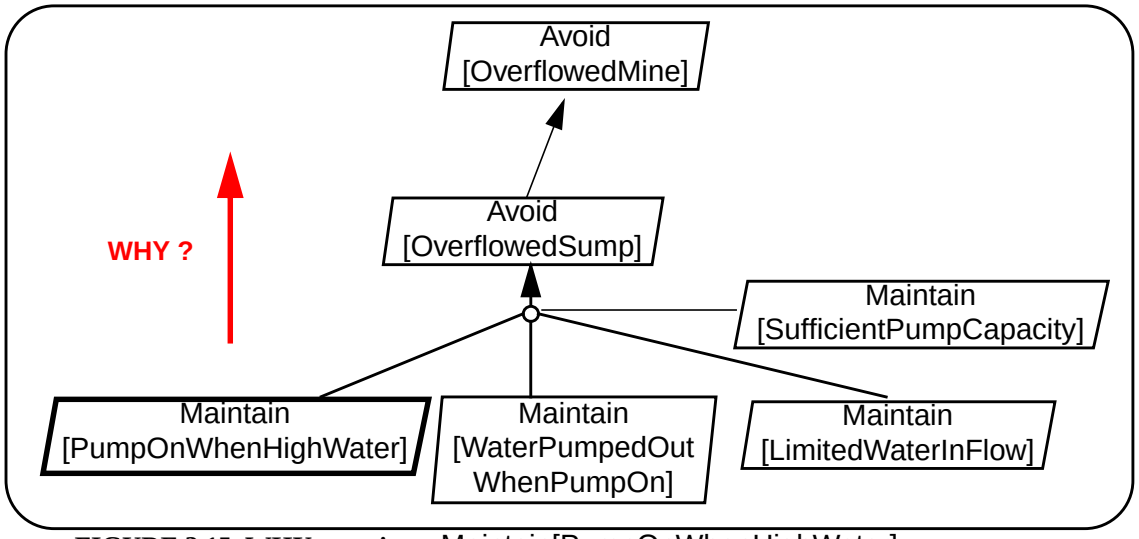


FIGURE 3.15. WHY questions: Maintain[PumpOnWhenHighWater]

Asking WHY questions about the other goals will similarly enrich the goal model with higher-level goals and companion subgoals. For instance, asking a WHY question about the goal Maintain[PumpOnWhenHighWater] yields the goal graph shown in Figure 3.15. The rationale for the goal Maintain[PumpOnWhenHighWater] is the goal Avoid[OverflowedSump] which in turn refines the goal Avoid[OverflowedMine].

Companion subgoals are elicited as well. For the goal Maintain[PumpOnWhenHighWater] to ensure the goal Avoid[OverflowedSump], one has to assume that (i) the pump is actually pumping water out of the mine when the pump motor is on (Maintain[WaterPumpedOutWhenPumpOn]), (ii) the rate at which water flows into the sump is bounded (Maintain[LimitedWaterFlow]), and (iii) the capacity of the pump is above the maximum rate of water that flows into the sump (Maintain[SufficientPumpCapacity]).

Asking WHY questions also allows one to consider *alternative subgoals* to the one initially described. This is illustrated by the following example.

Consider the initial goal Achieve[MineEvacuatedWhenPumpFailure]. Asking a WHY question about that goal yields the parent goal Avoid[MinerInOverflowedMine].

Through applications of formal refinement patterns, one identifies that the goal refinement is complete provided that when there is no pump failure, the mine remains not overflowed for at least one hour (Maintain[MineNotOverflowedWhenNoPumpFailure]), and provided that miners do not enter the mine when there is a pump failure (Avoid[MinerEnteringMineWhenPumpFailure]) -- see the And-refinement on the left of Figure 3.16.

The refinement is based on the assumption, called Maintain[MineNotOverflowedWhenNoPumpFailure], that only pump failures could cause the mine to be overflowed. This assumption is not valid, for instance, when the capacity of the pump is not sufficient to pump the water that flows into the mine, or when the methane level prevents the pump from being operated.

Note also that the initial goal requires the mine to be evacuated as soon as there is a pump failure. This may lead to unnecessary evacuations of the mine if the water that flows into the mine is low.

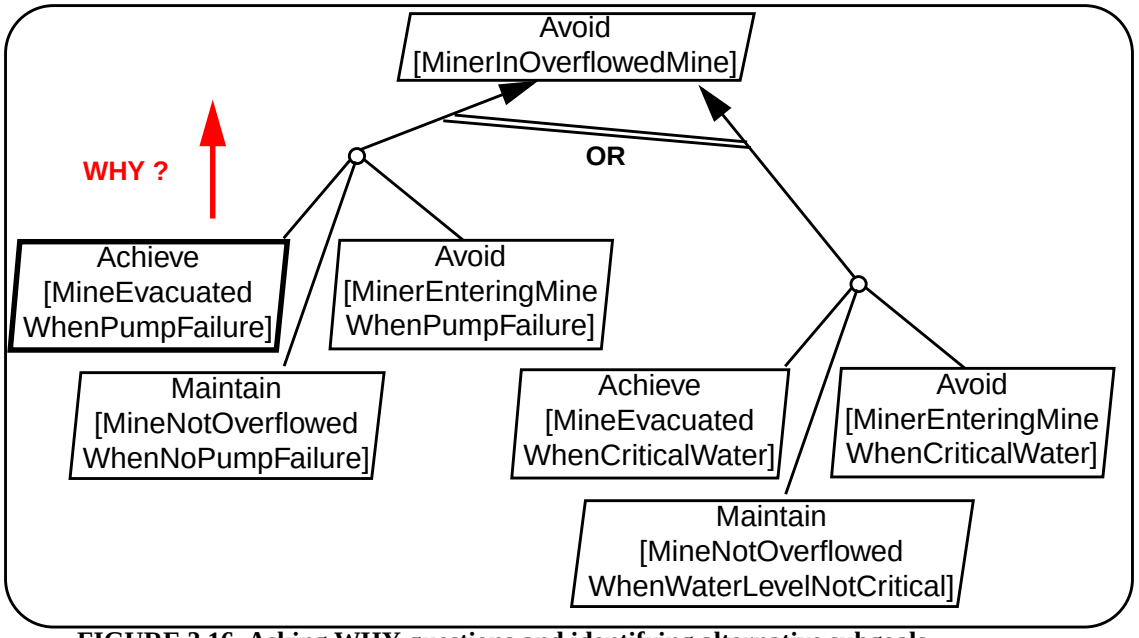


FIGURE 3.16. Asking WHY questions and identifying alternative subgoals

This suggests looking for an alternative refinement of the higher-level goal Avoid[MinerInOverflowedMine].

An alternative refinement of the goal is shown in Figure 3.16. This alternative requires the mine to be evacuated when the water level in the sump remains high during a certain period of time:

Goal Achieve[MineEvacuatedWhenCriticalWater]

Definition If the water level in the sump remains high during a period of 'CriticalDelay' time units, the mine should be evacuated within one hour.

FormalDef $\forall$ m: Mine, s: Sump

$\blacksquare_{\leq CriticalDelay}$ s.WaterLevel $\geq$ 'High' $\wedge$ SumpInMine(s,m)
$\Rightarrow \Diamond_{\leq 1h} \neg$ ($\exists$ p: Miner): InsideMine(p,m)

To summarize, identifying higher-level goals through WHY questions is important because:

- it provides a rationale for the initial goals;
- it enables one to formally identify companion subgoals that were overlooked in the first place;
- it enables one to explore alternative subgoals that may provide better solutions to the higher level goals.

The quest for higher-level goals and alternative subgoals should of course remain within the system's subject matter.

3.3.3.4 Eliciting new goals through HOW questions

Another step of the requirements elaboration process consists in refining goals until reaching subgoals that can be assigned to individual agents. Subgoals are identified by asking HOW questions about the goals already identified.

Mine Pump Examples.

For example, a HOW question about the goal Maintain[PumpOnWhenHighWater] yields the goal-refinement shown in Figure 17.

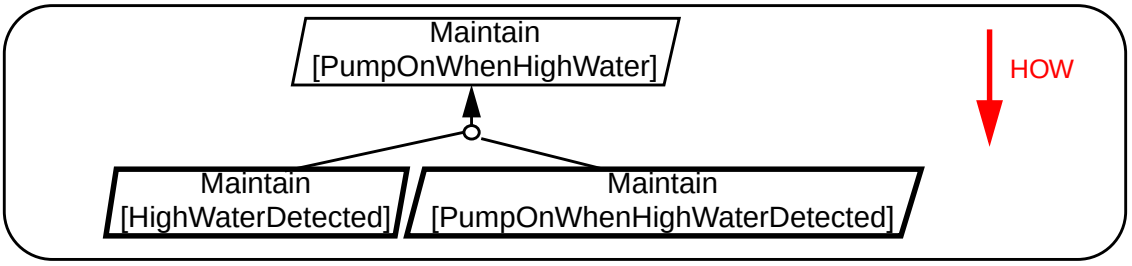


FIGURE 3.17. Refinement of the goal Maintain[PumpOnWhenHighWater]

The first subgoal is formally defined as follows:

Goal Maintain[HighWaterDetected]
InstOf AccuracyGoal
Definition The HighWaterSignal must be on when the water level in the sump is above high.
FormalDef $\forall$ s: Sump, c: PumpController
s.WaterLevel $\geq$ 'High' $\wedge$ HasCtrler(s,c) $\Rightarrow$ c.HighWaterSignal = 'on'

That goal is an accuracy goal relating the actual water level in the sump to a new attribute, HighWaterSignal, denoting a signal received by the PumpController agent from a HighWaterSensor agent. The second subgoal is defined as follows.

Goal Maintain[PumpOnWhenHighWaterDetected]
Definition The pump must be on when the HighWaterSignal is on.
FormalDef $\forall$ p: Pump, c: PumpController
c.HighWaterSignal = 'on $\wedge$ CtrlPump(c,p) $\Rightarrow$ p.Status = 'on'

The goal refinement also uses the two following domain properties:

$\forall$ s: Sump, $\exists$ c: PumpController: ❑ HasCtrler(s,c)

$\forall$ s: Sump, c: PumpController, p: Pump
HasCtrler(s,c) $\wedge$ HasPump(s,p) $\Rightarrow$ CtrlPump(c,p)

The formalization of the two subgoals together with the above properties may be used to prove that together they entail the parent goal Maintain[PumpOnWhenHighWater] formalized before.

Similarly, asking a HOW question about the generated subgoal Maintain[PumpOnWhen-HighWaterDetected] yields a new portion of the goal-refinement graph shown in Figure 18.

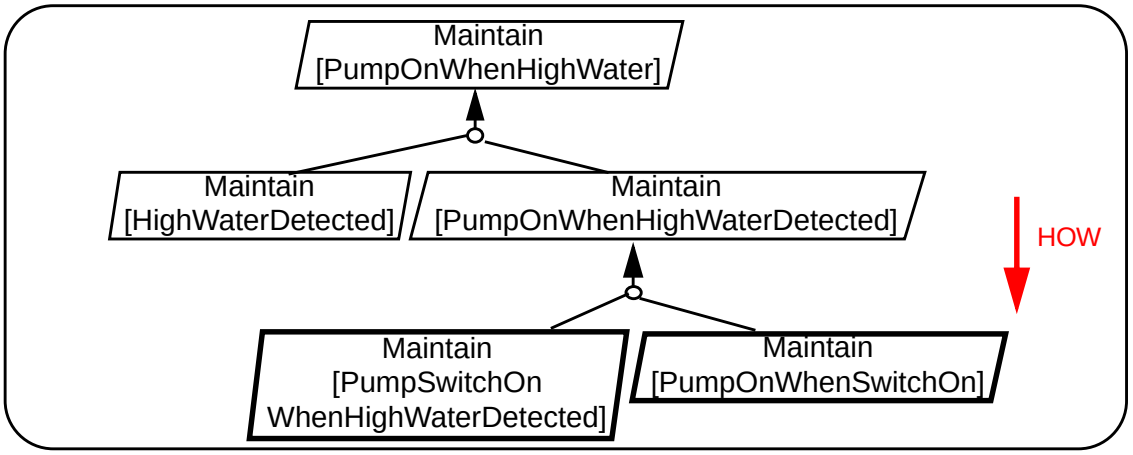


FIGURE 3.18. Refinement of the goal Maintain[PumpOnWhenHighWaterDetected]

The new subgoals are defined as follows.

Goal Maintain[PumpSwitchOnWhenHighWaterDetected]

Definition The pump switch must be set to on when the HighWater flag of the pump controller is on.

FormalDef $\forall$ p: Pump, c: PumpController

c.HighWater = 'on' $\wedge$ CtrlPump(c,p) $\Rightarrow$ p.Switch = 'on'

Goal Maintain[PumpOnWhenSwitchOn]

Definition The pump motor is on when the pump switch is set to on.

FormalDef $\forall$ p: Pump

p.Switch = 'on' $\Rightarrow$ p.Motor = 'on'

This goal refinement also leads to the identification of a new Pump.Switch attribute.

Again, the formalization of these subgoals can be used to prove that together they entail the parent goal Maintain[PumpOnWhenHighWaterDetected] formalized before.

Note that the above definitions are first approximations. They do not take reaction times into account. In fact, the pump motor cannot start running instantaneously when the pump switch is set to on. If PumpDelay is the time taken to switch the pump motor on, the last goal above can be formalized as follows:

$$\bullet\blacksquare_{\leq \text{PumpDelay}}\ \text{p.Switch = 'on'} \Rightarrow \text{p.Status = 'on'}$$

It asserts that if the pump switch has been on for the last PumpDelay time units, then the pump motor is on. (The $\bullet$ operator is a formal "noise" necessary in a later stage to derive operational requirements satisfying the goal.)

Similarly, the goal Maintain[PumpSwitchOnWhenHighWaterDetected] requires the pump controller to react synchronously to the values received from the high water sensor. If the reaction time of the pump controller is assumed to be a positive constant CtrlrDelay, the goal is temporally weakened as follows:

$$\bullet\blacksquare_{\leq \text{CtrlrDelay}}\ [\ \text{c.HighWater = 'on} \wedge \text{CtrlPump(c,p)}\ ] \Rightarrow \text{p.Switch = 'on'}$$

Finally, the goal Maintain[HighWaterDetected] is also temporally weakened to take into account the delay of the water sensor:

$$\bullet\blacksquare_{\leq \text{WaterSensorDelay}} [\ s.\text{WaterLevel} \geq \text{'High'} \wedge \text{HasCtrler}(s,c)\ ] \Rightarrow c.\text{HighWater} = \text{'on'}$$

The weakening of these goals is then propagated along the goal graph. As a result, the goal Maintain[PumpOnWhenHighWater] is now defined by

$$\blacksquare_{\leq \text{SafetyMargin}} [\ s.\text{WaterLevel} \geq \text{'High'} \wedge \text{HasPump}(s,p)\ ] \Rightarrow p.\text{Status} = \text{'On'}$$

For the goal refinements to be complete, the delays and safety margin must satisfy the following constraint:

$$\text{SafetyMargin} \geq \text{PumpDelay} + \text{CtrlrDelay} + \text{WaterSensorDelay} + 3\,\delta\ .$$

(δ is the smallest time unit. The term $3\,\delta$ comes from the $\bullet$ operator in the definitions of the subgoals.)

The safety margin is also defined so that the goal Maintain[PumpOnWhenHighWater] still ensures its parent goal Avoid[OverflowedSump] in Figure 15. The validity of the goal refinements in Figure 15 also relies on the domain properties that the relationships HasPump, HasCtrler, and CtrlPump do not change over time, i.e.

$$\text{HasPump}(s,p) \Leftrightarrow \Box\, \text{HasPump}(s,p)$$
$$\text{HasCtrler}(s,c) \Leftrightarrow \Box\, \text{HasCtrler}(s,c)$$
$$\text{CtrlPump}(c,p) \Leftrightarrow \Box\, \text{CtrlPump}(c,p)$$

These assertions are captured as domain properties on the corresponding relationships.

The goal Maintain[PumpOnWhenHighWater] has now been refined into subgoals that can be assigned as the responsibility of single agents (the potential responsibility assignments of goals to agents is considered below).

The goals Maintain[PumpOffWhenLowWater] and Maintain[PumpOffWhenCritical-Methane] are refined in a similar way. Their goal refinement graphs are shown in Figure 3.19.

3.3.4 Elaborating Alternative Agent Models

Alternative agent models are gradually elaborated for the goals elicited. The activity of elaborating alternative agent models consists in the following steps:

1. Identify potential agents and responsibility assignments of goals to agents;
2. Derive agent interfaces from responsibility assignments of goals to agents;
3. Operationalize goals through operations and associated requirements.

For the purpose of presenting the goal-oriented requirements elaboration method, the identification of agent monitoring and control capabilities is here delayed until after the goals have been assigned as responsibilities of single agents. This process emphasizes that alternative agent interfaces are a result of alternative goal refinements and responsibility assignments. In fact, considerations about possible agent monitoring and control capabilities influence the goal refinement process. For instance, the above refinements of

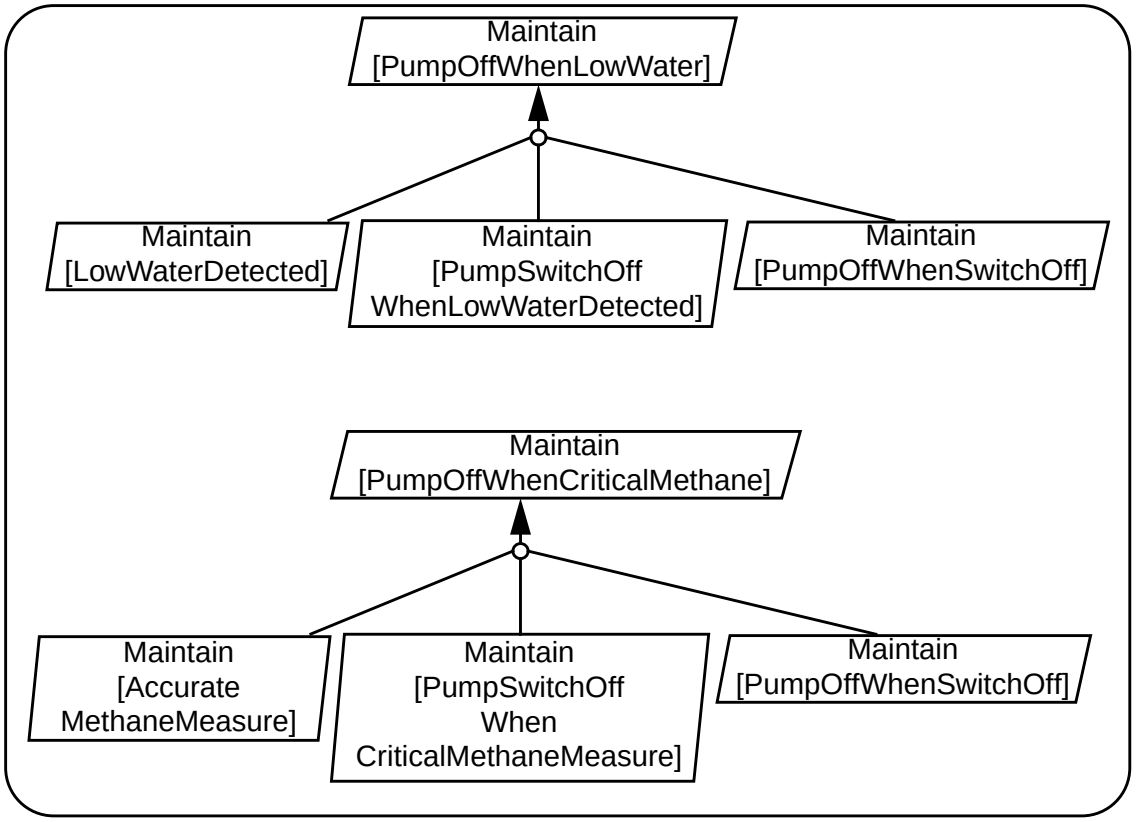


FIGURE 3.19. Refinement of the goals Maintain[PumpOffWhenHighWater] and Maintain[PumpOffWhenCriticalMethane]

the goal Maintain[PumpOnWhenHighWater] were driven by the need to resolve lack of monitoring and control capabilities of the PumpController agent. The activities of elaborating the goal and agent models are therefore much intertwined. Such intertwining is further described in Section 3.3.5.

3.3.4.1 Identifying potential responsibility assignments

Potential agents and potential responsibility assignments of goals to agents are identified during the goal refinement process.

Mine Pump Examples

Potential responsibility assignments for the goal refinement graph of the goal Maintain[PumpOnWhenHighWater] are given in Figure 3.20; the goal Maintain[HighWaterDetected] is assigned to a HighWaterSensor agent described in the initial problem statement; the goal Maintain[PumpSwitchOnWhenHighWaterDetected] is assigned to the PumpController software agent; and the goal Maintain[PumpOnWhenSwitchOn] is assigned to a PumpActuator agent.

Responsibility assignments for the subgoals of the goals Maintain[PumpOffWhenLowWater] and Maintain[PumpOffWhenCriticalMethane] are shown in Figure 3.21.

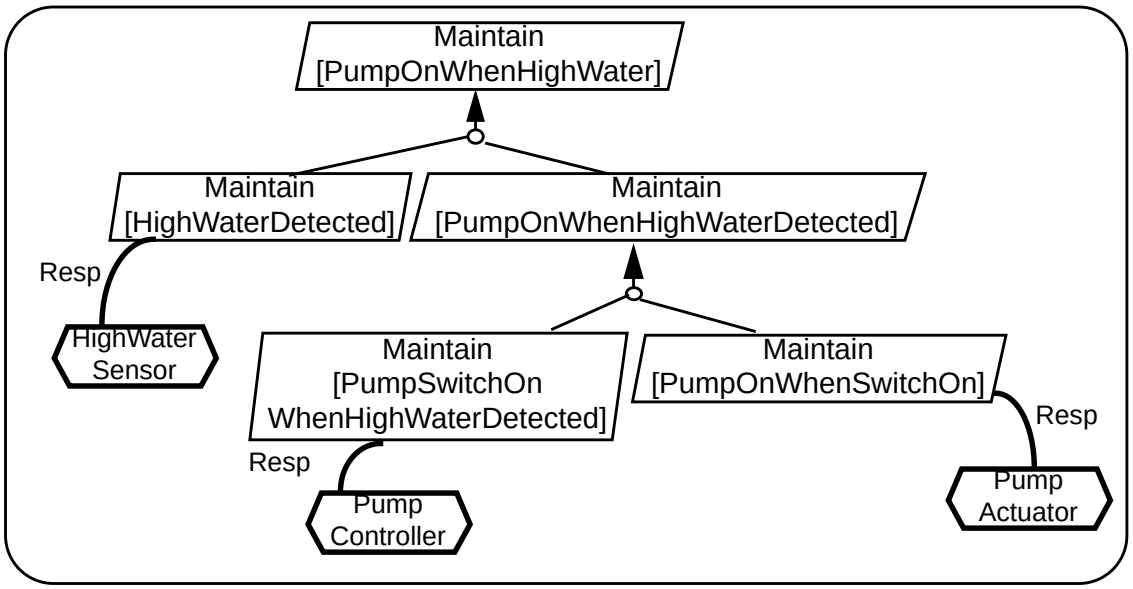


FIGURE 3.20. Responsibility assignments for the subgoals of Maintain[PumpOnWhenHighWater]

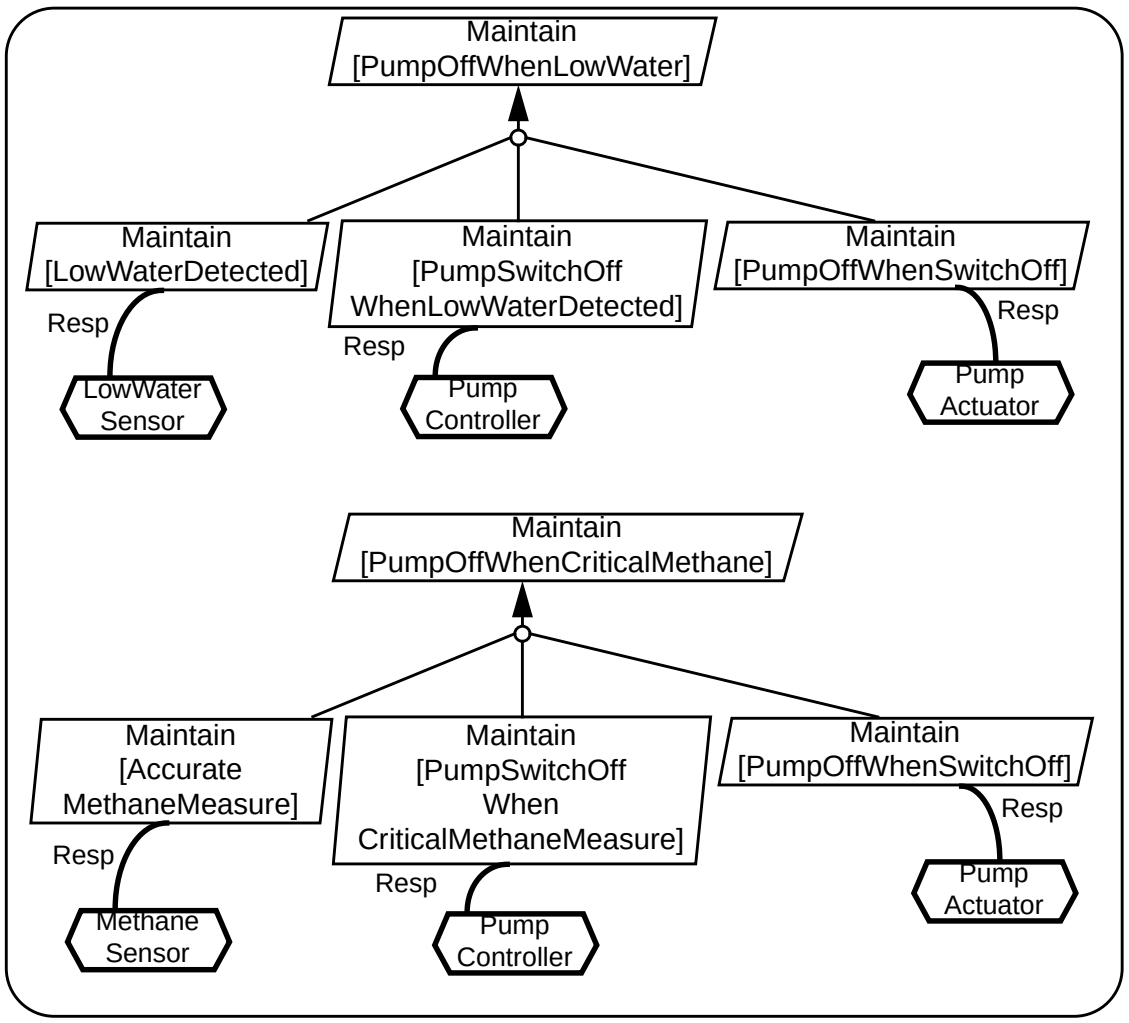


FIGURE 3.21. Responsibility assignments for the subgoals of Maintain[PumpOffWhenLowWater] and Maintain[PumpOffWhenCriticalMethane]

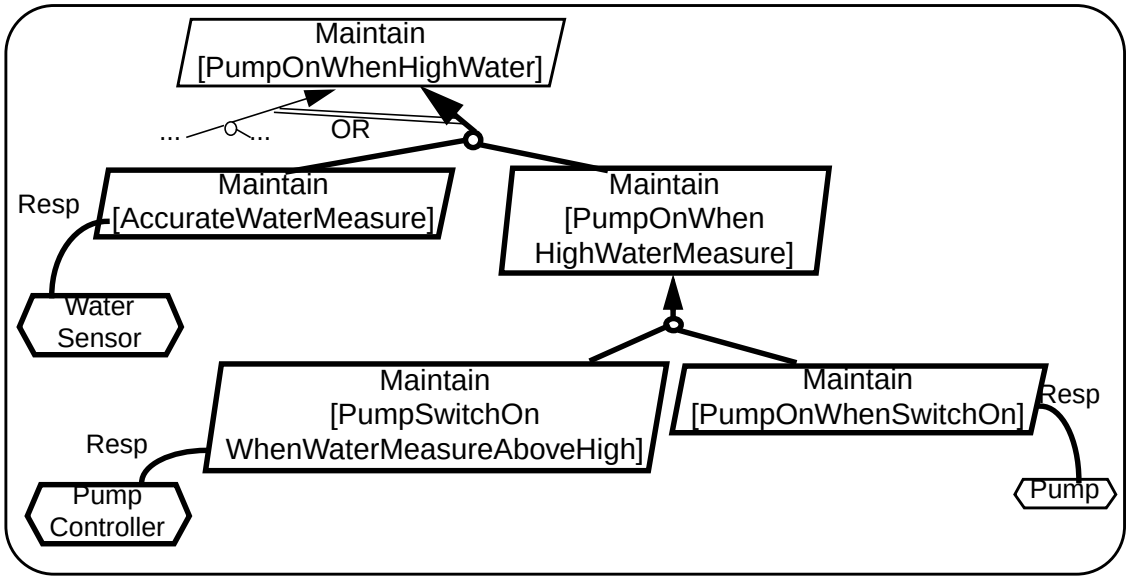


FIGURE 3.22. Alternative refinement and responsibility assignments for Maintain[PumpOnWhenHighWater]

These responsibility assignments are the ones suggested by the initial problem statement. Alternative goal refinements and responsibility assignments could have been explored. For instance, the goal Maintain[PumpOnWhenHighWater] could have been alternatively refined into the subgoals Maintain[AccurateWaterMeasure] and Maintain[PumpOnWhenHighWaterMeasure] defined as follows:

Goal Maintain[AccurateWaterMeasure]

InstOf AccuracyGoal

Definition The water measure should equal the actual water level in the sump.

FormalDef $\forall$ s: Sump, p: Pump, c: PumpController

CtrlPump(c,p) $\wedge$ HasPump(s,p) $\Rightarrow$ c.WaterLevelMeasure= s.WaterLevel

Goal Maintain[PumpOnWhenHighWaterMeasure]

Definition The pump must be on when the water measure is above high.

FormalDef $\forall$ p: Pump, c: PumpController

c.WaterLevelMeasure $\geq$ 'High' $\wedge$ CtrlPump(c,p) $\Rightarrow$ p.Motor = 'on'

The goal Maintain[AccurateWaterMeasure] is an accuracy goal that can be assigned as the responsibility of a water sensor agent. This goal refers to a WaterLevelMeasure attribute instead of the HighWaterSignal attribute referenced in the goal Maintain[HighWaterDetected]. The resulting alternative responsibility assignments are shown in Figure 3.22. Similarly, alternative refinements and responsibility assignments can be defined for the goal Maintain[PumpOffWhenLowWater]. In this alternative, a unique water sensor is responsible for measuring the water level in the sump, and the responsibility of comparing the measured water level to the High and Low water levels is now given to the PumpController.

More complex systems generally have much more radical alternative designs. Further examples of alternative goal refinements and responsibility assignments will be described for the LAS ambulance dispatching system and the BART train control system in Chapter 9.

3.3.4.2 Deriving agent interfaces

This step consists in deriving agent monitoring and control links from their responsibility assignments. The derivation will be informal here; it is guided by the realizability meta-constraint introduced in Section 3.8.2. Formal techniques for deriving agent interfaces from responsibility assignments are described in Chapter 7.

Mine Pump Examples

Consider the goal Maintain[PumpSwitchOnWhenHighWaterDetected] assigned as the responsibility of the PumpController agent. To fulfil its responsibility, the pump controller must monitor the HighWaterSignal attribute and control the value of the Switch attribute of the pump. The following Monitoring and Control links are thereby derived:

Monitoring [PumpController, PumpController.HighWaterSignal]
InstDecl ∀ c: PumpController
Mon(c, c.HighWaterSignal)

Control [PumpController, Pump.Switch]
InstDecl ∀ c: PumpController
CtrlPump(c,p) ⇒ Ctrl(c, p.Switch)

Through similar reasoning, the agent interface model in Figure 23 is derived from the responsibility assignments of Figure 3.21.

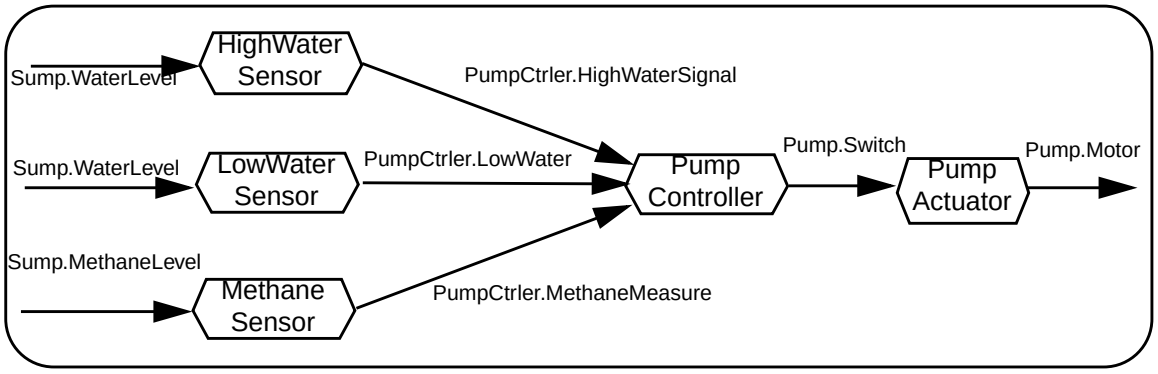


FIGURE 3.23. Agent interface model derived from the responsibility assignments of Figures 3.20 and 3.21

Note that alternative responsibility assignments generally yield alternative agent interfaces. For instance, the alternative in which the water level is measured by a single water sensor yields the agent interface model of Figure 3.24.

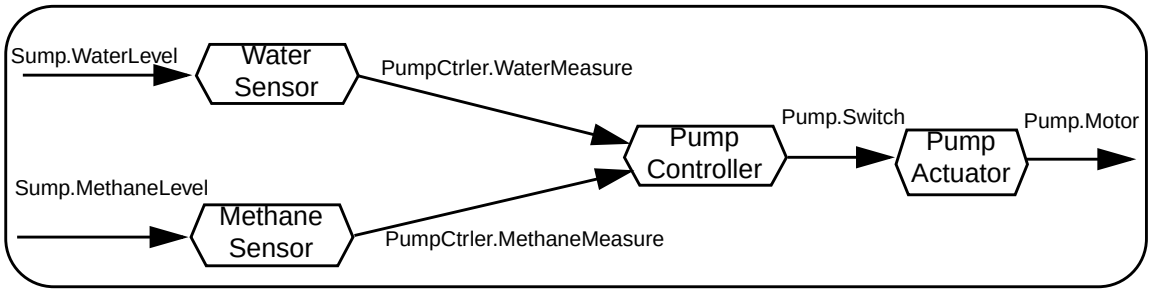


FIGURE 3.24. Alternative agent interface model derived from the alternative responsibility assignments in Figure 3.22

3.3.4.3 Operationalizing goals

This step consists in identifying operations relevant to the goals and deriving requirements on operations so that the goals are satisfied. there are two sub-steps:

1. *Identify operations*. Specific state transitions referred to in goals formulation are identified. Only elementary domain pre- and post-conditions are identified. Such domain pre- and post- conditions do not ensure the goal from which they are derived.

2. *Derive requirements on operations*. The identified operations are strengthened with required pre-, trigger, and post conditions so that the goals are satisfied.

Formal techniques for deriving operations and requirements on operations from goals will be explored in chapter 8.

Mine Pump Examples

The goal Maintain[PumpSwitchOnWhenHighWaterDetected] assigned to the PumpController agent constrains the value of the Pump.Switch attribute. The following operations are then identified from that goal.

Operation SwitchPumpOn
PerfBy PumpCtrler {**arg** c}
Input Pump {**arg** p}
Output Pump {**res** p} / Switch
DomPre p.Switch = 'Off' $\wedge$ CtrlPump(c,p)
DomPost p.Switch = 'On'

Operation SwitchPumpOff
PerfBy PumpCtrler {**arg** c}
Input Pump {**arg** p}
Output Pump {**res** p} / Switch
DomPre p.Switch = 'On' $\wedge$ CtrlPump(c,p)
DomPost p.Switch = 'Off'

These definitions minimally capture what the switching on and off of the pump is about in the application domain.

The following required conditions on the applications of the operations are derived so as to ensure the goal:

Operation SwitchPumpOn
...
ReqTrigFor Maintain[PumpSwitchOnWhenHighWaterDetected]
$\blacksquare_{\leq \text{CtrlrDelay}}$ c.HighWaterSignal = 'On'

Operation SwitchPumpOff
...
ReqPreFor Maintain[PumpSwitchOnWhenHighWaterDetected]
$\neg \blacksquare_{\leq \text{CtrlrDelay}}$ c.HighWaterSignal = 'On'

The *required trigger* condition on the SwitchPumpOn operation requires that the operation must be applied when the HighWaterSignal has been On for a delay of CtrlrDelay. The *required precondition* on the SwitchPumpOff operation requires that the operation is not applied if the HighWaterSignal has been On for a delay of CtrlrDelay.

Other goals assigned to the pump controller agent give rise to further requirements on these operations. For instance, the following operational requirements are derived from the goals Maintain[PumpSwitchOffWhenLowWaterDetected] and Maintain[PumpSwitchOffWhenCriticalMethaneMeasure]:

Operation SwitchPumpOn

...

ReqPreFor Maintain[PumpSwitchOffWhenLowWaterDetected]

$\neg \blacksquare_{\leq \text{CtrlrDelay}}$ c.LowWaterSignal = 'Off'

ReqPreFor Maintain[PumpSwitchOffWhenCriticalMethaneMeasure]

$\neg \blacksquare_{\leq \text{CtrlrDelay}}$ c.MethaneMeasure ≥ 'CriticalMethane'

Operation SwitchPumpOff

...

ReqTrigFor Maintain[PumpSwitchOffWhenLowWaterDetected]

$\blacksquare_{\leq \text{CtrlrDelay}}$ c.LowWaterSignal = 'Off'

ReqTrigFor Maintain[PumpSwitchOffWhenCriticalMethaneMeasure]

$\blacksquare_{\leq \text{CtrlrDelay}}$ c.MethaneMeasure ≥ 'CriticalMethane'

Note that the derived requirements on these operations violate the meta-constraint

ReqTrig ⇒ ReqPre.

For the operation SwitchPumpOn, the required trigger condition for the goal Maintain[PumpSwitchOnWhenHighWaterDetected] can be true while the required condition for the goal Maintain[PumpSwitchOffWhenCriticalMethaneMeasure] is false. This inconsistency is due to a conflict at the goal level between the goals Maintain[PumpOnWhenHighWater] and Maintain[PumpOffWhenCriticalMethane]; it needs to be solved at that level. The handling of conflicting goals is briefly described later.

3.3.5 Goal refinement and agent identification: an intertwined process

As mentioned before, the activities of refining goals into subgoals and generating alternative agent responsibilities and interfaces are much intertwined. The principal reason for refining goals into subgoals is to eventually reach subgoals that can be assigned as the responsibility of single agents. Preliminary information about agents potentially available is therefore needed to guide the goal refinement process.

The goal refinement process is guided by the following principles [Dar93]:

1. Stop refining a goal when it can be assigned as the responsibility of a single agent;

2. Refine goals into subgoals so that the latter require the cooperation of fewer potential agents.

The first principle provides a criterion for stopping the goal-refinement process. The second principle ensures the convergence of the goal refinement process towards subgoals that can be assigned to single agents.

The following activities are therefore recursively performed during the goal-refinement process.

Identifying potential agents and their capabilities -- Preliminary information about potential agents and their capabilities is needed to determine when the goal refinement process may stop, and to guide the refinement process towards subgoals that can be assigned to individual agents.

Preliminary information about agents and their monitoring and control capabilities is identified from initial goals by asking WHO could play a role in achieving the goals and WHO is capable of monitoring and controlling the objects referenced in the goal formulations. Such information about agents and their capabilities is also often described in the preliminary materials describing the envisioned system.

It is important to note that the result of this preliminary identification is a *partial* agent interface model that may include *alternative* monitorability and control links. Further agents and agent capabilities are then gradually and systematically identified during the goal-refinement process.

Mine Pump Example

For example, asking WHO could play a role in achieving the goal Maintain[PumpOnWhenHighWater] may lead to the identification of the PumpController software agent. Other agents such as the HighWaterSensor and the PumpActuator could also be identified at this preliminary stage by asking WHO could be capable of monitoring and controlling the objects referenced in this goal. These agents can also be systematically identified later during the goal refinement process.

Identifying goal unrealizability-- Every time a goal is produced one has to check whether the goal is realizable by a single agent already identified. If the goal is realizable by an agent, it may be potentially assigned as the responsibility of that agent. If the goal is not realizable by a single agent, the model has to be elaborated further.

Mine Pump Example.

Consider the goal Maintain[PumpOnWhenHighWater]. It describes a relation between the water level in the sump and the status of the pump motor. This goal is not realizable by the PumpController software agent, because it lacks monitoring capabilities for the actual water level in the sump; and it lacks control capabilities for the status of the pump motor. Therefore, the goal cannot be assigned as the responsibility of that agent, and the model has to be elaborated further.

Resolving goal unrealizability-- Once realizability problems have been identified, the model has to be elaborated so as to resolve theses problems. Resolution of realizability problems drives the identification of new agents and the refinement of goals into subgoals.

Mine Pump Examples

As mentioned above, the goal Maintain[PumpOnWhenHighWater] is not realizable by the PumpController software agent because it cannot monitor the actual water level in the sump and cannot control the status of the pump motor.

In order to resolve the lack of monitoring capabilities of the PumpController for the actual water level in the sump, the goal Maintain[PumpOnWhenHighWater] was refined into the subgoals:

Maintain[HighWaterDetected]

Maintain[PumpOnWhenHighWaterDetected].

During this requirement elaboration step, we also identified the new HighWaterSignal attribute monitorable by the PumpController agent, and the HighWaterSensor agent capable of monitoring whether the water level in the sump is above high and controlling the HighWaterSignal attribute. The generated subgoal Maintain[HighWaterDetected] can then be assigned as the responsibility of that agent.

The goal Maintain[PumpOnWhenHighWaterDetected] is still not realizable by the PumpController because it cannot control the variable Pump.Motor. In order to resolve this realizability problem, the goal is refined into the subgoals:

Maintain[PumpSwitchOnWhenHighWaterDetected]

Maintain[PumpOnWhenSwitchOn].

During this requirement elaboration step, we also identified the new Pump.Switch attribute controllable by the PumpController agent; and the PumpActuator agent capable of monitoring the pump switch and controlling the status of the pump motor. The generated subgoals Maintain[PumpSwitchOnWhenHighWaterDetected] and Maintain[PumpOnWhenSwitchOn] are now realizable by the PumpController and the PumpActuator agents, respectively.

A contribution of this thesis is to provide formal support to assist users in applying this intertwined goal refinement and agent identification process. Chapter 5 describes formal techniques for identifying realizability problems; Chapter 6 describes formal techniques for refining goals and identifying agents so as to resolve realizability problems.

Note that this process of elaborating the requirement model by identifying and resolving violations of the realizability meta-constraint is an application of a more general scheme that consists in using meta-constraints from the meta-model to guide the requirement elaboration process (see Section 3.2.2).

3.3.6 Goal-Oriented Analysis

There is more to goal-oriented requirements engineering than what has been outlined above. Three important aspects of requirements engineering have not been covered in the previous sections: (i) the handling of conflicts between goals, (ii) the handling of agent misbehaviour, and (iii) the evaluation and selection of alternatives. These aspects are now briefly discussed.

3.3.6.1 Conflict Analysis

During the elaboration of the goal model, conflicts between goals are identified, and alternative conflict resolutions are proposed. The selection of one alternative over the others is performed during the alternative evaluation activity. Formal techniques for identifying conflicts between goals and for generating alternative resolutions are described in [Lam98b].

Mine Pump Example

The conflict between the goals Maintain[PumpOnWhenHighWater] and Maintain[PumpOffWhenCriticalMethane] has been described in Section 3.3.2.4.5. The divergence between these goals, and the boundary condition for the divergence is recalled in Figure 3.5.

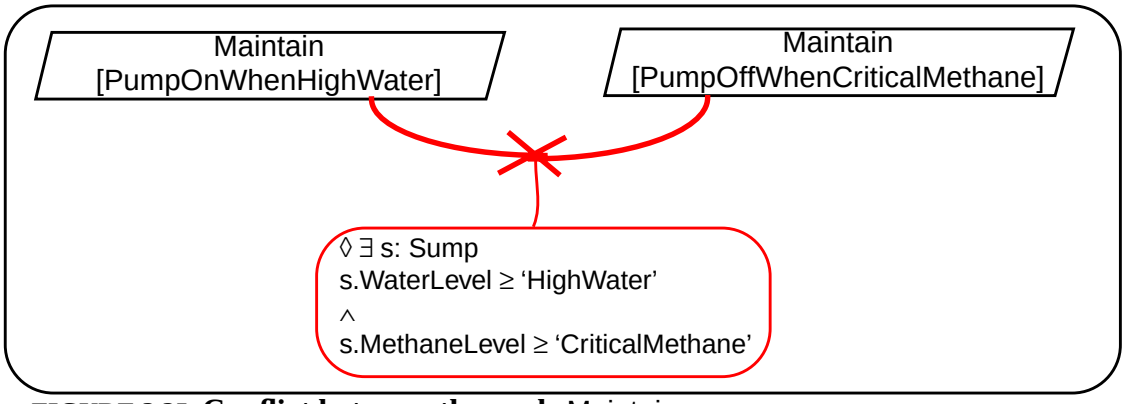


FIGURE 3.25. Conflict between the goals Maintain [PumpOnWhenHighWater] **and** Maintain[PumpOffWhenCriticalMethane]

The boundary condition states that the two goals become inconsistent when the methane level is critical while the water level is above high.

Various techniques for resolving divergences are described in [Lam98b]. Here, the divergence is resolved by weakening the goal Maintain[PumpOnWhenHighWater]:

Goal Maintain[PumpOnWhenHighWater]

Definition The pump must be On when the water level in the sump is above the high water level ***except if the methane level is critical***.
FormalDef ∀ s: Sump, p: Pump:
s.WaterLevel ≥ 'HighWater' ∧ HasPump(s,p)
⇒
p.Motor = 'On' ∨ **s.MethaneLevel ≥ 'CriticalMethane'**

Note that the goal is weakened by adding the boundary condition as a disjunct in the consequent of the goal.

Once the goal is weakened, the transformation of the goal definition is propagated up and down along the goal refinement and operationalization links.

3.3.6.2 Obstacle Analysis

Goals produced during the refinement process tend to be idealized. They are likely to be violated due to exceptional agent behaviours. In the mine pump example, the water sensors may fail to detect correctly the high and low water level; the pump may refuse to start or stop running; the pump controller may also fail to produce correct outputs in time; etc.

The identification such exceptional situations drives the elicitation of further goals to prevent them or to mitigate their consequences. For instance, one could identify a new goal specifying what should happen if the pump refuses to stop. Some goals described in the initial problem statement were already introduced to mitigate the consequences of agent failures. The goal requiring the mine to be evacuated in case of pump failure is an example of such goals.

Exceptional situations that prevent the fulfilment of goals are called *obstacles*. The precise definition of obstacles, and the definition of formal techniques for identifying obstacles and for resolving them are described in Chapter 9.

3.3.6.3 Alternative evaluation and selection

Another important aspect that has been left out up to now concerns the evaluation and selection of alternatives. As mentioned before, the selection of alternatives is based on softgoals and optimization goals such as reduce costs, minimize risks, etc. Preliminary techniques for evaluation and selection are described in [Fea91], [Rob90], and [Myl99]. This is an area where much research remains to be done; we will not address it in the thesis.

3.4. Summary and Outlook

The work reported in the thesis is based on an existing goal-oriented requirement elaboration method, called KAOS. This method consists in identifying goals and refining them into subgoals until the latter can be assigned as responsibilities of single agents. The method supports the exploration of alternative goal refinements, and alternative responsibility assignments of goals to agents, resulting in alternative system proposals in which the boundary between the automated system and its environment may be quite different.

This chapter has described the KAOS goal-oriented language and methods, and proposed extensions of the language to model *agent interfaces* through monitoring and control links. We also introduced a *realizability* meta-constraint relating an agent's responsibility for a goal to its interface: a goal is realizable by an agent if it defines a relation between objects monitored and controlled by the agent.

The realizability meta-constraint plays a significant role in the goal-oriented requirements elaboration process: violations of realizability drive the identification of agents and the refinement of goals into subgoals until the latter are realizable by individual agents.

Extensions to the KAOS goal-oriented method will described in the following chapters:

- Chapters 5 and 6 extend the KAOS goal-oriented method with formal techniques for identifying realizability problems and for resolving them by identifying agents and refining goals into subgoals. These techniques are based on a formal model of agent responsibility, monitorability, and control defined in Chapter 4.
- Chapter 7 describes techniques for deriving operational requirements from goals, and for deriving agent monitoring and control links from responsibility assignments.
- Chapter 8 describes formal techniques for handling exceptional agent behaviors that may block the fulfillment of idealized goals and assumptions.

Chapter 4
A Formal Model for Agents

This chapter defines a formal model of agents for goal-oriented engineering. The objective is to set the foundations over which the requirements elaboration techniques of the following chapters are grounded. In particular, we give a formal definition for the realizability meta-constraint between an agent's responsibility for goals and its interface.

The formal model of agents is intended to be part of a complete semantics of the KAOS language. Section 1 proposes a general framework for defining the semantics of the KAOS language, and describes how the formal model of agents fits in that framework. Section 2 defines the formal model of agents. Section 3 defines what is meant for a goal to be realizable by a single agent, and defines necessary and sufficient conditions characterizing realizable goals.

4.1. Towards a Formal Semantics for the KAOS Language

4.1.1. Motivation

In order to provide systematic formal support during the requirement elaboration process, it is necessary to assign a mathematical meaning to KAOS models. Such mathematical semantics for the language is not an objective per se but only a means for reaching our primary objective of providing systematic guidance during the requirement elaboration process. The two important objectives of a semantics of the KAOS language are:

- to clarify the meaning of language constructs; and
- to provide a basis on which to define and integrate dedicated reasoning techniques and tool support for the KAOS method.

In order to serve those objectives, the semantics of the language should meet the following qualities.

Mathematically grounded -- A mathematical semantics is needed for defining and integrating techniques and tool support for reasoning about KAOS specifications. It also helps in detecting and correcting ambiguities and other flaws in the definition of the language.

Simple and intuitive -- A formal semantics should clarify the meaning of a language, not obscure it. We wish the semantics to give clear insights to the *users* of the language. It is not intended to be solely for use by tool developers. We wish the semantics to both follow and give intuition about the language.

Supporting partial specifications -- The KAOS method is intended to support incremental elaboration of requirements; the language must therefore be able to support partial and incomplete specifications. It is thus fundamental to be able to give a meaning to

partial specifications and to support reasoning about them. This contrasts with many formal methods where formal verification is usually performed a posteriori, after the model has been fully specified.

Such need for reasoning about partial specifications has a strong influence on the semantics of the KAOS operation model defined in chapter 8.

Designed for ease of language extension and contraction -- Because the language is expected to evolve, its formal semantics should be easy to maintain. One may wish to add further models to the set of models already supported by the language. The integration in the KAOS language of the concept of obstacle (described in Chapter 8) is an example of such an extension. Among other likely language changes are the choice of the particular formalisms used for the formal layer of the language. There are many competing formalisms and progress towards more expressive and efficient formalisms are likely to occur. On the other side, the full power of the KAOS language is not always needed. In many cases, people will only use the 'semi-formal' declaration layer of the language. It is fundamental for them to be able to understand the meaning of their model without studying the particular formalism used at the formal layer of the language. Sometimes, it might also be decided to perform only goal analysis. It should then be possible to understand the semantics of the various constructs of the goal model - such as goal refinement, goal conflict, etc. - without referring to the other models.

Note that the objective of support for reasoning about partial specifications and the objective of a maintainable semantics of the language are different. The former requires modularity of the domain-level models expressed in the KAOS language. The latter requires modularity of the meta-level description of the KAOS language.

4.1.2. Choosing a Semantic Domain

Like for any language, the semantics of the KAOS language is given by a translation of every KAOS model into some other target language, called the *semantic domain* of the language. For the semantics to be useful, the semantic domain should be composed of simpler, more primitive constructs than the original language. The purpose of the original language is to provide syntactical constructs that facilitates the description of models at the semantic level.

The choice of an appropriate semantic domain is a critical concern in defining a semantics. The semantic domain may range from a very simple and general domain such as the standard models of first-order logic (FOL) to more complex and specialized domains.

This choice of semantic domain has an impact on the understandability of the semantics, the definition of reasoning procedures associated with the language, and the capability to define multi-paradigm languages. These concerns are discussed in turn.

Each semantic domain reflects a particular ontology, i.e. a particular view of the world. For instance, the standard model of FOL views the world as being composed of a universe of individuals and of relations between individuals. Such a model is simple and easy to understand. However, the mapping from the KAOS language to this model may become fairly complex. The assertions in FOL resulting from such mapping may become large and incomprehensible. On the other hand, a specialized semantic domain with a richer ontology is more complex, but the semantic function is simpler. The case for specialized semantic models reflecting the ontology of the language in a closed way has been articulated in [Par95], which calls for semantic domains describing the content of

specification documents independently from the particular notations used for describing such content. The 4-variable model is proposed there as the semantic domain for describing requirements. The SCR tabular notation [Hen80, Heit96] is one notation that can be used to describe 4-variable models. We believe that the semantics of the KAOS language would be easier to understand if we can define it over a specialized semantic domain that reflects the underlying ontology of the KAOS language.

A semantic domain also provides a basis on which to define reasoning procedures. An advantage of defining the semantics of a language in FOL is that the logic has well-known reasoning procedures. In theory, such reasoning procedures could then be used to reason about a specification at the semantic level. In practice, however, this approach is intractable, as the formulas in FOL resulting from the translation function may be too complex to handle; and *the result of the analysis performed in FOL is difficult to translate back into the original language*. More specialized semantic domains on the other hand make it possible to define more specialized and efficient reasoning procedures. Model-checking for propositional temporal logic formula is an example of such a procedure. The completeness and consistency checking techniques for RSML and SCR models [Heim96, Heit96] are other examples of powerful analysis techniques made possible by defining the semantics of these languages over specialized semantic domains, namely, state-machine models.

The choice of a semantic domain may also have an important impact on how the semantics of different languages can be combined to form a multi-paradigm language. A first approach consists in defining the semantics of various languages over a common semantic domain. Such a semantic domain must be very general in order to be able to model constructs from a wide range of languages. [Zav93] shows the feasibility and limits of this approach by defining and relating the semantics of various specification languages in FOL. A second approach consists in defining separate semantics for the different languages over separate, specialized semantic domains; and combine these semantics through inter-model consistency rules. The latter approach requires consistency rules to be defined between each pair of languages. It may not be appropriate for combining specifications written in many different languages. We are however interested in defining the semantics of a language which combines a few paradigms addressing orthogonal aspects of the system. The second approach is appealing as inter-model consistency rules correspond to the meta-constraints linking components of the KAOS meta-model.

4.1.3. Overview of a semantics for the KAOS language

To summarize, our approach for defining the semantics of the KAOS language is based on the following decisions:

- The semantics is structured according to the multi-paradigm structure of the language. A separate semantics is defined for each KAOS submodel, namely, the goal model, the object model, the operation model, the agent responsibility model, and the agent interface model.
- Each KAOS submodel is defined over a specialized semantic domain that describes the ontology of the model.
- The different models are combined through inter-model consistency rules at the semantic level. These semantic-level consistency rules yield language-level consistency rules which correspond to the meta-constraints over components of the meta-model.

To reflect the language structure, the semantics of the KAOS language will furthermore be composed of two parts.

A first part defines the outer layer of the language. It is composed of separate semantics for the goal model, the object model, the agent model, and includes consistency rules between those models.

A second part defines the syntax and semantics of the logical formalism used at the inner layer of the KAOS language; and includes consistency rules between the formal assertion layer and the declaration layer of KAOS models.

In the thesis, we are mainly concerned with the semantics of agent responsibility, monitoring, and control, together with the semantics of the operation model. The formal model of agents defined in the next section is intended to provide the semantic domain for these language constructs.

4.2. The Underlying Agent Model

This section defines the semantic domain for agents in the KAOS language. The mapping of constructs of the KAOS language to this semantic domain is also outlined.

Intuitively, an agent is characterized by the following items:

1) an *interface* which declares a set of state variables that the agent monitors, and a set of state variables that the agent controls;

2) a *transition system* which is composed of an initial condition on controlled states, and a "next state" relation mapping each sequence of monitored states to a next state of controlled variables; and

3) a set of *goals* the agent is responsible for.

The concepts of agent interface, transition system and responsibility assignments of goals to agents are formally defined in the following sections.

As a specialization of object, an agent also has attributes and links to other objects. Such features are part of the semantics of the object model.

Our model of agents is based on states, and interaction between agents are through shared state variables. An alternative paradigm would be to specify agent interactions in terms of shared actions.

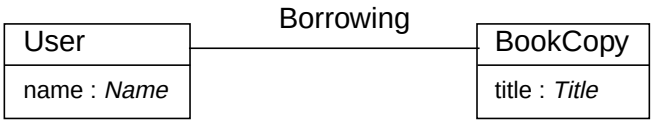
4.2.1. Preliminary Definitions: State Variables, States and Histories

We first recall the basic concepts of state variables, states, and histories [Man92].

State Variables

We assume a set VAR of all possible state variables symbols. We also assume that each state variable v has a type which is noted $\mathsf{TY(v)}$. The vocabulary of an application domain is given by a set $\mathsf{V} \subseteq \mathsf{VAR}$.

State variables correspond to attributes of object instances in the KAOS object model. As an example, consider the following partial object model for a library system.



The set of state variables corresponding to this object model is composed of:

u.name
with TY(u.name) = *Name* for every possible instance u of the User entity

bc.title
with TY(bc.title) = *Title* for every possible instance bc of the BookCopy entity

Borrowing(u,bc)
with TY(Borrowing(u,bc)) = Bool
for every possible instance u of the User entity,
and every possible instance bc of the BookCopy entity.

In the sequel, the concepts of the underlying agent model are illustrated with examples from the mine pump control system (see Section 3.3.2 of Chapter 3) for which we consider the following set of variables:

WaterLevel: Depth	the actual water level in the sump of the mine
HighWaterSignal: {On, Off}	the signal sent by the high water sensor
LowWaterSignal: {On, Off}	the signal sent by the low water sensor
PumpSwitch: {On, Off}	the position of the switch that commands the pump
PumpMotor: {On, Off}	the status of the pump motor

State

Let $V \subseteq VAR$ be a set of state variables symbols. A *state* s of V is an interpretation function for the variables in V, that is, it is a function

$s: V \rightarrow TY(V)$.

The set of all possible states of V is noted State(V).

An example of state s1 for the above set of state variables of the mine pump control system is given by:

s1(WaterLevel) = 7.2

s1(HighWaterSignal) = On

s1(LowWaterSignal) = On

s1(PumpSwitch) = On

s1(PumpMotor) = On

Histories and Paths

A *history* over a set of variables V is an infinite sequence of states of V, modelled as a function

$h: Nat \rightarrow State(V)$

The set of all histories over a set of variables V is noted History(V).

In order to model real-time properties, sequences of states are extended with real-time tags as defined in Section 3.2.3 of Chapter 3; there is a function

$time: Nat \rightarrow Time$

which assigns a real-time value to each position $i \in Nat$. The function time is defined as follows:

time(0) is the time at the initial position of every history

$time(i) = time(0) + \delta\, i$, where δ is the time elapsed between successive states.

In the sequel, we are also interested by partial histories, i.e., by finite sequences of states. A *path* over a set of variables V is a finite or infinite sequence of states of V. The set of all paths over a set of variables V is noted Path(V).

The following notations for sequences are used. Let σ be a path, and let i be a natural number such that $i \leq length(\sigma)$,

$\sigma(i)$ denotes the state of σ at position i,

$\sigma[i]$ denotes the prefix of σ up to position i,

$\sigma+s$ denotes the concatenation of sequence σ with state s.

As mentioned in Chapter 3, a goal defines a set of histories. The fact that an history h satisfies a goal G is noted

$h \models G$.

We also use the notation

$\sigma \models G$

to denote that the path σ satisfies the goal G. This satisfaction relation is formally defined as follows:

$\sigma \models G$ iff there exists an infinite suffix h of σ such that $h \models G$.

As an example, if G is a state invariant $\Box P$ where P is a state-formula, $\sigma \models \Box P$ is true iff $\sigma(i) \models P$ for all $i \in [0 \ldots length(\sigma)]$.

4.2.2. Agent Interface

Agent interfaces in the underlying agent model are formally defined as follows.

Definition (Agent Interface) -- An agent interface model Γ over a set V of variables is a tuple composed of the following items:

- a set AGENT of agent *instances*;

and for each $ag \in AGENT$, a signature SIGN(ag) composed of the following items

- a set $Mon(ag) \subseteq V$ of variables monitored by the agent;
- a set $Ctrl(ag) \subseteq V$ of variables controlled by the agent;

We use the notation Voc(ag) to denote the union of Mon(ag) and Ctrl(ag).

An agent signature must furthermore satisfy the following constraints:

(1) for each agent, the sets of monitored and controlled variables are disjoint, i.e.

$Mon(ag) \cap Ctrl(ag) = \phi$.

(2) every variable is controlled by at most one agent, i.e.

if $ag \neq ag'$ then $Ctrl(ag) \cap Ctrl(ag') = \phi$.

The second constraint is used when defining transitions of agents in order to avoid interference between concurrent executions of agents.

For example, an agent interface model for the mine pump control system is given by:

Agent = {pump_ctrler, high_sensor, low_sensor}

Mon(pump_ctrler) = {HighWaterSignal, LowWaterSignal}
Ctrl(pump_ctrler) = {PumpSwitch}

Mon(high_sensor) = {WaterLevel}
Ctrl(high_sensor) = {HighWaterSignal}

Mon(low_sensor) = {WaterLevel}
Ctrl(low_sensor) = {LowWaterSignal}

Here, the agents pump_ctrler, high_sensor, low_sensor are agent instances.

The semantics of KAOS agent interface models is defined over agent signatures by a function that maps instance declarations of monitoring and control links of the KAOS model to agents signatures of the underlying agent model.

In the thesis, we make the simplifying assumption that agent interfaces are static.

Note that in our model, agent interfaces are composed of *monitored and controlled state* variables. With an event-based formalism, agent interfaces would be composed of *monitored and controlled events*. In our model, the occurrence of an event is viewed as a state variable that holds at a single point in time. (Remember that a KAOS event is a special kind of object; see Section 3.2.5 in Chapter 3). Interaction through shared events is therefore also supported by our model.

4.2.3. Agent Views and Indistinguishability

Agent interfaces define the views that agents have on the system. If $s \in State(V)$ is a global state of the system, the *view of an agent* on the global system state is given by the projection $s_{|Voc(ag)}$ of s on $Voc(ag)$.

We use the notation

$$s \sim_{Voc(ag)} s'$$

to express that two states are *indistinguishable* by an agent; we have thus:

$$s \sim_{Voc(ag)} s' \quad \text{iff} \quad s_{|Voc(ag)} = s'_{|Voc(ag)}$$

As an example, consider the agent interface model above and two states $s1$ and $s2$ such that

s1(WaterLevel) = 7.2	s2(WaterLevel) = 8
s1(HighWaterSignal) = On	s2(HighWaterSignal) = On
s1(LowWaterSignal) = On	s2(LowWaterSignal) = On
s1(PumpSwitch) = On	s2(PumpSwitch) = On
s1(PumpMotor) = On	s2(PumpMotor) = Off

The two states are indistinguishable by the pump_ctrler agent, because the variables at its interface all have the same values in the two states.

The notions of agent views and indistinguishability are extended to sequences of states as follows

$\sigma_{|Voc(ag)}$ = the projection of every state of σ on $Voc(ag)$,

$\sigma \sim_{Voc(ag)} \sigma'$ iff $\sigma_{|Voc(ag)} = \sigma'_{|Voc(ag)}$.

Similarly, the set of variables controlled by an agent defines a projection on global states of the system and an equivalence relation between global states of the system:

$s_{|Ctrl(ag)}$ = the projection of s on $Ctrl(ag)$,

$s \sim_{Ctrl(ag)} s'$ iff $s_{|Ctrl(ag)} = s'_{|Ctrl(ag)}$.

4.2.4. Agents Transition Systems

The transition system of agents is defined as follows.

Definition (Agent Transition System) -- A multi-agent transition system Π is a tuple composed of the following items:

- an agent interface model $\Gamma = <AGENT, Sign>$;
- for each $ag \in AGENT$, a transition system $\Delta(ag)$ composed of the following items:
  - a set
    $Init(ag) \subseteq State(Ctrl(ag))$
    of initial states for the variables controlled by the agent;

- a "next state" relation
  $Next(ag) \subseteq Path(Voc(ag)) \times State(Ctrl(ag))$
  that relates sequences of states of variables in the signature of the agent to a next state of variables controlled by the agent.

The transition system of an agent is furthermore submitted to the following constraints:

(1) the set of initial states of an agent is not empty, i.e,

$Init(ag) \neq \phi$

(2) the Next relation is total, i.e.,

for all $\sigma_m \in Path(Voc(ag))$, there exists $s_c \in State(Ctrl(ag))$
such that $<\sigma_m, s_c> \in Next(ag)$

Even for small systems, it would be too long to define the Next state relation in extension by listing all the pairs $<\sigma_m, s_c> \in Next(ag)$. Specification languages are introduced for defining this relation concisely. They can take different forms: tabular notations in the spirit of SCR [Hen80, Heit96], transition diagrams in the spirit of [Har87] and [Lev94], or state-based style specifications in the spirit of Z [Spi89], VDM [Jon90] or B [Abr96].

We will use the KAOS operation model to define the underlying multi-agent transition system. The semantics of KAOS operation models is defined by a relation between these models and the underlying transition systems defined here.

4.2.5. Agent Runs

An agent run is a sequence of states generated by the transition system of the agent. Agent runs are formally defined as follows.

Definition (Runs of an Agent Transition System) -- Let $\Pi = <\Gamma, \Delta>$ be a multi-agent transition system.

(a) The runs of an agent $ag \in AGENT$ is a set

$Run(ag) \subseteq Path(V)$

such that $\sigma \in Run(ag)$ iff it satisfies the following constraints:

- satisfaction of the initial condition:
  $\sigma(0)_{|Ctrl(ag)} \in Init(ag)$
- satisfaction of the Next relation:
  $< \sigma[i-1]_{|Voc(ag)}, \sigma(i)_{|Ctrl(ag)} > \in Next(ag)$ for all $i \in [1..\ length(\sigma)]$

The set of infinite runs of the agent is noted Behaviour(ag).

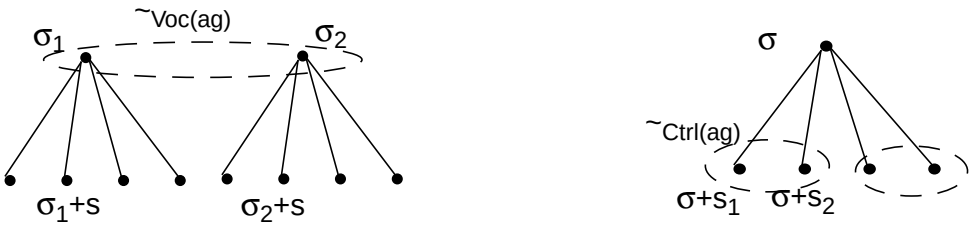
(b) The runs of the multi-agent system is a set

$Run(\Pi) \subseteq Path(V)$

such that $\sigma \in Run(\Pi)$ iff $\sigma \in Run(ag)$ for all $ag \in AGENT$.

Note that the runs of an agent in these definitions are histories on global system states, rather than histories on state of variables in the signature of the agents.

Also note that the transitions of agents occur concurrently. Interference between agents is avoided because each variable is controlled by at most one agent.



(a) interface restriction (b) control restriction

FIGURE 4.1. Two properties of agent runs

In the previous section, we required the Next relation to be a total relation. This ensures that every finite run can be extended into an infinite run.

4.2.6. Properties of Agent Runs

We now describe three important properties of agent runs. They will be used later to define necessary and sufficient conditions for a goal to be realizable by an agent. The full proofs of the properties are given in Appendix A.

(a) The domain of the Next state relation of an agent is restricted to the view the agent has on the global system states. As a consequence, the set of runs of an agent is constrained by the interface of the agent. Suppose that the system is at a point σ_1 and that $\sigma_1 \sim_{Voc(ag)} \sigma_2$ (see Figure 4.1.a). If $<\sigma_{1|Voc(ag)}, s_{|Ctrl(ag)}> \in Next(ag)$, then since $\sigma_1 \sim_{Voc(ag)} \sigma_2$, we also have that $<\sigma_{2|Voc(ag)}, s_{|Ctrl(ag)}> \in Next(ag)$. Thus, if $\sigma_1+s \in Run(ag)$ then $\sigma_2+s \in Run(ag)$. Therefore, the following property characterizes agent runs.

Property 1 (interface restriction) -- For all $\sigma_1, \sigma_2 \in Run(ag)$, $s \in State(V)$,
if $\sigma_1 \sim_{Voc(ag)} \sigma_2$ then $\sigma_1+s \in Run(ag)$ iff $\sigma_2+s \in Run(ag)$

(b) The second property is related to the fact that the range of the Next state relation of an agent is restricted to the set of variables controlled by the agent. Suppose that the system is at a point σ, and that there are two states s1 and s2 such that $s1 \sim_{Ctrl(ag)} s2$ (see Figure 4.1.b). If $<\sigma_{|Voc(ag)}, s1_{|Ctrl(ag)}> \in Next(ag)$, then since $s1 \sim_{Ctrl(ag)} s2$, we also have that $<\sigma_{|Voc(ag)}, s2_{|Ctrl(ag)}> \in Next(ag)$. Thus, if $s_1 \sim_{Ctrl(ag)} s_2$ then $\sigma+s_1 \in Run(ag)$ iff $\sigma+s_2 \in Run(ag)$. This is formally captured by the following property.

Property 2 (control restriction) -- For all $\sigma \in Run(ag)$, $s_1, s_2 \in State(V)$,
if $s_1 \sim_{Ctrl(ag)} s_2$ then $\sigma+s_1 \in Run(ag)$ iff $\sigma+s_2 \in Run(ag)$

The third property states that the specification of an agent transition system is violable in a finite time. This is formally captured by the following property.

Property 3 (finitely violable) -- For all $h \in History(V)$,
if $h \notin Behaviour(ag)$ then there exists a finite prefix σ of h such that $\sigma \notin Run(ag)$

The property asserts that the set of histories generated by a transition systems define a safety property according to the classical safety/liveness distinction of temporal logics [Alp87]. It is a consequence of our definition of transition systems that does not include fairness conditions.

4.2.7. Agent Responsibilities

The agent responsibility model is defined as follows.

Definition (Agent Responsibility Model) -- An agent responsibility model is given by the following items:

- a set AGENT of agent instances
- a set GOAL of goals, where each $G \in GOAL$ defines a set of histories on global system states;
- a relation $Resp \subseteq AGENT \times GOAL$.

As an example, consider the goal defined by

HighWaterSignal = 'On' $\Rightarrow \bigcirc$ PumpSwitch = 'On'.

The responsibility assignment of that goal to the pump_ctrler agent is declared by:

Resp(pump_ctrler, HighWaterSignal = 'On' $\Rightarrow \bigcirc$ PumpSwitch = 'On').

The responsibility relation of the underlying agent model is derived from the instance declarations of the Responsibility links in the KAOS model.

The Resp relation of the underlying agent model is a primitive relation. The responsibility assignments of goals to agents are purely syntactical relations between goals and agents. The semantics of responsibility assignments is captured in the next section by a responsibility consistency rule between the transition system of agents and their responsibility assignments.

4.2.8. Relating agent responsibilities and the agent's transition system

The semantics of responsibility assignment of a goal to an agent is that the agent must restrict its behaviour so as to ensure the goal [Fea87]. The semantics is captured by the following consistency rule relating the transition system of an agent to its responsibility assignments.

Definition (Responsibility consistency rule) -- Given an agent transition system and an agent responsibility model, the following constraint must be satisfied:

for all $ag \in AGENT, G \in GOAL$

if Resp(ag, G), then $Behaviour(ag) \subseteq G$.

Note that this formal notion of responsibility does not say that if the behaviours of an agent satisfy a goal then the agent is responsible for that goal (an agent could satisfy a property without being required to satisfy it). This consistency rule is therefore not a formal definition (in the strict mathematical sense) of responsibility (that is, a statement that defines responsibility assignments in terms of other more primitive concepts). It is however a formal property that captures the meaning of responsibility assignments.

The responsibility consistency rule yields meta-constraints at the language level. In order to satisfy the responsibility consistency rule at the semantic level, a responsibility assignment at the language level must be operationalized by a set of operational requirements that satisfy the responsibility meta-constraint (see Section 3.2.7 in Chapter 3).

4.3. Defining Realizability

The *realizability* meta-constraint is intended to provide a precise criterion for identifying whether a goal can be assigned as the responsibility of an agent based on its monitoring and control capabilities, *before an operational model of the agent is available*.

As mentioned in Chapter 3, realizability plays a significant role in the goal-driven requirement elaboration process. It provides a criterion for stopping the goal refinement process (a goal realizable by an agent need not to be further refined), and it guides the refinement of goals into subgoals until the latter are realizable by individual agents.

This section is structured as follows. Section 2.1. gives a formal definition of realizability for a single goal. Section 2.2. gives necessary and sufficient conditions for a goal to be realizable. These conditions are defined at the semantic level on the set of histories admitted by the goal. Section 2.3 defines realizability for multiple goals.

4.3.1. Defining Realizability of single responsibility assignments

The realizability meta-constraint is intended to capture what are admissible responsibility assignments of goals to agents based on the agent's monitoring and control capabilities.

We say that a goal is realizable by an agent if, given the agent monitoring and control capabilities, there exists a transition system for the agent such that the behaviour of the agent is *equal* to the set of histories admitted by the goal.

Definition (Realizability) -- Let ag be an agent instance with interface variables Mon(ag), Ctrl(ag), and let G be a goal. The goal G is *realizable* by an agent ag
iff there exists a transition system Δ(ag) = <Init(ag), Next(ag)> with

- Init(ag) $\subseteq$ State(Ctrl(ag)
- Next(ag) $\subseteq$ Path(Voc(ag)) $\times$ State(Ctrl(ag))

such that Behaviour(ag) = G.

This definition of realizability requires the existence of an appropriate transition system for the agent. In the following section and in Chapter 5, we will see how one can establish whether a goal is realizable or not by an agent from the monitoring and control capabilities of the agent, without referring, implicitly or explicitly, to the transition system of the agent.

Note that the definition of realizability requires the existence of a transition system for the agent such that Behaviour(ag) = G, instead of simply requiring that Behaviour(ag) $\subseteq$ G. Intuitively, this means that the agent should be able to satisfy the goal *without being more restrictive than required by the goal*. This will be illustrated by the third example below.

Let us first consider an example of a realizable goal. Consider the goal Maintain[PumpSwitchOnWhenHighWaterDetected], defined by:

HighWaterSignal='On' $\Rightarrow$ ❍ PumpSwitch = 'On'.

The goal is realizable by the pump_ctrler agent that monitors the HighWaterSignal variable and controls the PumpSwitch variable. A transition system for the pump_ctrler whose set of behaviours is equivalent to the goal is given by the pair <Init, Next> such that

(i) $s_c \in$ Init for all $s_c \in$ State(Ctrl(pump_ctrler))

(ii) $<\sigma_m, s_c> \in$ Next iff if σ_m (n)(HighWaterSignal) = On (where n= length(σ_m))
then s_c(PumpSwitch) = On

As a first example of an unrealizable goal, consider the goal Maintain[PumpOnWhenHighWater], defined by:

WaterLevel ≥ 'High' ⇒ ❍ PumpMotor = 'On'.

Note that the pump_ctrler agent does not monitor the WaterLevel variable and does not control the PumpMotor variable. As a result, it is impossible to find a transition system for the pump_ctrler such that Behaviour(pump_ctrler) = PumpOnWhenHighWater. (A formal proof that such a transition system does not exist is given in the following section.) The goal is not realizable by the pump_ctrler and should therefore not be assigned to that agent.

To illustrate why we require equality in the definition of realizability, consider the goal Maintain[PumpSwitchOnWhenHighWater], defined by:

WaterLevel ≥ 'High' ⇒ ❍ PumpSwitch = 'On'.

The pump_ctrler agent controls the PumpSwitch variable, but does not monitor the WaterLevel variable. The agent can ensure the goal by always keeping the PumpSwitch variable set to 'On', regardless of the value of the WaterLevel variable. More explicitly, one can show that the following transition for the pump_ctrler system satisfies the goal:

(i) $s_c \in$ Init for all $s_c \in$ State(Ctrl(pump_ctrler))

(ii) $<\sigma_m, s_c> \in$ Next iff s_c(PumpSwitch) = On

The behaviours generated by this transition system are defined by:

❍ ❑ PumpSwitch = 'On'

These behaviours are stronger than required by the goal. The goal is actually not realizable by the pump_ctrler, and cannot be assigned to that agent. (We will show in the next section that there is no transition system for the pump_ctrler such that Behaviour(pump_ctrler) = PumpSwitchOnWhenHighWater.)

Therefore, the definition of realizability prevents the assignment of a goal to an agent if *the agent does not have the monitorability and control capabilities to satisfy the goal without being more restrictive than required by the goal*.

Note that realizability requires that is must be *possible* for an agent to satisfy a goal without being more restrictive than required by the goal; but that when several goals are assigned to an agent, the actual transition system of the agent may be stronger than required by any single goal.

This definition of realizable goal can be viewed as the equivalent at the goal level of the concept of realizable specification: a specification is said to be realizable if there exists a program that implements it [Aba89]. There are however two important differences: (i) our concept of realizability explicitly refers to the variables monitored and controlled by

the agent, and (ii) we require the existence of a transition system whose behaviours are *equal* to the set of histories admitted by the goal, whereas only inclusion is required in [Aba89].

4.3.2. Semantic Conditions for Realizability

To show that a goal G is realizable by an agent ag, one can exhibit a transition system Δ(ag) such that Behaviour(ag) = G. In this section, we are interested in the dual problem of showing that a goal is *not* realizable by an agent. A brute force approach would consist in generating every possible agent transition system and showing for every one of them that Behaviour(ag) $\neq$ G. This approach is clearly not feasible. This motivates the introduction of the following theorem. It states three necessary and sufficient conditions for a goal to be realizable by an agent. These conditions are defined at the semantic level on the set of paths admitted by the goal. (Syntactical conditions for identifying unrealizable goals will be considered in Chapter 5.) The limitations of the theorem will be discussed next.

Theorem 1 (Semantic Conditions for Realizability) -- For all G $\subseteq$ Hist(V) such that G $\neq \phi$, G is realizable by an agent ag iff the following conditions hold:

(1) for all $\sigma_1, \sigma_2 \in$ Path(V), s $\in$ State(V),
if $\sigma_1 \sim_{Voc(ag)} \sigma_2$ then σ_1+s |=G iff σ_2+s |= G

(2) for all $\sigma \in$ Path(V), $s_1, s_2 \in$ State(V),
if $s_1 \sim_{Ctrl(ag)} s_2$ then $\sigma + s_1$ |= G iff $\sigma + s_2$|= G

(3) for all h $\in$ History(V)
if h |$\neq$G then there exists a finite prefix σ of h such that σ |$\neq$ G

The proof of the theorem is given in appendix A.

We illustrate the three conditions of the theorem by showing how they can be used to show that a goal is not realizable by an agent. Since the conditions are necessary and sufficient, every goal that is not realizable by an agent violates at least one of the conditions.

1. Goals violating the first condition

Consider the first condition. One can show that a goal is not realizable by an agent by showing that the goal violates the first condition, i.e. by giving σ1, σ2 $\in$ Path(V), and s $\in$ State(V) such that

$\sigma_1 \sim_{Voc(ag)} \sigma_2$ and σ_1+s |= G and σ_2+s |$\neq$ G

Example 1. Consider the goal

WaterLevel $\geq$ 'High' $\Rightarrow$ ❍ PumpSwitch = 'On'

We show that the goal is not realizable by the pump_ctrler agent because it violates the first condition of Theorem 1.

Let us take two paths σ_1, σ_2, such that n = length(σ_1) = length(σ_2), and defined as follows:

σ_1 (i) = σ_2 (i) for i ∈ [0 ... n - 1]

σ_1 (n) (WaterLevel) < 'High'

σ_2 (n) (WaterLevel) ≥ 'High'

σ_1 (n) (v) = σ_2 (n) (v) for all v ≠ s.WaterLevel

Since WaterLevel ∉ Voc(pump_ctrler), we have that $\sigma_1 \sim_{Voc(ag)} \sigma_2$. If we now take s such that

s(PumpSwitch) = 'Off',

we have that σ_1+s |= G and σ_2+s |≠ G. The first condition of the theorem is therefore violated, and we have shown that the goal is not realizable by the pump_ctrler.

This goal violates the first condition of Theorem 1 because it refers to the variable WaterLevel that is not at the interface of the pump_ctrler.

2. Goals violating the second condition

Consider the second condition. One can violate the second condition by giving σ ∈ Path(V), and s_1, s_2 ∈ State(V) such that

$s_1 \sim_{Ctrl(ag)} s_2$ and σ +s_1 |= G and σ +s_2|≠ G.

We give three examples of goals that violates the second condition of Theorem 1. Each of these examples shows a different cause of violation of this condition.

Example 2. Consider the goal

HighWaterSignal = 'On' ⇒ ❍ PumpMotor = 'On'.

We show that the goal is not realizable by the pump_ctrler agent because it violates the second condition of Theorem 1.

Let us take a path σ of length n such that σ has not violated G, i.e. σ |= G, and

σ (n) (HighWaterSignal) = On.

We now take s1 and s2 such that

s1(PumpMotor) = 'On' , s2(PumpMotor) = 'Off'

s1(v) = s2(v) for all v ≠ PumpMotor

Since PumpMotor ∉ Ctrl(pump_ctrler), we have that $s_1 \sim_{Ctrl(ag)} s_2$. We also have that σ +s_1 |= G and σ +s_2|≠ G. The second condition of the theorem is therefore violated, and we have shown that the goal is not realizable by the pump_ctrler.

This goal violates the second condition of Theorem 1 because it refers to the variable PumpMotor that is not controlled by the pump_ctrler.

Example 3. As an example of goal violating the second condition for a different reason, consider a goal defined by:

$\Box_{\leq d}$ HighWaterSignal = 'On' $\Rightarrow \bigcirc$ PumpSwitch = 'On'.

This goal is defined only in terms of variables monitored and controlled by the pump_ctrler agent. Intuitively, that goal is not realizable by the pump_ctrler agent because it *refers to the future* values of the variable HighWaterSignal. One can show that this goal also violates the second condition of Theorem 1 as follows.

Let us take a path σ of length n, defined by:

σ (i) (HighWaterSignal) = Off for all i such that $0 \leq i \leq n - d$

σ (i) (HighWaterSignal) = On for all i such that $n - d < i \leq n$

σ (i) (PumpSwitch) = Off for all i such that $0 \leq i \leq n$

together s1, s2 such that

s1(HighWaterSignal) = 'Off', s2(HighWaterSignal) = 'On'

s1(v) = s2(v) for all $v \neq$ HighWaterSignal

Since HighWaterSignal $\notin$ Ctrl(pump_ctrler), we have that $s_1 \sim_{Ctrl(ag)} s_2$.

Note that the assertion $\Box_{\leq d}$ HighWaterSignal = On never holds for the path σ+s1. Therefore, we have that $\sigma + s_1 \models G$. However, for the path σ+s2, we have:

σ+s2 (n-d+1) $\models \Box_{\leq d}$ HighWaterSignal = On

σ+s2 (n-d+1) $\not\models$ PumpSwitch = On

Therefore, $\sigma + s_2 \not\models G$; and we have shown that the goal is not realizable by the pump_ctrler.

Example 4. As a last example of goal violating the second condition of Theorem 1, consider the meeting scheduling problem and the goal:

m.PlanningRequest
$\Rightarrow \bigcirc$ ($\exists$ d: Date):
m.Date = d $\wedge$ ($\forall$ p: Prtcpt): $\bullet$ Intended(p,m) $\rightarrow$ d $\notin$ $\bullet$ Cstr[p,m].exclset

For the sake of the example, assume that a Scheduler agent is capable of controlling the date of the meeting, and is capable of monitoring the planing requests and exclusion sets of participants. One can show that this goal is not realizable by the Scheduler agent, because it violates the second condition of Theorem 1.

Let us take a path σ of length n defined as follows:

σ (i) (m.PlaningRequest) = false for all i such that. $0 \leq i \leq n$

and choose s1, s2 such that

s1 $\not\models$ m.PlaningRequest

s2 $\models$ m.PlaningRequest
$\wedge \neg$ ($\exists$ d: Date): ($\forall$ p: Prtcpt): $\bullet$ Intended(p,m) $\rightarrow$ d $\notin$ $\bullet$ Cstr[p,m].exclset

Since the state variables m.PlaningRequest, Intended(p,m), and Cstr[p,m].exclset are not controlled by the Scheduler agent, we have that $s_1 \sim_{Ctrl(ag)} s_2$. We also have that $\sigma + s_1 \models G$ and $\sigma + s_2 \not\models G$, because from $\sigma + s_2$ there is no next state that satisfies the goal. The second condition of the theorem is therefore violated, and we have shown that the goal is not realizable by the Scheduler agent.

In this example, the goal is defined only in terms of variables monitored and controlled by the agent, and does not refer to the future values of monitored variables. Intuitively, the goal is not realizable by the agent because there exists a behaviour of the agent's environment that makes the goal impossible to satisfy. The specialized condition, called *unsatisfiability,* characterizing this kind of realizability problem will be defined in chapter 5.

3. Goals violating the third condition

The third condition of Theorem 1 is violated by giving an infinite history h such that

$h \not\models G$

$\sigma \models G$ for all finite prefix σ of h.

Example 5. As an example of goal violating the third condition, consider a goal defined by:

$Request \Rightarrow \Diamond\, Service$

We show that this goal violates the third condition as follows.

Consider an history h such that

$(h, i) \models Request$ for some $i \geq 0$, and

$(h, j) \not\models Service$ for all $j \geq i$.

Clearly, $h \not\models G$. However, any finite prefix σ of h satisfies the goal because for any finite prefix σ of h, there exists a suffix h' of σ such that $h' \models G$. That is, the goal cannot be violated in a finite time.

According to the third condition, a "pure" liveness property (as classically defined) cannot be assigned as the responsibility of an agent. The intuition for such restriction is that a liveness property does not really constrain the behaviours of the agent: the agent can indefinitely postpone the satisfaction of the goal without infringing on its responsibility. A violation of a liveness property can also never be observed in the running system.

Even though a liveness property cannot be assigned as the responsibility of a single agent, a goal on the global system can be a liveness property whose satisfaction can be achieved through the cooperation of several agents.

Also note that bounded Achieve goals of the form

$R \Rightarrow \Diamond_{\leq d}\, S$

are not liveliness properties in the classical sense. If the agent is responsible for satisfying any request within 2 days, the goal is violated as soon as a request has not been satisfied during two days.

4. Practical Limitations of Theorem 1

Theorem 1 defines necessary and sufficient conditions that allow one to prove that a goal is not realizable by an agent. However, the theorem does not provide efficient guidance for identifying realizability problems during the requirements elaboration process:

- proving a violation of one of the conditions of the theorem requires *tedious reasoning at the semantic level* on the set of paths admitted by the goal;
- the fact that a goal violates one of the conditions of the theorem does not provide sufficient explanation about *why* the goal is not realizable by the agent (see examples 3 and 4 above).

In Chapter 5, we will provide a complete taxonomy of realizability problems that (i) allows one to identify realizability problems syntactically form the formal definition of goals, and (ii) explain why a goal is not realizable, thereby providing guidance for elaborating the requirements model so as to resolve the cause of the realizability problem.

4.3.3. Defining Realizability of multiple responsibility assignments

Up to now, we have considered the notion of realizability for a *single* goal. We extend the concept of realizability to multiple goals as follows.

Definition (Realizability of multiple goals) -- A set of goals $\{G_1,..., G_n\}$ is *realizable* by an agent ag iff there exists a transition system $\Delta(ag) = \langle Init(ag), Next(ag)\rangle$ with

- $Init(ag) \subseteq State(Ctrl(ag))$
- $Next(ag) \subseteq Path(Voc(ag)) \times State(Ctrl(ag))$

such that

$$Behaviour(ag) = \bigcap G_i.$$

It is worth pointing out that if two goals are realizable separately, they may not be realizable together. As an example, consider again the goal Maintain[PumpSwitchOnWhenHighWaterDetected], defined by:

$$HighWaterSignal='On' \Rightarrow \bigcirc PumpSwitch = 'On'$$

and the goal Maintain[PumpSwitchOffWhenCriticalMethanMeasure]:

$$MethaneMeasure \geq 'Critical' \Rightarrow \bigcirc PumpSwitch = 'Off'$$

where MethaneMeasure is monitored by the pump_ctrler. Each of the two goals is separately realizable by the pump_ctrler. However, the two goals are not realizable together by the pump_ctrler. Indeed, there is no transition system for the agent that satisfies both goals. Since the variables HighWaterSignal and MethaneMeasure are not controlled by the pump_ctrler, this agent cannot prevent the system from reaching a state s for which

$$s(HighWaterSignal) = On$$

$$s(MethaneMeasure) \geq 'Critical'.$$

In such a state, there is no next state s' that satisfies both goals: the first goal requires the next state to satisfy

s'(PumpSwitch) = 'On',

whereas the second goal requires the next state to satisfy

s'(PumpSwitch) = 'Off'.

4.4. Summary

This chapter has defined a foundation on which the techniques presented in the following chapters are built. We have defined a formal model of agents that provides the underlying semantic domain for the KAOS language features related to the concept of agent. We have formally defined the realizability meta-constraint that relates the responsibility of an agent to its monitoring and control capabilities. We also identified necessary and sufficient conditions allowing unrealizability to be identified by reasoning at the semantic level on the set of paths admitted by the goal. These conditions however do not provide practical support for reasoning about realizability during the requirements elaboration process; they require tedious reasoning at the semantic level, and do not provide sufficient explanation about the cause of unrealizability. The purpose of the next chapter is to define checkable, syntactical conditions for identifying and classifying realizability problems during the goal refinement process.

Chapter 5
Identifying and Classifying Unrealizable Goals

The objective of this chapter is to propose systematic techniques for supporting the identification of realizability problems during the goal refinement process.

For that purpose, we define a taxonomy of unrealizability conditions that

- provides *checkable syntactical conditions* for identifying unrealizable goals during the goal refinement process;
- explains *why the goal is not realizable*, to provide guidance for elaborating the model so as to resolve the cause of unrealizability.

In brief, a goal is not realizable by an agent for at least one of the following reasons:

- **lack of monitorability**: the goal is defined in terms of variables that are not in the interface of the agent;
- **lack of control**: the goal requires control of some variables that are not controlled by the agent;
- **reference to future**: the goal constrains the values of controlled variables in terms of future values of monitored variables;
- **goal unsatisfiability**: there exists a behaviour of the agent's environment that makes the goal impossible to satisfy;
- **not finitely violable goal**: the goal does not constrain the finite runs of the agent, i.e. it defines a liveness property.

The conditions of unrealizability are precisely defined in the following sections. We also show that the taxonomy is complete, that is, every goal that is not realizable by an agent satisfies at least one of the above conditions.

5.1. Viewing Goals as Relations

The taxonomy of realizability problems is intended to provide effective explanations of why a goal is not realizable. In order to provide such explanation, it is useful to identify what are the variables that are *intended to be constrained* by the goal.

As an example, consider the goal Maintain[PumpOnWhenHighWater] defined by:

$$\mathsf{WaterLevel} \geq \text{`High'} \Rightarrow \bigcirc\, \mathsf{PumpMotor} = \text{`On'}.$$

Intuitively, that goal is not realizable by the PumpController agent because it cannot monitor the WaterLevel variable, and cannot control the PumpMotor variable. It would not be meaningful to say that the goal is not realizable by the PumpController because this agent cannot *control* the WaterLevel, *monitor* the PumpMotor; and because the goal constrains the WaterLevel based on the future value of the PumpMotor. The appropriate realizability problems are identified only because we know from domain knowledge that this goal is intended to constrain the PumpMotor, and not the WaterLevel.

As another example, consider the 'utility' goal Maintain[GateOpenWhenNoTrainInCrossing] of the railroad crossing problem [Heit96b], for which we consider the following simplified definition:

$$\Box_{\leq d} \neg \text{TrainInCrossing} \Rightarrow \text{GateOpen}$$

This goal is intended to constrain the variable GateOpen based on future values of the variable TrainInCrossing. Therefore, the goal is not realizable by a GateController agent because: (i) this agent cannot monitor the variable TrainInCrossing; (ii) it cannot control the variable GateOpen; and (iii) the goal refers to future values of the variable TrainInCrossing.

Note that the temporal logic definition of that goal is semantically equivalent to:

$$\neg \text{GateOpen} \Rightarrow \Diamond_{\leq d} \text{TrainInCrossing}.$$

Therefore, if that goal was intended to constrain the variable TrainInCrossing rather than the variable GateOpen, it would have a completely different meaning: it would constrain the future values of the variable TrainInCrossing based on the current value of the variable GateOpen; and the realizability problems would be quite different.

Note also that the concept of reference to the future is a goal property that is independent of the actual interface of the agent: for the above goal, we want to be able to say that it constrains the variable GateOpen based on future values of the variable TrainInCrossing even though this variable is not monitored by the GateController.

As yet another example, consider the goal Achieve[ConvenientMeetingPlanned] in the meeting scheduling problem:

$$\begin{aligned} &\text{m.Requested} \\ &\Rightarrow \Diamond\, (\exists\ \text{d: Date}): \\ &\text{m.Date} = \text{d} \wedge (\forall\ \text{p: Prtcpt}): \bullet\ \text{Intended(p,m)} \rightarrow \text{d} \notin \bullet\ \text{Cstr[p,m].exclset} \end{aligned}$$

From domain knowledge, we know that this goal is intended to constrain the date of the meeting based of the exclusion sets of all intended participants. Therefore, we will identify that this goal is not realizable by a Scheduler software agent because (i) this agent cannot monitor the actual exclusion sets of intended participants, and (ii) the goal is unsatisfiable if the intersection of the exclusion sets of the participants is empty.

Note that goal unsatisfiability is another goal property that is independent of the actual interface of the agent. In the last example, the actual constraints of the participants are not monitored by the Scheduler.

In order to define a taxonomy of realizability problems that provide appropriate explanation why a goal is not realizable, we associate to each goal G, a set

$$\text{Ctrl(G)} \subseteq \text{Voc(G)}$$

that denotes the set of variables that are *intended to be constrained by the goal*. We also define the set $\text{Mon(G)} \subseteq \text{Voc(G)}$ such that $\text{Mon(G)} = \text{Voc(G)} \setminus \text{Ctrl(G)}$.

We assume that the sets Ctrl(G) are given by domain knowledge. For instance, we know from domain knowledge that the goal Maintain[GateOpenWhenNoTrain] is intended to constrain the values of the variables GateOpen. Default choices for the sets Ctrl(G) could also be given based on the syntactical pattern of the goal definition. Usually, the variables intended to be constrained by the goal appear in the consequent of the goal.

5.2. A Complete Taxonomy of Realizability Problems

We now give a fundamental theorem that gives necessary and sufficient conditions for a goal relation to be realizable by an agent.

In this theorem, the following notations about goal relations are used to help defining reference to the future: the set $G_{(M, C)}(h_m)$ returns the set of all histories of constrained variables that satisfy the goal for the history h_m; and the set $G_{(M, C)}(h_m)[i]$ returns all prefixes up to time i of the histories of constrained variables in $G_{(M, C)}(h_m)$. Formally,

$G_{(M, C)}(h_m) = \{h_c \in Hist(C) \mid (h_m, h_c) \in G_{(M,C)}\}$

$G_{(M, C)}(h_m)[i] = \{\sigma_c \in Path(C) \mid \sigma_c = h_c[i]$ for some $h_c \in G_{(M, C)}(h_m)\}$.

Theorem 2 -- Let $G \subseteq Hist(V)$ and ag an agent with monitoring and control capabilities given by Mon(ag) and Ctrl(ag), respectively. G is realizable by ag if, and only if, there exists $C \subseteq Voc(G)$ and $M = Voc(G) \backslash Ctrl(ag)$ such that the following conditions hold:

(i) the agent has sufficient monitoring capabilities

$M \subseteq Voc(ag)$

(ii) the agent has sufficient control capabilities

$C \subseteq Ctrl(ag)$

(iii) $G_{(M,C)}$ is a total relation, that is,

for all $h_m \in Hist(M)$ there exists $h_c \in Hist(C)$ such that $(h_m, h_c) \in G_{(M,C)}$

(iv) $G_{(M,C)}$ does not refer to future values of M, that is, the values at time i of variables in C only depend on the previous values of variables in M up to time i -1.

(v) G is finitely violable, i.e.

for all $h \in History(V)$, if $h \not\models G$ then there exists a finite prefix σ of h such that $\sigma \not\models G$

Condition (iv) is formally captured by the following property requiring that if two arbitrary histories of variables in M are equal up to time i - 1, then they accept the same path of variables in C up to time i:

for all $h_m , h_m' \in$ dom $G_{(M,C)}$ and $i \geq 0$
if $h_m'[i-1] = h_m[i-1]$ or $i = 0$ then $G_{(M, C)}(h_m)[i] = G_{(M, C)}(h_m')[i]$

The proof of the theorem is given in Appendix B.

Note that conditions (i) and (ii) are syntactical conditions. Conditions (iii) to (v) are semantic conditions defined on the set of histories admitted by the goal. Syntactical conditions for verifying these conditions are proposed below.

The taxonomy of unrealizability conditions is defined with respect to a given goal relation $G_{(M,C)}$ by taking the negation of conditions (i) to (v) in the above theorem.

For a goal G, and a given pair (M, C) we will thus use the following criteria:

(i) **Lack of monitorability**: an agent ag lacks monitorability for $G_{(M,C)}$ iff $M \not\subseteq Voc(ag)$

(ii) **Lack of control**: an agent ag lacks control for $G_{(M,C)}$ iff $C \not\subseteq Ctrl(ag)$

(iii) **Goal unsatisfiability**: $G_{(M,C)}$ is unsatisfiable iff $G_{(M,C)}$ is not a total relation

(iv) **Reference to future**: $G_{(M,C)}$ refers to the future values of M iff condition (iv) of theorem 2 is violated

(v) **Not finitely violable goal**: G is not finitely violable iff condition (v) of theorem 2 is violated.

Since the conditions of Theorem 2 are sufficient, we have shown that the taxonomy of unrealizability conditions is complete.

Each of these conditions is illustrated in turn. We also discuss how such conditions can be checked syntactically from the formal definition of the goal.

5.3. Identifying Lack of Monitorability

As mentioned before, an agent lacks monitorability in order to realize a goal $G_{(M, C)}$ iff

$M \not\subseteq Voc(ag)$.

The lack of monitorability of an agent for a goal $G_{(M, C)}$ is thus defined by the set

$M \setminus Voc(ag)$.

As a first example, consider the goal Maintain[PumpOnWhenHighWater] defined by

$WaterLevel \geq High \Rightarrow \bigcirc PumpMotor = \text{`On'}$

The goal is intended to constrain the variable PumpMotor based on the variable WaterLevel. That is, we consider the goal relation

$PumpOnWhenHighWater_{(\{WaterLevel\}, \{PumpMotor\})}$

Since the variable WaterLevel is not among the vocabulary of the PumpController agent, we have identified that the PumpController lacks monitorability for that variable.

As another example, consider a train control system (cfr. Chapter 9) and the goal Maintain[SafeAcceleration] formally defined by

$Following(tr1, tr2) \Rightarrow tr1.Acc \leq F(tr1.Loc, tr2.Loc, tr1.Speed)$.

That is, the acceleration of a train tr1 following a train tr2 should be less than some value which is a function of the positions of trains tr1 and tr2 and of the speed of tr1. The goal is intended to constrain the variable tr1.Acc based on the values of the variables tr1.Loc, tr2.Loc, tr1.Speed, and Following(tr1, tr2). The Loc and Speed attributes of the Train entity denote the physical location and speed of trains. These attributes are not monitored by the TrainController software agent. Therefore, the TrainController cannot realize the goal because it lacks monitorability for the variables tr1.Loc, tr2.Loc, tr1.Speed and Following(tr1, tr2).

Having identified the lack of monitorability of an agent provides guidance on how to elaborate the model so as to resolve the lack of monitorability. Possible ways to resolve lack of monitorability are described in Section 6.5 of Chapter 6.

5.4. Identifying Lack of Control

As mentioned before, an agent lacks control in order to realize a goal $G_{(M, C)}$ iff

$C \not\subseteq Ctrl(ag)$.

The lack of control of an agent for a goal $G_{(M, C)}$ is thus defined by the set $C \setminus Ctrl(ag)$.

As a first example, consider again the goal Maintain[PumpOnWhenHighWater]. The goal constrains the value of the variable PumpMotor; the latter is not directly controllable by the PumpController agent. Therefore, the goal is not realizable by the PumpController because it lacks control of the variable PumpMotor.

As a second example, consider the goal Maintain[SafeAccelaration] defined above. The Acc attribute of the Train entity denotes the physical acceleration of the train. This attribute is constrained by the goal but is not directly controlled by the TrainController agent. Therefore, the goal is not realizable by the TrainController because it lacks control for the variable tr1.Acc.

Having identified the lack of control of an agent provides us guidance for elaborating the model. Possible ways to resolve lack of control are described in Section 6.6 of Chapter 6.

5.5. Identifying Unsatisfiable Goals

As mentioned before, a goal relation $G_{(M, C)}$ is unsatisfiable if $G_{(M, C)}$ is not a total relation, i.e.

there exists $h_m \in Hist(M)$ such that there is no $h_c \in Hist(C)$,
such that $(h_m, h_c) \in G_{(M,C)}$.

Note the similarity between the concept of unsatisfiability of a goal relation and the concept of unsatisfiability of an operation in VDM [Jon90]: an operation there is unsatisfiable iff

there exists a s satisfying the precondition such that there is no next state s'
such that $(s, s') \in Post$.

We define the *domain of unsatisfiability* of a goal relation $G_{(M, C)}$ to be a set of histories $B \subseteq Hist(M)$ such that

$B = Hist(M) \setminus dom\ G_{(M, C)}$

A goal is thus unsatisfiable iff $B \neq \phi$.

(Note also that when the goal is logically inconsistent, we have $B = Hist(M)$.)

The following proposition characterizes goal unsatisfiability at the language level.

Proposition -- A goal relation $G_{(M, C)}$ is unsatisfiable iff there exists a temporal formula B whose state variables are all in M, such that:

(i) $B \models \neg G$

(ii) $B \not\models false$

The proposition suggests applying the following steps in order to identify unsatisfiability:

1. Negate the goal;

2. Strengthen the goal negation so as to remove predicates involving variables in C.

As an example, consider the meeting scheduling problem and the goal defined by

$$
\begin{aligned}
&m.PlanningRequest \\
&\Rightarrow \bigcirc\ (\exists\ d: Date): \\
&(\ m.Date = d \\
&\quad \wedge \bullet\ (\forall\ p: Participant): Intended(p,m) \rightarrow d \notin Constaint[p,m].exclset\ )
\end{aligned}
$$

The goal requires that when a planning request event occurs for a meeting, a meeting date is defined so that it is outside the exclusion set of all intended participants. The goal constrains the possible values of the Date attribute of the meeting.

The negation of the goal is given by the formula

$$
\begin{aligned}
&\Diamond\ (\exists\ m: Meeting): \\
&m.PlanningRequest \\
&\wedge \bigcirc \neg\ (\exists\ d: Date): \\
&(\ m.Date = d \\
&\quad \wedge \bullet\ (\forall\ p: Participant): Intended(p,m) \rightarrow d \notin Constaint[p,m].exclset\ )
\end{aligned}
$$

The goal negation can then be strengthened by removing the predicate m.Date = d, yielding the formula:

$$
\begin{aligned}
&\Diamond\ (\exists\ m: Meeting): \\
&m.PlaningRequest \\
&\wedge \neg\ (\exists\ d: Date): (\forall\ p: Participant): Intended(p,m) \rightarrow d \notin Constaint[p,m].exclset
\end{aligned}
$$

By construction, this formula satisfies condition (i) in the above proposition. Since the formula is not inconsistent, we have shown that the goal is unsatisfiable. Therefore, we have identified that the goal is unsatisfiable when a planing request occurs and there is no date that is outside the exclusion sets of all intended participants.

Techniques for resolving goal unsatisfiability are described in Section 6.7 of Chapter 6.

5.6. Identifying References to the Future

As mentioned before, a goal relation does not refer to the future of M iff the following condition holds:

there exists h_m , $h_m' \in$ dom $G_{(M,C)}$ and $i \geq 0$ such that
($h_m'[i-1] = h_m[i-1]$ or $i = 0$) and $G_{(M, C)}(h_m)[i] \neq G_{(M, C)}(h_m')[i]$

Note that the condition of no reference to the future requires that when two histories of variables in M are equal up to time **i-1**, they must accept the same path of variables in C up to time **i**. This means that a goal should not constrain the values of variables in C based on the future or *current* values of variables in M.

In the sequel, we say that a goal *refers to the strict future* of M iff the following condition holds:

there exists h_m , $h_m' \in$ dom $G_{(M,C)}$ and $i \geq 0$ such that
$h_m'[i] = h_m[i]$ and $G_{(M, C)}(h_m)[i] \neq G_{(M, C)}(h_m')[i]$

We say that a goal formulation has a *synchronization problem* iff it refers to the future of M and does not refer to the strict future of M.

As an example of goal that refers to the strict future, consider the railroad crossing problem [Heit96b] and the 'utility' goal Maintain[GateOpenWhenNoTrain]:

$\square_{\leq d} \neg$ (∃ tr: Train): InCrossing(tr,cr) ⇒ cr.Gate = 'opened'

One can show that the value of the Gate attribute is constrained by the future values of the InCrossing relationship.

As an example of a goal with synchronization problem, consider a train control system and the goal Maintain[DoorsClosedWhileMoving]:

tr.Moving ⇒ tr.DoorsState = 'Closed'

The goal constrains the value of the variable tr.DoorsState based on the current value of the variable tr.Moving. Therefore, a DoorController agent monitoring the variable tr.Moving and controlling the variable tr.DoorsState cannot realize the goal. Indeed, if at some time the doors are opened, and the train starts moving, the door controller would have to close the doors *simultaneously* with the departure of the train.

Identifying whether a goal refers to future values of monitored variables by reasoning at the semantic level is a tedious process. Therefore, we would like to be able to identify that a goal refers to the future values of monitored variables from the syntax of the goal definition. For instance, one can immediately see from the definition of the goal Maintain[GateOpenWhenNoTrain] that the value of the attribute cr.Gate is constrained by the future values of the relationship InCrossing.

Unfortunately, it is not easy to give a general characterization of temporal formulas that constrain a set of variables C based on the future values of variables in M. The fact that a goal constrains variables in C based on future values of variables in M is not equivalent to the fact that an occurrence of a variable in M appears in a "future" subformula of the goal definition. Consider, for instance the following temporal formula:

- $P \Rightarrow Q \; W \; (Q \wedge R)$.

Pattern of reference to strict future

$\mathsf{Future(M)} \Rightarrow \mathsf{Past(C)}$

where Future(M) is a future temporal formula on variables in M, and Past(C) is a past temporal formula on variables in C.

Patterns of synchronization problem

Maintain Goals

$\mathsf{P(M)} \Rightarrow \mathsf{Q(C)}$

$\mathsf{P(M)} \Rightarrow \boxdot\, \mathsf{Q(C)}$

$\mathsf{P(M)} \Rightarrow \mathsf{Q(C)}\ W\ \mathsf{S}$

$\bullet\ \mathsf{P(M)} \Rightarrow \mathsf{Q(C)}\ W\ \mathsf{R(M)}$

$\blacksquare_{\leq d}\ \mathsf{P(M)} \Rightarrow \mathsf{Q(C)}$

where P(M), R(M) are state formulas on variables in M;
Q(C) is a state formula on variables in C;
S is any state formula.

Achieve Goals

$\mathsf{P(M)} \Rightarrow \bigcirc\, \mathsf{Q(C} \cup \mathsf{M)}$

$\mathsf{P(M)} \Rightarrow \Diamond\, \mathsf{Q(C} \cup \mathsf{M)}$

$\mathsf{P(M)} \Rightarrow \Diamond_{\leq d}\, \mathsf{Q(C)}$
where $d \in M$.

where P(M) is a state formula on variables in M;
Q(C) is a state formula on variables in C;
Q(C ∪ M) is a state formula on variables in C and M.

FIGURE 5.1. Patterns of references to the future

The occurrence of the variable R appears in a “future” subformula. However, the formula does not constrain Q based on the future value of R; one can show that the goal is realizable by monitoring P, R, and controlling Q.

In order to help in detecting references to the future, we have started to build a library of recurrent patterns of goal definitions with references to the future. These patterns are listed in Figure 5.1.

As an example, one can identify that the goal Maintain[GateOpenWhenNoTrain] refers to the strict future because its formal definition matches the pattern of reference to strict future in Figure 5.1. Similarly, one identifies that the goal Maintain[DoorsClosedWhileMoving] suffers from synchronization problem because its formal definition matches the first pattern of synchronization problems for Maintain goals.

As an example of a synchronization problem for Achieve goals, consider the goal

$$\begin{aligned} & m.Requested \\ \Rightarrow\ & \Diamond\ (\exists\ d: Date) \\ & m.Date = d \\ & \quad \wedge\ (\forall\ p: Participant): Intended(p,m) \rightarrow d \notin Constraint[p,m].\ exclset \end{aligned}$$

The goal is intended to constrain the variable m.Date based on the variables m.Requested, Intended(p,m) and Constraint[p,m].exclset. The goal matches the following pattern of synchronization problems:

$$P(M) \Rightarrow \Diamond\ Q(C \cup M)$$

The goal is not realizable because it constrains the value of the variable m.Date based on the *current* value of the variables Intended(p,m) and Constraint[p,m].exclset.

Techniques for resolving references to the strict future and synchronization problems are described in Section 6.8 of Chapter 6.

5.7. Identifying Unbounded Achieve Goals

The last class of unrealizable goals are goals that do not constrain the finite runs of the agent assigned to them.

One can identify that a temporal formula is a liveness property from the structure of the Buchi automaton equivalent to that formula [Alp87].

In practice, we consider that a goal does not constrain the finite runs of an agent if its formal definition matches one of the following patterns:

$$R \Rightarrow \Diamond\ S,\ R \Rightarrow P\ U\ S$$

5.8. Summary

We have defined a taxonomy of realizability problems that allows one to identify unrealizability from the formal definition of goals, and provide explanations about why the goal is unrealizable. We have also proved that this taxonomy is complete: every unrealizable goal satisfies at least one of the realizability problem. Techniques for resolving each kind of realizability problem are described in the following chapter.

Chapter 6
Agent-Driven Tactics for Elaborating Goal Models

This chapter proposes a systematic technique for identifying agents and their capabilities, and for refining goals into subgoals until the latter are realizable by single agents.

The general principle is to provide a library of specification elaboration tactics whose applications are driven by the need to resolve violations of the realizability meta-constraint. Specific tactics are provided for each category of realizability problem. These tactics provide systematic guidance for recursively refining goals into subgoals, and for identifying new agents. Alternative goal refinements are explored through the application of alternative tactics. Formal goal refinement patterns associated with the tactics provide guidance for elaborating goal refinements that are proved correct.

The chapter is structured as follows. Section 6.1 describes the general principle of using agent-driven tactics for elaborating goal models. Section 6.2 shows a first example of tactics and its application. Section 6.3 discusses the benefits of agent-driven tactics for elaborating requirements. Section 6.4 describes how the library of tactics has been built and discusses the coverage of the library. Sections 6.5 to 6.11 describe the library of agent-driven tactics in detail.

6.1. Basic Idea

Specification elaboration tactics are heuristic rules that provide guidance for elaborating requirements models. Applications of tactics transform the model so as to satisfy some process-level objectives [Dar95].

Agent-driven tactics are specification elaboration tactics whose applications are driven by the need to resolve unrealizable goals. Applications of agent-driven tactics result in transformed goal, object, and agent models.

The description of tactics used in the thesis is based on the ICARUS meta-model for process description and its extension to process-level objectives [Dar95]. Specialization links between tactics are furthermore introduced to help requirements engineers browse the library of tactics.

Each tactic is defined by the following items:

- a **motivation** that describes the process-level objective addressed by the tactics.
- a **precondition** that characterizes the current state of the specification model to which the tactic can be applied.
- an application **heuristic** that defines when the tactic should be applied. Application heuristics are proposed to and evaluated by requirements engineers. They guide them in selecting which tactic to apply.
- a **postcondition** that characterizes the state of the specification model after application of the tactic. Postconditions are defined by their effects on the object model, goal

model and agent models, respectively. A set of *formal goal refinement patterns* [Dar95, Dar96] associated to the tactics is used to define formally the goal refinements produced by the tactic.

The tactics are organized into a **specialization hierarchy**. This hierarchy is used to browse the library of tactics and select some appropriate one. Specialized tactics inherit features from their parent tactics in the usual way: more specialized tactics have stronger pre- and post-conditions as well as stronger heuristics. More specialized tactics therefore provide more specific guidance.

Figure 6.1 shows the top structure of the library of agent-driven tactics. The library is organized according to the taxonomy of realizability problems defined in Chapter 5. Specialized tactics for resolving each kind of realizability problem are defined in Sections 6.5 to 6.11.

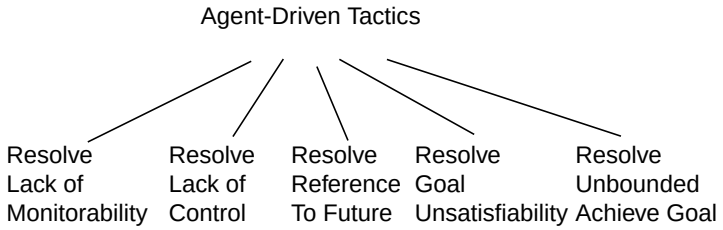


FIGURE 6.1. The library of agent-driven tactics

6.2. A First Example

As a first example, Figure 6.2 shows the definition of the tactic introduce accuracy goal that is a specialization of the tactic resolve lack of monitorability.

As an example of application of the tactic, consider the mine pump problem and the goal Maintain[PumpOnWhenHighWater]:

WaterLevel ≥ 'High' ⇒ PumpMotor = 'On'

The goal is unrealizable by a PumpController agent because it lacks monitorability for the WaterLevel variable. An application of the tactic introduce accuracy goal resolves that lack of monitorability by elaborating the object and goal models as follows:

1. Object model elaboration: the object model is enriched with a new variable

HighWaterSignal.

2. Goal model elaboration: the goal Maintain[PumpOnWhenHighWater] is refined into the subgoals:

Maintain[HighWaterDetected]

Maintain[PumpOnWhenHighWaterDetected].

Tactic introduce accuracy goal

Motivation: resolve lack of monitorability

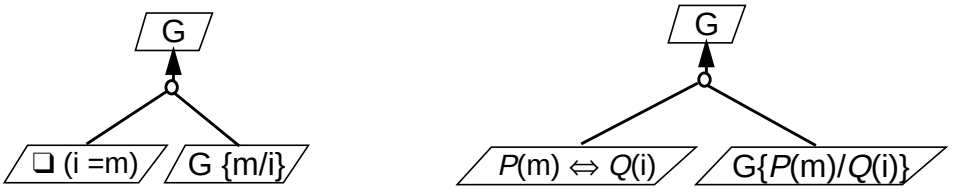
Precondition: the agent ag lacks monitorability of m in order to realize $G_{(M, C)}$

Heuristics: the tactic should be applied when one can identify an intermediate variable i that can be related to m through some accuracy goal. (The accuracy property relating i to m can also be a domain property.)

PostCondition: The tactic elaborates the object and goal models as follows:

1. Object model elaboration. The object model is enriched with a new variable i denoting an image of the variable m.

2. Goal model elaboration. The unrealizable goal is refined into an accuracy goal relating i to m, and a companion subgoal whose definition refers to the variables i instead of m. Two formal refinement patterns for the tactic are shown below.



(a) introduce accuracy goal *on variable* (b) introduce accuracy goal *on predicate*

The notation G{x/y} is used to denote the substitution of every occurrence of x by y in the definition of G. The validity of these goal refinement patterns is based on the substitutivity property of temporal logic [Man92]. In the pattern introduce accuracy goal *on predicate*, the symbols *P*(m) and *Q*(i) are used to denote formulas involving the state variables m and i respectively.

The patterns introduce accuracy goal *on variable* and introduce accuracy goal *on predicate* correspond to alternative ways of applying the tactic.

Variants of these patterns dealing with *non-ideal accuracy goals* involving tolerances and delays are discussed in Section 6.5.4.3.

Specialization: introduce tracking object, introduce sensor agent

FIGURE 6.2. The tactic introduce accuracy goal

These goals are formally defined by applying the pattern introduce accuracy goal *on predicate* with the following instantiation:

P(m): WaterLevel ≥ ‘High’ Q(i): HighWaterSignal = ‘On’

The resulting goal definitions are thereby obtained:

HighWaterSignal = ‘On’ ⇔ WaterLevel ≥ ‘High’

HighWaterSignal = ‘On’ ⇒ PumpMotor = ‘On’.

As another example of application of the same tactics, consider the patient monitoring problem [Ste74] and the goal Achieve[AlarmRaisedForCriticalPulseRate]. Assume that the goal is formally defined as follows:

$$p.PulseRate \notin p.SafePulse \Rightarrow \Diamond_{\leq d} (\exists\ a: Alarm): a.Raised \wedge a.Loc = p.BedNbr$$

The goal is not realizable by the PatientMonitoring software agent because it lacks monitorability of the variables p.Pulserate, p.SafePulse, and p.BedNbr. The tactics introduce accuracy goal can be used to resolve such lack of monitorability. An application of that tactic yields the following elaboration of the object and goal models:

1. Object model elaboration -- The object model is enriched with a new object PatientInfo denoting information known about patient status, and a Tracking relationship relating PatientInfo to Patient (see Figure 6.3).

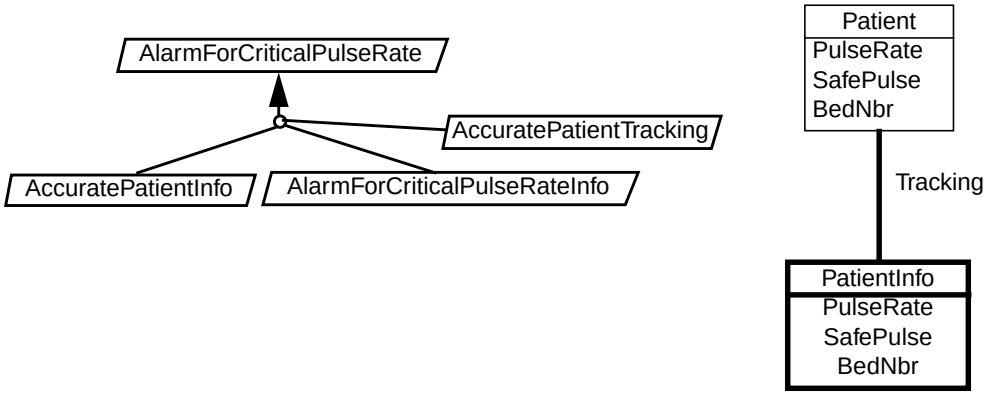


FIGURE 6.3. Goal model and object model for the patient monitoring system after application of the tactic introduce accuracy goal

2. Goal Model Elaboration -- The goal Achieve[AlarmRaisedForCriticalPulseRate] is refined into the following subgoals (Figure 6.3):

Goal Maintain[AccuratePatientInfo]
FormalDef $\forall$ p: Patient, pi: PatientInfo
Tracking(pi,p)
$\Rightarrow$
pi.PulseRate = p.PulseRate
$\wedge$ pi.SafePulse = p.SafePulse
$\wedge$ pi.BedNbr = p.BedNbr

Goal Achieve[AlarmRaisedForCriticalPulseRateInfo]
FormalDef $\forall$ pi: PatientInfo
$pi.PulseRate \notin pi.SafePulse \Rightarrow \Diamond_{\leq d} (\exists\ a: Alarm): a.Raised \wedge a.Loc = pi.BedNbr$

Goal Maintain[PatientTracked]
FormalDef
$(\forall\ p: Patient)\ \square\ (\exists\ !\ pi: PatientInfo): Tracking(pi,p)$
$\wedge$
$\forall$ p: Patient, pi: PatientInfo
$Tracking(pi,p) \Rightarrow \square\ Tracking(pi,p)$

The process of identifying realizability problems and applying agent-driven tactics to resolve these problems is applied recursively until all subgoals can be made realizable by single agents. For instance, the goal Maintain[AccuratePatientInfo] is still not realizable by a single agent in the domain considered. Further refinements of that goal will be described below.

6.3. Benefits of Agent-Driven Tactics

Agent-driven tactics are useful for the following reasons:

- They provide *systematic and recursive guidance* for elaborating requirements models;
- They provide ways to *explore alternative goal refinements*;
- Formal goal refinement patterns associated with the tactics generate goal refinements that are *proved correct* while *hiding formal reasoning* from the requirements engineers [Dar95].

Each point is illustrated in turn.

6.3.1. Systematic elaboration of requirements

Agent-driven tactics provide systematic guidance for recursively refining unrealizable goals into subgoals until the latter can be assigned as responsibilities of single agents.

For example, Figure 6.4 shows a portion of the goal refinement graph for the patient monitoring problem; the graph is generated by recursively applying agent-driven tactics so as to resolve realizability problems.

At the top of Figure 6.4, the goal Achieve[NurseInterventionForCriticalPulseRate] is defined by:

$$p.PulseRate \notin p.SafePulse \Rightarrow \Diamond_{\leq interevention\_delay} (\exists\ n: Nurse): Intervention(n,p)$$

That goal is unrealizable by Nurse agents because they lack monitorability of the patients' pulse rates. A tactic called split lack of monitorability with milestone is used to resolve such lack of monitorability by generating the subgoals:

Achieve[AlarmForCriticalPulseRate]

Achieve[NurseInterventionForAlarm].

The application of this tactic also introduces the new Alarm entity together with the PatientMonitoring software agent that controls this entity. The Alarm entity is monitored by Nurse agents, and the second subgoal is assigned as responsibility of that agent.

As seen before, the generated subgoal Achieve[AlarmForCriticalPulseRate] is unrealizable by the PatientMonitoring software agent; and the tactic introduce accuracy goal can be used to resolve that lack of monitorability; it generates the subgoals:

Maintain[AccuratePatientInfo]

Achieve[AlarmForCriticalPulseRate]

Maintain[AccuratePatientTracking]

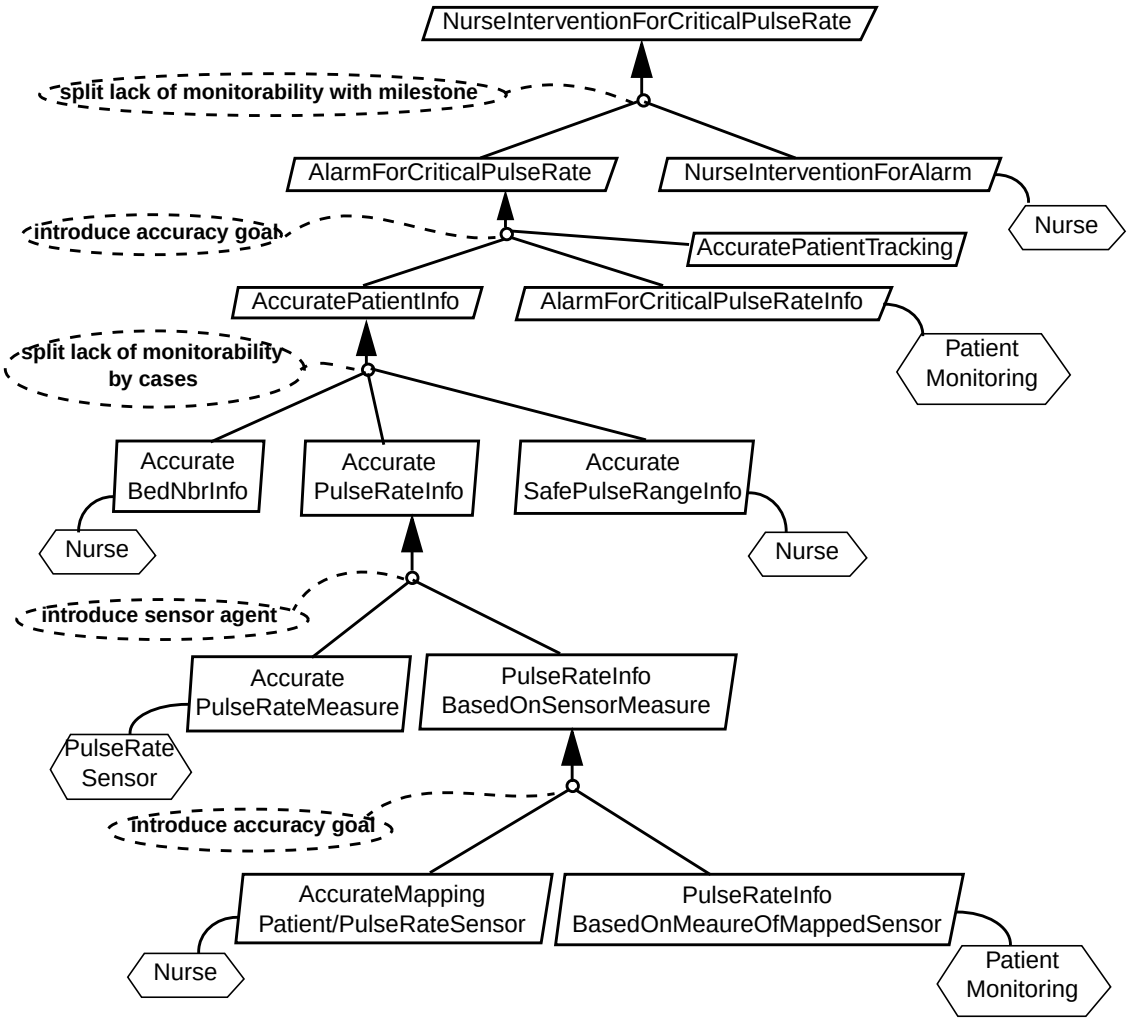


FIGURE 6.4. Applying agent-driven tactics for the patient monitoring problem

The entity PatientInfo is also identified through the application of this tactic. This entity is declared as an internal variable of the PatientMonitoring agent, and the goal Achieve[AlarmForCriticalPulseRate] is now realizable by that agent.

The tactic split lack of monitorability by cases is then used to refine the goal Maintain[AccuratePatientInfo] into cases. The generated formal definition for the goal Maintain[AccuratePulseRateInfo] is given by:

$$Tracking(pi,p) \Rightarrow pi.PulseRate = p.PulseRate$$

This subgoal is unrealizable by the PatientMonitoring software agent because it lacks monitorability of the patients' pulse rates. The tactic introduce sensor agent is then used to refine that goal into the subgoals:

Maintain[AccuratePulseRateMeasure]

Maintain[PulseRateInfoBasedOnSensorMeasure]

The application of this tactics also introduces the PulseRateSensor agent capable of monitoring patients' pulse rates, and assigns the goal Maintain[AccuratePulseRateMeasure] to that agent. The generated formal definition for the companion subgoal Maintain[PulseRateInfoBasedOnSensorMeasure] is given by:

$$\text{Tracking}(pi,p) \wedge \text{HasPulseRateSensor}(p, s) \Rightarrow pi.\text{PulseRate} = s.\text{PulseMeasure}$$

This goal is still not realizable by the PatientMonitoring agent because the agent lacks monitorability of the relationship HasPulseRateSensor. (In order the realize the goal, the PatientMonitoring agent has to know which pulse rate sensor is connected to which patient.) Another application of the tactic introduce accuracy goal resolves that lack of monitorability by refining the goal into:

Maintain[AccurateMappingPatient/PulseRateSensor]

Maintain[PulseRateInfoBasedOnMeasureOfMappedSensor].

The first subgoal is an accuracy goal that could be assigned as responsibility of the Nurse agent; the second subgoal can be assigned as responsibility of the PatientMonitoring software agent.

The agent interface model that has been gradually elaborated by the above tactics is shown in Figure 6.5.

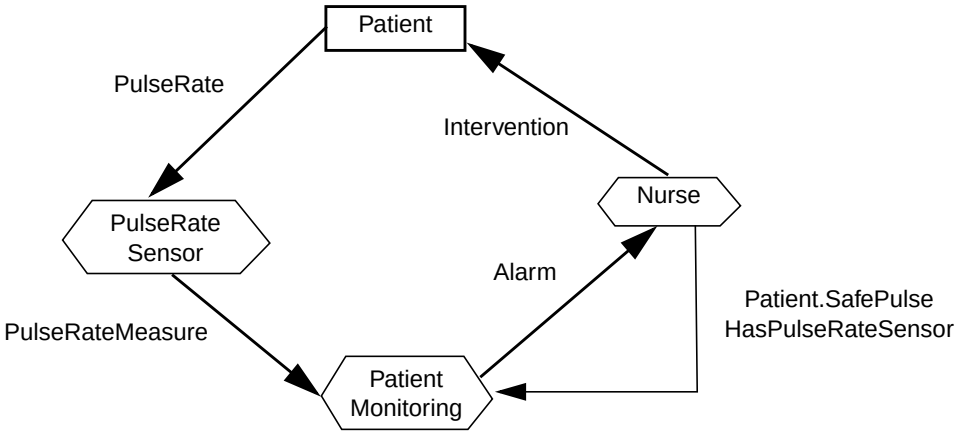


FIGURE 6.5. Partial agent interface model for the patient monitoring problem derived by application of the tactics in Figure 6.4.

6.3.2. Exploration of alternatives

Agent-driven tactics allow for the exploration of *alternative* goal refinements and responsibility assignments. For every realizability problem, alternative applications of agent-driven tactics can be considered. Such alternative applications produce alternative goal refinement and agent models, corresponding to alternative system designs in which the boundaries between the automated system and its environment may be fairly different.

As a first example, consider the meeting scheduling problem and the goal Achieve[PrtcptsCstrKnown]; this goal requires that information about participants' date constraints are eventually accurately known by the scheduler.

This goal is not realizable by the MeetingScheduler agent because the latter cannot monitor the actual date constraints of participants. A first way to resolve this lack of monitorability is to apply the tactic split lack of monitorability with milestone so as to produce the first And-refinement in Figure 6.6. An alternative way to resolve this lack of monitorability is to apply the tactic introduce accuracy goal so as to produce the second And-refinement in Figure 6.6. In the first alternative, participants' constraints are obtained by sending constraint request messages to the participants; in the second alternative, they are obtained from electronic agendas of the participants.

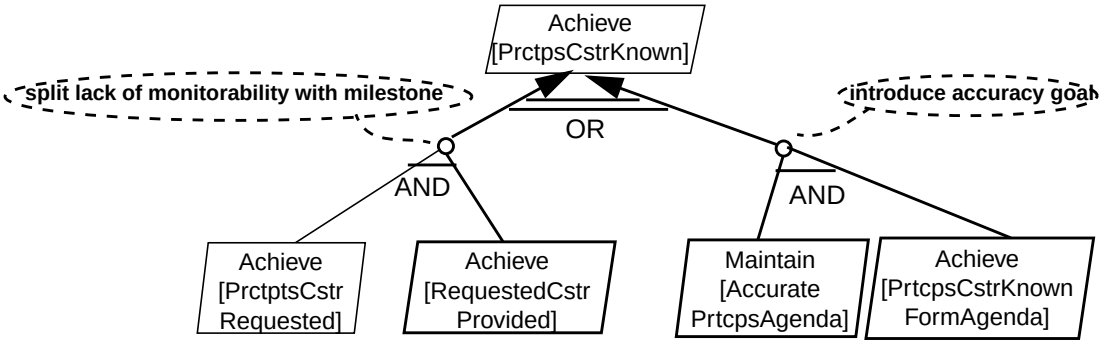


FIGURE 6.6. Alternative refinements of the goal Achieve[PrtcptsCstrKnown]

As another example, consider again the mine pump control system and the goal Maintain[PumpOnWhenHighWater]. We previously applied the tactic introduce accuracy goal *on predicate* to produce the subgoals Maintain[HighWaterDetected] and Maintain[PumpOnWhenHighWaterDetected]. The alternative tactic introduce accuracy goal *on variable* can be used to produce the alternative subgoals Maintain[AccurateWaterMeasure] and Maintain[PumpOnWhenHighWaterMeasure]. This alternative refinement results in alternative agent responsibilities and interfaces: a unique WaterSensor agent is responsible for measuring the water level, and the task of comparing the measured water level against its high level is transferred to the PumpController agent.

6.3.3. Formally complete goal refinements

Formal goal refinement patterns associated with the tactics help produce goal refinements that are proved complete [Dar95, Dar96]. Formal goal refinement patterns are abstract refinement links between abstract goal definitions. They are proved correct once and for all, and can be reused through instantiation in many application domains. Reusing a pattern entails reusing its proof. Formal reasoning is therefore hidden from the requirements engineer.

Our work focuses on the use of goal refinement patterns for the constructive elaboration of goal refinement graphs. Formal goal refinement patterns can also be used for checking given goal refinements for completeness, and for identifying goals and assumptions that were overlooked in the first place (see [Dar95, Dar96] for details).

6.4. Building a Library of Agent-Driven Tactics

To be effective in practice, the library of agent-driven tactics should have the following qualities.

- **Coverage**: the library should provide effective guidance for situations that are encountered by requirements engineers.
- **Relevance:** the library should not be polluted by irrelevant tactics.
- **Retrievability**: the library should be organized so that relevant tactics can be easily retrieved.

We first describe how relevant tactics have been identified and classified for retrievability. The coverage of the library is addressed next.

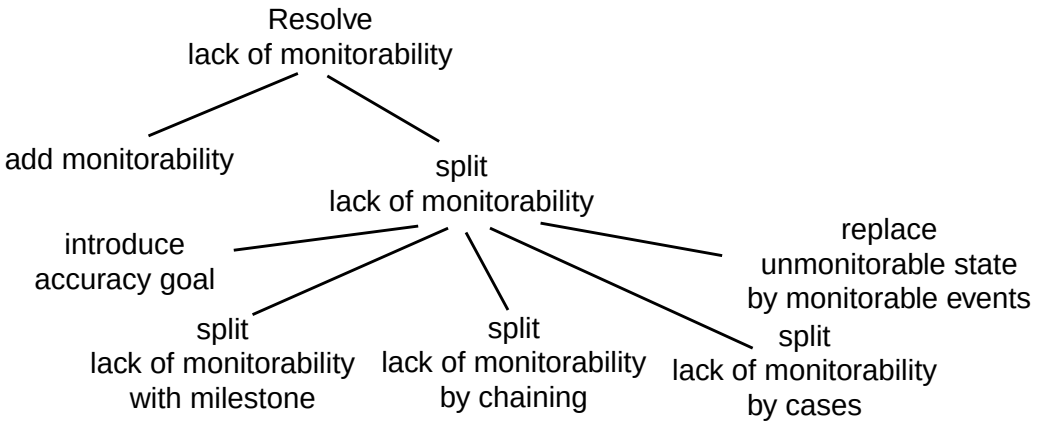
6.4.1. Identifying tactics

Two complementary approaches were followed to identify tactics.

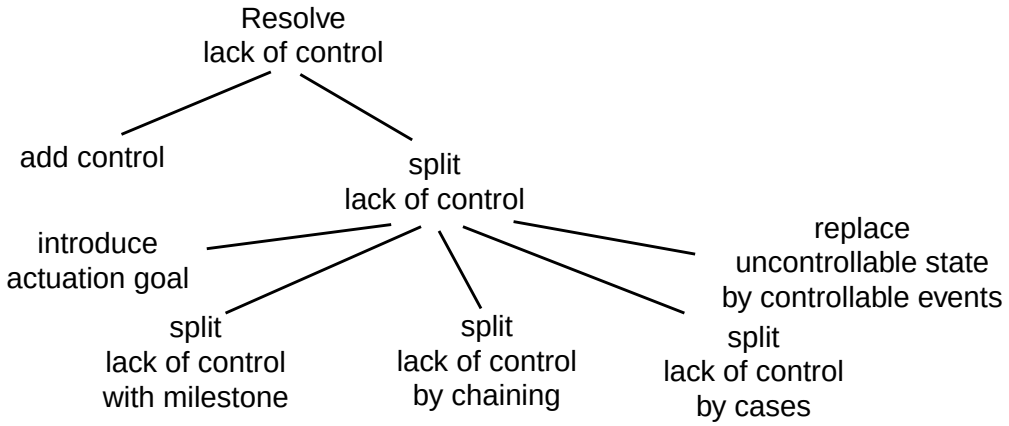
Inferring tactics from examples. We inferred tactics from examples of goal-oriented specifications in the literature. These include a lift system [Dar91, Fea87], a library system [Dar91], the meeting scheduler problem [Lam95], resource allocation systems [Dar95], the package router [Lon82], and a turnstile system [Jack95]. We also used examples of applications of formal goal refinement patterns given in [Dar95, Dar96]. Besides, we inferred tactics from our own cases studies; the latter were performed by systematically identifying and resolving realizability problems. These case-studies include the mine pump problem [Jos96], a patient monitoring system [Ste74], an ambulance dispatching system [LAS93], and an automated train control system [Win99]. In all these examples, we tried to identify goal refinements that could be motivated by the need to resolve violations of the realizability meta-constraint; we generalized them into corresponding agent-driven tactics. In some cases, the goal refinements had to be adapted because they were incomplete or because they did not adequately resolve violations of realizability. These tactics were then classified according to the kind of realizability problem they resolve.

Systematic exploration of the space of tactics. To complement the identification of tactics from examples, we also used the specialization hierarchy as a map to systematically explore the space of tactics. For each realizability problem, we tried to systematically identify from the definition of that problem what alternative tactics could be used to resolve it. In order to achieve complete coverage of the space of tactics, we tried to identify tactics so that the specialized tactics offer a complete specialization of the parent tactics. We also noted that lack of monitorability and lack of control are symmetric problems. We exploited that symmetry to identify symmetric tactics for resolving lack of monitorability and lack of control.

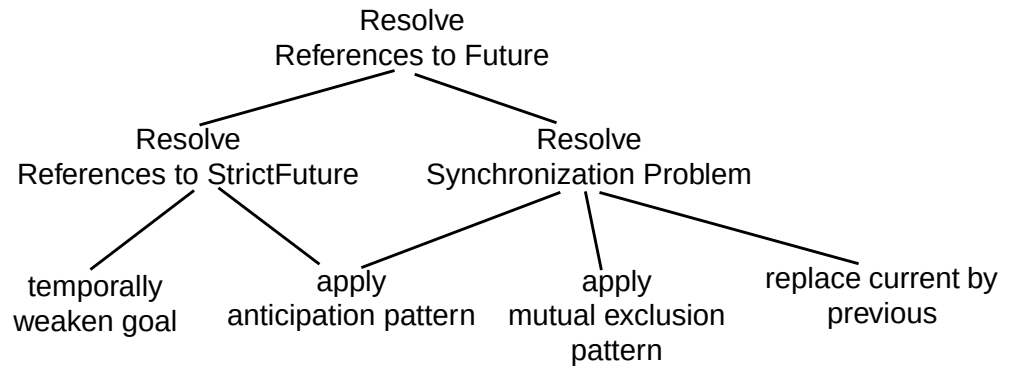
To give an idea of the space of tactics covered, Figure 6.7 shows the tactics that have been identified so far. These tactics are fully defined in Sections 6.5 to 6.11.



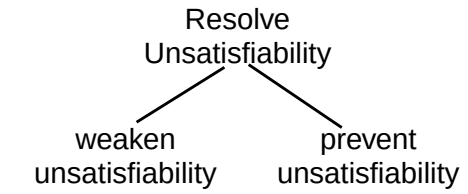
(a) Tactics for resolving lack of monitorability



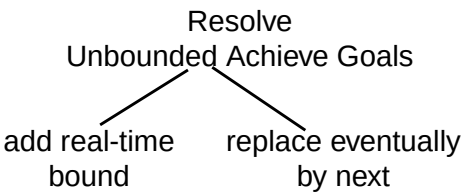
(b) Tactics for resolving lack of control



(c) Tactics for resolving references to the future



(d) Tactics for resolving goal unsatisfiability



(e) Tactics for resolving unbounded Achieve goals

FIGURE 6.7. The library of agent-driven tactics

6.4.2. Coverage of the library

The coverage of the library can be assessed *theoretically* by determining the coverage of the specialization links of the library. Note that the top structure of the library of agent-driven tactics is complete, because Theorem 2 defines a complete taxonomy of realizability problems (see Section 5.2). However, these high-level tactics provide only limited guidance for elaborating the model. Further specialized tactics are therefore defined to provide more specific guidance. Theoretically determining the coverage of these specialized tactics is more difficult. Specialized tactics are based on recurrent patterns of goal definitions and goal refinements. The coverage of those specialized tactics is then relative to the coverage of the taxonomy of goal patterns.

The coverage of the library can also be assessed *empirically* by determining to what extent these tactics can be used to effectively specify goal refinement graphs for real systems. In our experience, the current library of tactics provides effective guidance for elaborating goal refinement graphs for all case-studies we have considered. A systematic experiment based on further case studies, in the spirit of the one described in [Dwy99], should be carried out for validating the coverage of the library more precisely and possibly for identifying further tactics.

The tactics identified so far make little use of goal categories (such as satisfaction goals, information goals, accuracy goals, security goals, etc.). Further specialized tactics could be explored by identifying specialized ways of resolving realizability problems for specific categories of goals. The basic idea for such specialized tactics would be similar to the idea of using problem frames [Jac95b, Jac2K].

6.5. Resolving Lack Of Monitorability

We first detail tactics for resolving lack of monitorability. Two ways of resolving lack of monitorability of an agent consists in adding the necessary monitoring links to the agent or refining the goal further. The tactics add monitorability and split lack of monitorability are described next. We then describe further specialization of the tactic split lack of monitorability.

6.5.1. Add monitorability

The simplest way to resolve an agent's lack of monitorability for a variable m is to add a monitoring link between the agent and the variable. This tactic can however be applied only if m can actually be monitored by the agent. When m cannot be monitored by the agent, the goal has to be refined further.

Tactic add monitorability

Motivation: resolve lack of monitorability

Precondition: the agent ag lacks monitorability of variable m in order to realize $G_{(M, C)}$

Heuristics: the tactic should be applied only if the variable can actually be made monitorable by the agent. When the variable cannot be made monitorable, one has to consider the alternative tactic split lack of monitorability described below.

PostCondition: a new monitorability link is created between ag and m.

Example: Consider the meeting scheduling problem and the goal Achieve[ParticipantsCstrRequested] defined by:

Intended(p,m) ∧ Scheduling(sch,m)
⇒ ◊ (∃ cstr_req: CstrRequest):
cstr_req.Sent
∧ cstr_req.MeetingName = ● m.Name
∧ cstr_req.delivery_address = ● p.address

Assume that given the current agent interface model, the Scheduler agent lacks monitorability of the predicate Intended(p,m). The tactic add monitorability resolves that lack of monitorability by declaring the following monitoring link and instance-level declaration:

Monitoring [Scheduler, Intended]
InstDecl: Scheduling(sch,m) ⇒ *Mon*(sch, Intended(p,m))

Note that in order to realize the goal the Scheduler also lacks monitorability of the variable p.Address that denotes the actual e-mail address of the participant. If that variable is not directly monitorable by the Scheduler agent, the tactic add monitorability cannot be applied and the goal has to be further refined.

6.5.2. Split lack of monitorability

When an agent lacks of monitorability for a variable that cannot be made monitorable by that agent, the goal has to be refined by applying the tactic split lack of monitorability.

Tactic split lack of monitorability

Motivation: resolve lack of monitorability

Precondition: the agent ag lacks monitorability of m in order to realize $G_{(M, C)}$.
(The symbol m denotes a set of variables, possibly a singleton, included in M).

Heuristics the tactic should be applied when m cannot be monitored by ag.

PostCondition

1. Object model elaboration. The object model is enriched with new intermediate variables i. (i also denotes a set of variables).

2. Goal model elaboration. The goal G is refined into the subgoals: $G1_{(m, i)}$, that relates i to m, and $G2_{((M \setminus m) \cup i, C)}$. Note that G2 does not refer to the unmonitorable variables m.

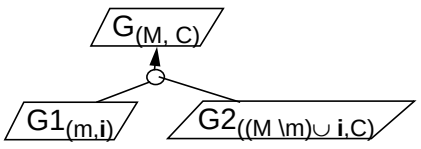


FIGURE 6.8. split lack of monitorability

Example. Consider again the mine pump control system and the refinement of the goal Maintain[PumpOnWhenHighWater] into the subgoals:

Maintain[HighWaterDetected]

Maintain[PumpOnWhenHighWaterDetected].

This refinement corresponds to an application of the tactic split lack of monitorability instantiated as follows:

m: Waterlevel i: HighWaterSignal

$G1_{(m,i)}$: Maintain[HighWaterDetected]

G2 : Maintain[PumpOnWhenHighWaterDetected].

The description of the tactic is slightly simplified. As will be seen below, tactics for resolving lack of monitorability may sometimes refine G into more than two subgoals. Also, the assertion G1 relating i to m can be a domain property instead of a goal.

Further guidance for refining goals so as to split lack of monitorability are described in specialized tactics.

3. Agent model elaboration. The new intermediate variables i must be monitorable by ag or must be internal variables of ag. This ensures that the subgoal G2 refers to fewer unmonitorable variables than the parent goal G.

A first way to elaborate the agent model consists in identifying a *monitoring agent* capable of monitoring m and controlling i, and in assigning the goal $G1_{(m,i)}$ to that agent (Figure 6.9). In the context of control systems, external sensors are typical examples of such monitoring agents.

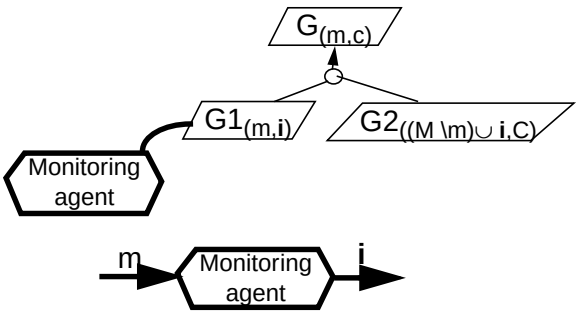


FIGURE 6.9. Introduce monitoring agent

Alternatively, the new intermediate variable i can also be declared as an internal variable of some agent. This alternative elaboration of the agent model makes the goal refinement graph more robust with respect to changes of monitoring agent or agent interfaces. A heuristics of the goal refinement process is therefore to favour such a device-independent goal refinement and to introduce monitoring agents and interfaces as late as possible in the goal refinement process. This heuristics is further discussed below.

Example 1. In the above refinement of the goal Maintain[PumpOnWhenHighWater], the agent model is also elaborated by introducing the HighWaterSensor agent, capable of (i) monitoring whether the actual water level is above high and (ii) controlling the HighWaterSignal variable; The goal Maintain[HighWaterDetected] is then assigned to that agent.

Example 2. To illustrate the case where the new intermediate variable is an internal variable, consider the patient monitoring system and the refinement of the goal Achieve[AlarmForCriticalPulseRate] into the following subgoals described in Section 6.2:

Maintain[AccuratePatientInfo]

Achieve[AlarmForCriticalPulseRateInfo]

Maintain[PatientTracked]

This refinement corresponds to an application of the tactic split lack of monitorability instantiated as follows:

m: {Patient.PulseRate, Patient.SafePulse, Patient.BedNbr}

i: {PatientInfo.PulseRate, PatientInfo.SafePulse, PatientInfo.BedNbr}

$G1_{(m,i)}$: Maintain[AccuratePatientInfo]

G2 : Achieve[AlarmForCriticalPulseRateInfo]

(The goal refinement also uses a third subgoal Maintain[PatientTracked] that requires every patient to be tracked by exactly one instance of the PatientInfo entity.)

Note that in this application of the tactic, the intermediate entity PatientInfo is declared as an internal object of the PatientMonitoring software agent. This elaboration does not yet commit to particular sensor agents and interfaces between sensors and the PatientMonitoring software. Such agents and interfaces are identified latter on when refining the goal Maintain[AccuratePatientInfo].

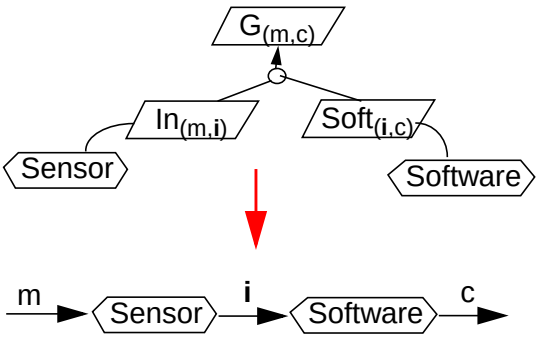
Device-independent vs. device-dependent goal refinements

A good practice for elaborating goal refinement graphs is to start with device-independent goal refinements and introduce monitoring agents as late as possible in the goal refinement process. This makes goal graphs more robust with respect to changes of monitoring agents and interfaces between agents. The reader may compare the two goal refinement graphs shown in Figure 6.10.

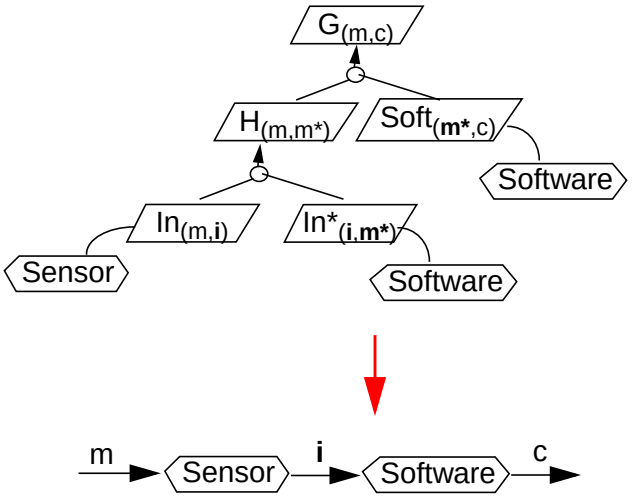
In the device dependent goal refinement (Figure 6.10.a), $G_{(m,c)}$ is refined into subgoals $G1_{(m,i)}$ and $G2_{(i,c)}$, and the intermediate variable i is declared as an interface variable between the Sensor agent and the Software agent.

In the device-independent goal refinement (Figure 6.10.b), an internal variable m* is first introduced to denote an internal image of the environment variable m. The goal $H_{(m,m^*)}$ is then refined by introducing the sensor agent and the interface variable i. The two goal graphs describe the same system; only the model structures differ.

The benefit of device-independent goal refinement is more robustness of the resulting goal graph with respect to changes of agents and agent interfaces. Alternative interfaces between the sensor and the software can be explored through alternative refinements of the goal $H_{(m,m^*)}$ while keeping the goal $Soft_{(m^*,c)}$ unchanged. (The same principle is used in the SCR method to make the description of the software specification easier to change [Heit99].)



(a) device-dependent goal refinement



(b) device-independent goal refinement

FIGURE 6.10. Device-dependent vs. device-independent goal refinements

One may argue that such practice violates the principle requiring agent specifications to be stated in terms of interface variables only [Zav97]. Note however that this principle is applicable only when the interfaces between the agents are known. This is generally not the case during the early phase of the requirement elaboration process. Furthermore, the internal model that a software agent maintains about its environment is more stable than the interfaces of the agents, and determines the possible evolutions of the system [Jac83]. Therefore, favouring device-independent goal refinements in which subgoals are defined in terms of internal variables allows goals to be refined without too early commitments to specific interfaces, and alternative agent interfaces to be explored later on in the goal refinement process.

Note that once the agent interfaces are known, the internal variables can be declared as derived variables defined in terms of interface variables so that the requirement on the software and other agents are then defined in terms of interface variables only [Zav97].

Specializations: Specialized tactics for splitting lack of monitorability define specialized patterns for refining a goal so as to resolve lack of monitorability. These tactics vary according to the pattern of the goals G, G1, G2, and the nature of the intermediate variables i. These specializations are shown in Figure 6.11.

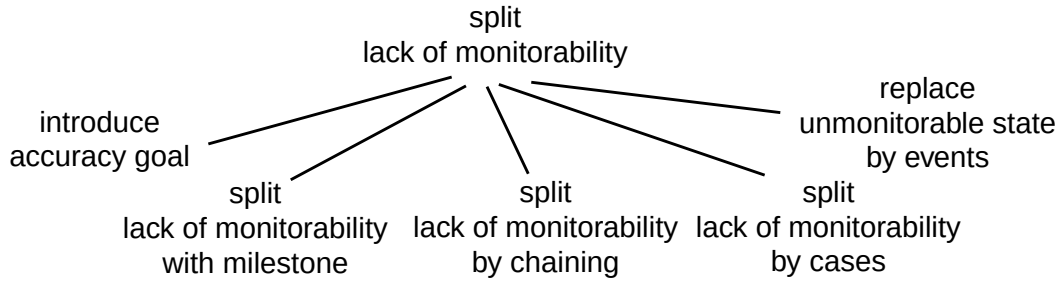


FIGURE 6.11. Specializations of the tactic split lack of monitorability

6.5.3. Introduce Accuracy Goals

We first discuss the basic tactic for splitting lack of monitorability by introduction of accuracy goals. (This tactic was already defined in Section 6.2.) Specialization and variants of this tactic are described next.

6.5.3.1. The basic tactic

The basic tactic introduce accuracy goal is defined in Section 6.2 (see in Figure 6.2). We make some further comments here.

Remark 1. The elaboration of the agent model for this tactic is inherited from the parent tactic split lack of monitorability. That is, a first way of elaborating the agent model is to identify a monitoring agent capable of monitoring m and controlling i, and assign the accuracy goal as responsibility of that agent. An alternative way of elaborating the agent model is to declare the intermediate variable as an internal variable.

Remark 2. The critical step in applying the tactic introduce accuracy goal is to identify the intermediate variable i. The choice of such a variable is ultimately constrained by the capabilities of agents available in the domain. Backtracking may be necessary if it latter turns out that available sensors are not sufficient for ensuring the accuracy goal relating i to m.

The tactic introduce accuracy goal *on variable* and introduce accuracy goal *on predicate* correspond to two alternative choices the intermediate variables.

> ***Example***: Remember the alternative refinements of the goal Maintain[PumpOnWhenHighWater] described in Section 6.3.2. The tactic introduce accuracy goal on variable ultimately results in assigning to a WaterSensor agent the responsibility of measuring the water level; the tactic introduce accuracy goal on predicate ultimately results in assigning to a HighWaterSensor agent the responsibility of detecting whether the water level is above some high measure.

Remark 3. Instead of an accuracy goal, one can also identify a domain property relating the unmonitorable quantity m to the intermediate variable i.

> ***Example [Jac93]:*** A goal in certain avionic systems is to ensure that reverse thrust can be engaged if, and only if, the plane is landing and already on the runway:
>
> RunningOnGround ⇔ RevThrustEnabled
>
> The goal is not realizable by the software agent because it cannot monitor the predicate RunningOnGround. The tactic introduce accuracy goal can then be applied by observing that when the plane is running on the ground, its wheels are turning. (This

property happens to be false when the plane is aquaplaning. We will come back to this problem in Chapter 8.) The refinement pattern introduce accuracy goal on predicate is then instantiated as follows:

P(m) : RunningOnGround *Q*(i) : WheelsTurning

The following assertions are thereby derived:

DomProp: RunningOnGround ⇔ WheelsTurning

SubGoal: WheelsTurning ⇔ RevThrustEnabled

Note that the accuracy property is declared in this example as a domain property rather than as a goal.

The resulting subgoal is itself not realizable because the software agent lacks monitorability for the predicate WheelsTurning. The tactic introduce accuracy domain property can again be used to resolve such lack of monitorability by introducing the variable WheelsPulse that is monitorable by the software agent, and using by the domain property

WheelsTurning ⇔ WheelsPulse = 'On'

to generate the subgoal

WheelsPulse = 'On ⇔ RevThrustEnabled

This subgoal is now realizable by the software agent.

6.5.3.2. Introduce tracking object

The tactic introduce tracking object is defined as follows.

Tactic introduce tracking object

Specialization Of introduce accuracy goal

Heuristics. This tactic should be considered when lack of monitorability for an object Obj can be resolved by maintaining an internal image of the object.

PostCondition.

1. Object model elaboration. If the unmonitorable variable m is an attribute of some object Obj, the intermediate variable i is modelled as an attribute of a new object ObjInfo denoting an internal image of the object Obj. A Tracking relationship is also introduced to relate the objects Obj and ObjInfo (see Figure 6.12).

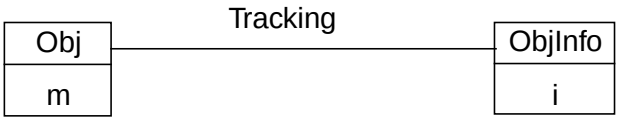


FIGURE 6.12. Modeling an object and its image

(This Tracking relationship is based on the same idea as the Mapping meta-relationship discussed in [Dar93]. As opposed to the latter, a Tracking relationship is *domain-dependent*. Such a domain-dependent relationship must be used in place of the Mapping meta-relationship. In this thesis, the Mapping meta-relationship has been suppressed from the KAOS meta-model.)

Examples. In the patient monitoring problem, lack of monitorability for attributes of the Patient entity is resolved by introducing the entity PatientInfo that denotes an image of the actual state of the patient. In an ambulance despatching problem, lack of monitorability for ambulance status and location is resolved by introducing an AmbulanceInfo object with the corresponding attributes (see Chapter 9); similarly, lack of monitorability for attributes of the Incident object can be resolved by introducing an IncidentInfo object.

The Tracking relationship is required to be a *one-to-one static relationship*, i.e. it is constrained by the following requirements:

(i) Tracking(oi, ob) $\Rightarrow \neg$ ($\exists$ oi'): oi' $\neq$ ob $\wedge$ Tracking(oi', ob)

(ii) Tracking(oi, ob) $\Rightarrow \neg$ ($\exists$ o'): ob' $\neq$ ob $\wedge$ Tracking(oi, ob')

(iii) $\forall$ ob ❑ $\exists$ oi Tracking(oi, ob)

(iii) $\forall$ oi ❑ $\exists$ ob Tracking(oi, ob)

(v) Tracking(oi, ob) $\Rightarrow$ ❑ Tracking(oi, ob)

The first four assertions constrain the cardinality of the relationship to be at most one on both sides. The fifth assertion requires Tracking to be a static relationship.

Typically, these requirements on the Tracking relationship are ensured by defining this relationship in terms of some domain attribute Attr that can act as an identifier of the object:

Tracking(oi, ob) $\Leftrightarrow$ oi.Attr = ob.Attr

where Attr is an injective and immutable attribute of the object.

Examples. In the patient monitoring problem, the Tracking relationship between Patient and PatientInfo may be defined in terms of the patient social security number. For the ambulance despatching problem, the Tracking relationship between Ambulance and AmbulanceInfo may be defined in terms of the ambulance license plate.

Theses attributes must not be confused with the 'internal' identities of the objects. It is important that the attributes used to define the Tracking relationship be observable by some agent in the domain. For instance, in the LAS problem, a Tracking relationship relating Incident and IncidentInfo must not be defined in terms of some unobservable identifier for the Incident object. A possible solution here consists in defining that two incidents are considered to be the same if they occur at the same place and time.

The problem of ensuring the above properties for the Tracking relationship may in some cases be fairly complex. In the LAS system, duplicate calls have to be appropriately detected to ensure accurate correspondence between Incident and IncidentInfo objects. For an air traffic control system, the problem of correlating data received from multiple noisy sensors in order to track airplanes is known to be complex [Jon2K].

This problem is also related to the notion of 'identities concern' [Jac2K].

2. Goal model elaboration. The unrealizable goal G is refined into the subgoals

Maintain[ObjectTracked]

Maintain[AccurateObjectInfo]

G {Object/ObjectInfo}

The *first* subgoal constrains the Tracking relationship to be a one-to-one static relationship as defined above. The *second* subgoal is an accuracy goal that relates an object and its image. The *third* subgoal is obtained roughly by replacing references to the actual object by references to the image of the object. Formally, one needs to uses variants of the basic formal goal refinement patterns for the tactic introduce accuracy goal in order to define that goal precisely.

Example 1. The refinement of the goal Achieve[AlarmForCriticalPulseRate] described in Section 6.3.1 is a first example of application of the tactic introduce tracking object.

Example 2. Consider an ambulance despatching system and the goal Achieve[AmbulanceMobilization] requiring that for every reported incident, an available ambulance whose location is close to the incident scene should be mobilized within 3 minutes.

This goal is unrealizable by an AmbulanceAllocator agent because it lacks of monitorability for the actual availability and location of ambulances. The tactic introduce tracking object can be used to resolve this lack of monitorability by introducing the new object AmbulanceInfo whose attributes record information about ambulance availability and location, and by refining the goal Achieve[AmbulanceMobilization] into:

Maintain[AmbulanceTracked]

Maintain[AccurateAmbulanceInfo]

Achieve[AmbulanceMobilizationBasedOnAmbulanceInfo]

The first subgoal requires every ambulance to have a corresponding AmbulanceInfo object. The second subgoal relates the values of the Available and Location attributes of the AmbulanceInfo object instances to the actual availability and location of the corresponding ambulances. The third subgoal requires ambulances to be mobilized based on information about availability and location held in the AmbulanceInfo objects.

6.5.3.3. Introduce sensor agent

The tactic introduce sensor agent is another specialization of the tactic introduce accuracy goal. It is defined as follows.

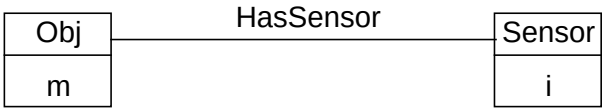
Tactic introduce sensor agent

Specialization Of introduce accuracy goal

Heuristics This tactic should be considered when lack of monitorability for an object Obj can be resolved by introducing a sensor agent capable of monitoring Obj.

PostCondition

1. Object model elaboration. If the unmonitorable variable m is an attribute of some object Obj, the intermediate variable i is modelled as an attribute of the Sensor agent. A relationship HasSensor is also introduced to relate the object Obj to the Sensor.



Example. Consider the patient monitoring problem, and the goal Maintain[AccuratePulseRateInfo]. An application of this tactic would generate the new agent PulseRateSensor, and the relationship HasPulseRateSensor linking Patient to PulseRateSensor.

The cardinality requirements on the relationship HasSensor will depend on the problem. However, one has to require that every object is measured by at least one sensor. The relationship can be static, but most often it is allowed to change over time.

Example. In the patient monitoring problem, the HasPulseRateSensor relationship is obviously not static.

2. Goal model elaboration. The unrealizable goal G is refined into the following three subgoals

Maintain[ObjectMeasuredBySensor]

Maintain[AccurateSensorMeasure]

G {m/ i}

The *first* subgoal constrains the HasSensor relationship. As mentioned above, it usually requires every object Obj to be connected to at least one sensor; the HasSensor relationship can be static or dynamic. The *second* subgoal is an accuracy goal concerning the measures of the sensor. The *third* subgoal is obtained roughly by replacing references to the unmonitorable variables by references to the measures of the sensor. Difficulties may arise when the HasSensor relationship is dynamic. The definition of formal refinement patterns to deal with such situations requires further work.

3. Agent model elaboration. The accuracy goal Maintain[AccurateSensorMeasure] is assigned to the Sensor agent. That agent is declared to monitor the variables m and to control the variables i.

Example. Consider the patient monitoring problem and the goal Maintain[AccuratePulseRateInfo]. As mentioned in Section 6.3.1, that goal is not realizable by the PatientMonitoring software agent because the latter lacks of monitorability for the actual pulse rate of patients. The tactic introduce sensor agent can then be used to generate the subgoals:

Maintain[PatientConnectedToPulseRateSensor]

Maintain[AccurateSensorMeasure]

Maintain[PulseRateInfoBasedOnSensorMeasure].

The agent PulseRateSensor is identified and is assigned to the goal Maintain[AccurateSensorMeasure].

6.5.3.4. Deidealizing accuracy goals through tolerances and delays

We favour a requirement elaboration process in which one starts by building requirements models based on ideal accuracy goals, and in a later step deidealize the models to take into account realizable accuracy goals involving tolerances and delays.

This section describes preliminary work for deidealizing accuracy goals. We propose (i) goal definition patterns for specifying accuracy goals involving tolerance and delays, and (ii) techniques for propagating such deidealization along goal refinement links.

Relevant techniques have been proposed in [Smi2K] for introducing tolerances and delays into an ideal requirements specification, and for deriving properties of the deidealized specification from those of the idealized specification. Further work is required to extend and adapt such techniques in the context of the KAOS goal-oriented method.

1. Formal definition patterns

A few patterns of goal definitions for accuracy goals involving tolerances and delays are shown in Table 6.1. Each pattern is described in turn and illustrated with an example.

	Formal Definition Pattern (accuracy goal on *variable*)	**Formal Definition Pattern (accuracy goal on *predicate*)**
ideal	❑ (i = m)	Q ⇔ *P*
tolerance (deviation)	❑ (m - dev ≤ i ≤ m + dev)	StongP ⇒ Q ∧ Q ⇒ *WeakP* with StongP ⇒ P, and *P* ⇒ *WeakP*
delay (periodic)	❑ ◊ $_{\leq d}$ (i = m)	■$_{\leq d}$ P ⇒ *Q* ∧ *Q* ⇒ ♦$_{\leq d}$ *P*

Table 6.1. Patterns of idealized and deidealized accuracy goals

Consider *accuracy goals on variable* first. An example of idealized accuracy goal on state variables for the mine pump control system is the goal

❑ (WaterMeasure = WaterLevel)

The first tolerance pattern for accuracy goal would state that the value of the variable i is always between m - dev and m + dev, where dev is the imprecision allowed between m and i. For the accuracy goal on the water level, it yields:

❑ (WaterLevel - Dev ≤ WaterMeasure ≤ WaterLevel + Dev)

where Dev is the imprecision allowed between the actual water level and the measured water level.

The delay pattern can be used to specify accuracy goals in which the variable i is updated periodically at least every d time units. For instance, if the water level is measured periodically at intervals of length Delay, the goal can be defined as follows:

❑ ◊ $_{\leq Delay}$ WaterMeasure = WaterLevel

We now consider formal definition patterns for *accuracy goals on predicates*. An example of an idealized accuracy goal on predicate for the mine pump control system is the goal

❑ (HighWaterSignal = ‘On’ ⇔ WaterLevel ≤ ‘High’)

The tolerance pattern can be used when the predicate Q is not equivalent to P but is bounded by predicates StrongP and WeakP which are stronger and weaker than P, respectively. For the accuracy goal on water level, the pattern can be instantiated as follows:

StrongP : WaterLevel ≤ 'High' + Dev

WeakP: WaterLevel ≥ 'High' - Dev

The following formal definition is thereby obtained:

WaterLevel ≥ 'High' + Dev ⇒ HighWaterSignal = 'On'
∧ HighWaterSignal = 'On' ⇒ WaterLevel ≥ 'High' - Dev.

The Delay pattern can be used when the value of the predicate *Q* is updated periodically at least every d time units. (Note that the formulae $\blacksquare_{\leq d}$ P and $\blacklozenge_{\leq d}$ *P* involved in the pattern are stronger and weaker than *P,* respectively.) The delay pattern yields the following definition for the accuracy goal on water level:

$\blacksquare_{\leq Delay}$ WaterLevel ≥ 'High' ⇒ HighWaterSignal = 'On'
∧ HighWaterSignal = 'On' ⇒ $\blacklozenge_{\leq Delay}$WaterLevel ≥ 'High'

Further alternative patterns for non-ideal accuracy goals involving tolerances and delays can be identified. The description of such patterns is subject to further work.

Goal-refinement with deidealized accuracy goals

We now consider the impact of deidealized accuracy goals on goal refinement links. There are three ways in which a goal refinement link can be modified to deal with deidealized accuracy goal:

(i) the parent goal can be weakened,

(ii) the companion subgoal can be strengthened, and

(iii) an assumption on the variable m can be added to the goal refinement link (generally, this assumption is a bound on the rate of change of m).

Examples. Consider the goal Maintain[PumpOnWhenHighWater]:

WaterLevel ≥ 'High' ⇒ PumpMotor = 'On'

and its refinement generated by the tactic introduce accuracy goal on variable:

❑ (WaterMeasure = WaterLevel) ,

WaterMeasure ≥ 'High' ⇒ PumpMotor = 'On'.

Assume now that the accuracy goal is deidealized to allow some tolerance with respect to the actual value of the water level:

❑ (WaterLevel - Dev ≤ WaterMeasure ≤ WaterLevel + Dev).

A first way to propagate this deidealization in the goal refinement consists in weakening the parent goal into:

WaterLevel ≥ 'High' **+ Dev** ⇒ PumpMotor = 'On',

Alternatively, one could also strengthen the companion subgoal into:

WaterMeasure ≥ 'High' **- Dev** ⇒ PumpMotor = 'On'.

To illustrate the adding of an assumption on variable m, assume now that the above ideal accuracy goal is deidealized to allow a delay between the measured and actual water levels:

$\Box \Diamond_{\leq Delay}$ WaterMeasure = WaterLevel.

One way to propagate this deidealization along the goal refinement link is to make the following assumption on the rate of change of the water level:

WaterLevel =lev $\Rightarrow \Box_{\leq delay}$ WaterLevel $\leq$ lev + Dev

(where Δ denotes the maximum increase of water level in delay time units), and to strengthen the companion subgoal into:

WaterMeasure $\geq$ 'High' - Dev $\Rightarrow$ PumpMotor = 'On'.

An alternative way to deidealize the same goal refinement link is to weaken the parent goal into:

$\blacksquare_{\leq delay}$ WaterLevel $\geq$ 'High' $\Rightarrow$ PumpMotor = 'On'.

Further work is needed to define and classify patterns for adding tolerances and delays in accuracy goals.

6.5.4. Split Lack of Monitorability with Milestone

The tactic split lack of monitorability with milestone can be used to resolve the lack of monitorability for a variable in the *antecedent* of an *Achieve* goal, by refining it according to some intermediate milestone.

Tactic split lack of monitorability with milestone

Motivation: resolve lack of monitorability

Precondition: the unrealizable goal is an *Achieve* goal of the form C $\Rightarrow \Diamond$ T and the agent lacks of monitorability for a variable appearing in the antecedent of that goal.

Heuristics: the tactic is worth being considered when an intermediate milestone M for reaching T from C can be identified.

Postcondition:

1. Object model elaboration: the object model is enriched with the new variables appearing in the definition of the milestone M

2. Goal model elaboration: the Achieve goal is refined according to the following milestone-driven goal refinement pattern [Dar95, Dar96]:

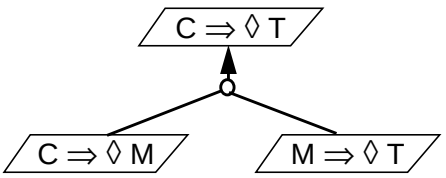


FIGURE 6.13. Split lack of monitorability with milestone

Variants of this basic milestone-driven goal refinement pattern are defined in Table 6.2 [Dar95]. (In this table, the symbols x and y denote logical variables, and should not be confused with state variables).

Parent Goal	Subgoal	Subgoal
C ⇒ ◊ T	C ⇒ ◊ M	M ⇒ ◊ T
C(x) ⇒ ◊ ∃ y T(x,y)	C(x) ⇒ ◊ M(x)	M(x) ⇒ ◊ ∃ y T(x,y)
C(x) ⇒ ◊ ∃ y T(x,y)	C(x) ⇒ ◊ ∃ y M(x,y)	M(x, y) ⇒ ◊ T(x,y)
C(x) ⇒ ◊ ∃ y T(x,y)	C(x) ⇒ ◊ ∃ z M(x,z)	M(x, z) ⇒ ◊ ∃ y T(x,y)

Table 6.2. Goal-refinement patterns for splitting lack of monitorability with milestone

(Note that not all milestone-driven goal refinement patterns of [Dar95] can be used to resolve lack of monitorability because some of them still refer to the unmonitorable antecedent in both subgoals.)

3. Agent model elaboration: the elaboration of the agent model is inherited from the parent tactic split lack of monitorability. That is, the intermediate milestone can be defined in terms of interface variables of an agent monitoring variables in C, or it can be defined in terms of internal variables of some agent.

Example 1. Consider the patient monitoring problem and the goal Achieve[NurseInterventionForCriticalPulseRate]:

p.PulseRate ∉ p.SafePulse ⇒ (∃ n: Nurse): Intervention(n,p)

That goal is unrealizable by the Nurse agent because the latter lacks of control for the patients pulse rate and safe pulse. The tactic split lack of monitorability with milestone can then be applied with the following milestone:

M: (∃ a: Alarm): a.Raised ∧ a.Loc = p.BedNbr

This yields the following two subgoals

Achieve[AlarmForCriticalPulseRate]

Achieve[NurseIntervetionForAlarm]

Example 2. Consider a resource allocation problem and the goal

Wishing(u,res) ⇒ ◊ Allocation(u, res).

The goal is not realizable by the ResourceAllocator agent, because the latter lacks of monitorability for the relationship Wishing(u,res).

The tactic split lack of monitorability with milestone can be used to resolve such lack of monitorability by identifying the following milestone:

M : Requesting(u,res),

thereby generating the subgoals

Achieve[ResourceRequested]

Achieve[RequestedResourceAllocated]

The relationship Requesting(u, res) is an interface variable controlled by the User agent and monitored by the ResourceAllocator. The first subgoal is therefore realizable by the User agent, and the second one by the ResourceAlloctor agent.

Specialisation: More specific guidance for identifying alternative milestones can be provided based on the category of the goal to be refined. The idea is be to identify typical alternative choices of milestone for each category of goal. For instance,

- ***Satisfaction goals***: the tactic introduce request milestone is used to resolve lack of monitorability for *agent wishes*;
- **S*afety goals***: the tactic introduce alarm milestone is used to resolve lack of monitorability for *hazardous states*.

The study of alternative goal refinement tactics for specific goal categories is subject to further work.

6.5.5. Split Lack of Monitorability by Chaining

The tactic split lack of monitorability by chaining is similar to the tactic split lack of monitorability with milestone, but is to be used with Maintain goals instead of Achieve goals. It is defined as follows.

Tactic split lack of monitorability by chaining

Motivation: resolve lack of monitorability

Precondition: the unrealizable goal is a Maintain goal of the form $P \Rightarrow Q$, $P \Rightarrow \Box Q$, or $P \Rightarrow Q\ W\ R$; and the agent lacks of monitorability for variables in P or R. (We assume that constrained variables are in Q).

Heuristics: the tactic is worth being used in the following cases:

(i) to resolve lack of monitorability for a variable in P when an intermediate condition M can be identified such that $P \Rightarrow M$;

(ii) to resolve lack of monitorability for a variable in R when an intermediate condition M can be identified such that $M \Rightarrow R$;

Postcondition:

1. Object model elaboration: the object model is enriched with new variables appearing in the definition of the new predicate M.

2. Goal model elaboration: the Maintain goal is refined according to one of the chain-driven goal refinement patterns in Table 6.3 [Dar95, Dar96].

Parent Goal	Subgoal	Subgoal
$P \Rightarrow Q$	$P \Rightarrow M$	$M \Rightarrow Q$
$P \Rightarrow \Box Q$	$P \Rightarrow M$	$M \Rightarrow \Box Q$
$P \Rightarrow Q\ W\ R$	$P \Rightarrow M$	$M \Rightarrow Q\ W\ R$
$P \Rightarrow Q\ W\ R$	$M \Rightarrow R$	$P \Rightarrow Q\ W\ M$

Table 6.3. Split lack of monitorability by chaining

The specialized refinement patterns in Table 6.4 are frequently used in the particular case where the Maintain goal has the general form:

$P(x) \Rightarrow \neg (\exists x'): x' \neq x \wedge P(x')$.

Parent Goal	Subgoal	Subgoal	Subgoal
$P(x) \Rightarrow$ $\neg (\exists x'): x' \neq x \wedge P(x')$	$P(x) \Rightarrow$ **M(x)**	**M(x)** $\Rightarrow$ $\neg (\exists x'): x' \neq x \wedge$ **M(x')**	--
$P(x) \Rightarrow$ $\neg (\exists x'): x' \neq x \wedge P(x')$	$P(x) \Rightarrow \bullet$ **M(x)**	**M(x)** $\Rightarrow$ $\neg (\exists x'): x' \neq x \wedge$ **M(x')**	$P(x) \rightarrow$ $\neg (\exists x'): x' \neq x \wedge P(x')$

Table 6.4. Chain-driven goal refinement patterns for $P(x) \Rightarrow \neg (\exists x'): x' \neq x \wedge P(x')$

3. agent model elaboration: the elaboration of the agent model is inherited from from the parent tactic split lack of monitorability.

Example 1. Consider the mine pump problem and the goal Maintain[PumpOnWhenHighWater]:

$WaterLevel \geq$ 'High' $\Rightarrow$ PumpMotor = 'On'

Lack of monitorability for the water level has been previously resolved by applying the tactic introduce accuracy goal to refine the goal into the subgoals Maintain[HighWaterDetected] and Maintain[PumpOnWhenHighWaterDetected].

A similar goal refinement can also be generated by applying the tactic split lack of monitorability by chaining with the following instantiation:

M: HighWaterSignal = 'On'

thereby generating the subgoals:

$WaterLevel \geq$ 'High' $\Rightarrow$ HighWaterSignal = 'On'

HighWaterSignal = 'On' $\Rightarrow$ PumpMotor = 'On'.

Example 2. Consider the resource allocation problem and the goal Avoid[SimultaneousUses]:

$Using(u,res) \Rightarrow \neg (\exists u'): u' \neq u \wedge Using(u', res)$

This goal is not realizable by User agents because the latter lacks of monitorability for the relationship Using(u', res) of other agents. The tactic split lack of monitorability by chaining is used to resolve that lack of monitorability by applying the first pattern in Table 6.4 with the following instantiation:

M: Allocation(u,res),

thereby generating the following two goals:

$Using(u,res) \Rightarrow Allocation(u,res)$

$Allocation(u,res) \Rightarrow \neg (\exists u'): u' \neq u \wedge Allocation(u', res)$.

6.5.6. Split Lack of Monitorability By Cases

The tactic split lack of monitorability by cases is defined as follows:

Tactic split lack of monitorability by cases

Motivation: resolve lack of monitorability

Precondition: the agent ag lacks monitorability for m in order to realize $G_{(M, C)}$

Heuristics: the tactic is worth being applied when different cases can be identified in which lack of monitorability for m can be resolved in specific ways for each case.

PostCondition:

1. Object model elaboration: the object model is enriched with new variables appearing in the definition of the different cases.

2. Goal model elaboration: different goal refinement patterns for splitting lack of monitorability by cases are given in Table 6.5 and Table 6.6. Patterns in Table 2 are to be used for splitting the antecedent of the goal; patterns in Table 3 are to be used for splitting the consequent of the goal. Most of these patterns were previously identified in [Dar95].

Parent Goal	Subgoal	Subgoal	Subgoal
$P1 \vee P2 \Rightarrow Q$	$P1 \Rightarrow Q$	$P2 \Rightarrow Q$	--
$P \Rightarrow Q$	$P \wedge \mathbf{C} \Rightarrow Q$	$P \wedge \neg \mathbf{C} \Rightarrow Q$	--
$P \Rightarrow Q$	$P \wedge \mathbf{C1} \Rightarrow Q$	$P \wedge \mathbf{C2} \Rightarrow Q$	❑ (**C1** $\vee$ **C2**)
$P \Rightarrow Q$	$P \wedge \mathbf{C1} \Rightarrow Q$	$P \wedge \mathbf{C2} \Rightarrow Q$	$P \Rightarrow$ **C1** $\vee$ **C2**
$P \Rightarrow Q$	**C1** $\Rightarrow Q$	**C2** $\Rightarrow Q$	$P \Rightarrow$ **C1** $\vee$ **C2**
$(\forall x)\ P(x) \Rightarrow Q(x)$	**C(x,y)** $\Rightarrow Q(x)$	--	$P(x) \Rightarrow \exists y$: **C(x,y)**
$(\forall x)\ P(x) \Rightarrow Q(x)$	$P(x) \wedge$ **C(x,y)** $\Rightarrow Q(x)$	--	$P(x) \Rightarrow \exists y$: **C(x,y)**
$(\forall x)\ P(x) \Rightarrow Q(x)$	$P(x) \wedge$ **C(x,y)** $\Rightarrow Q(x)$	--	$\forall x$ ❑ $\exists y$: **C(x,y)**
$(\forall x)\ P(x) \Rightarrow Q(x)$	$P(x) \wedge$ **C(x,y)** $\Rightarrow Q(x)$	--	$\forall x \exists y$: ❑ **C(x,y)**

Table 6.5. Case-driven goal refinement patterns: split antecedent

Parent Goal	Subgoal	Subgoal	Subgoal
$P \Rightarrow Q1 \wedge Q2$	$P \Rightarrow Q1$	$P \Rightarrow Q2$	--
$P \Rightarrow Q$	$P \Rightarrow$ **C1**	$P \Rightarrow$ **C2**	**C1** $\wedge$ **C2** $\Rightarrow Q$
$(\forall x)\ P(x) \Rightarrow Q(x)$	$P(x) \Rightarrow (\exists y)$: **C(x,y)**	--	**C(x,y)** $\Rightarrow Q(x)$

Table 6.6. Case-driven goal refinement patterns: split consequent

Example 1. Consider the following goal for the mine pump control system:

MethaneLevel ≥ 'MethaneCritical' ∨ COLevel ≥ 'COCritical' ⇒ Alarm

The goal is not realizable by the PumpController agent because it lacks of monitorability for the variables MethaneLevel and COLevel appearing in the antecedent of the goal. The tactic split lack of monitorability by cases can then be used by instantiating the first pattern in Table 6.5, thereby generating the two goals:

MethaneLevel ≥ 'MethaneCritical' ⇒ Alarm

COLevel ≥ 'COCritical' ⇒ Alarm

Lack of monitorability in each of these two goals can then handled separately.

Example 2. Consider the patient monitoring problem and the goal

Tracking(pi,p)
⇒
pi.Pulse = p.Pulse ∧ pi.SafePulse = p.SafePulse ∧ pi.BedNbr = p.BedNbr

The first refinement pattern in Table 6.6 refines that goal into the following three subgoals:

Tracking(pi,p) ⇒ pi.Pulse = p.Pulse

Tracking(pi,p) ⇒ pi.SafePulse = p.SafePulse

Tracking(pi,p) ⇒ pi.BedNbr = p.BedNbr

Lack of monitorability in each of these three goals can then handled separately.

Example 3. Consider the meeting scheduling problem and the goal Achieve[PrtcptsCstrKnown] defined by:

Intended(p,m) ⇒ ◊ CstrInfo[p,m].exclset = Cstr[p,m].exclset

The goal is not realizable by the Scheduler agent because the latter lacks of monitorability for the exclusion set of participants. Suppose that this set is to be obtained differently according to whether the participant maintains an electronic agenda or not. The tactic split lack of monitorability by cases can then be used by instantiating the second pattern in Table 6.5 with:

C: (∃ ag): HasAgenda(p, ag)

The following subgoals are thereby generated:

Intended(p,m) ∧ (∃ ag): HasAgenda(p, ag)
⇒ ◊ CstrInfo[p,m].exclset = Cstr[p,m].excset

Intended(p,m)∧ ¬ (∃ ag): HasAgenda(p, ag)
⇒ ◊ CstrInfo[p,m].exclset = Cst[p,m].excset

Different resolutions for the lack of monitorability of the participants' constraints can then be applied in each case.

Example 4. Consider a flight guidance system [Joh91] and the goal Achieve[PlaneBackOnCourse]:

pl.Loc ∉ pl.FlightPlan ⇒ ◊ pl.Loc ∈ pl.FlightPlan

The FGS (Flight Guidance System) software agent cannot realize the goal because it lacks monitorability for the location of planes. As a first step towards resolving such lack of monitorability, the goal is split into cases according to the region in which the plane is currently located. The tactic split lack of monitorability by cases is therefore applied by instantiating the eighth pattern of Table 6.5 with the following case:

C(x,y): InRegion(pl, fgs)

The following subgoals are thereby generated:

pl.Loc ∉ pl.FlightPlan ∧ InRegion(pl, fgs) ⇒ ◊ pl.Loc ∈ pl.FlightPlan

∀ pl ❑ ∃ fgs InRegion(pl, fgs)

6.5.7. Replace Unmonitorable States by Events

The tactic replace unmonitorable state by events provides a standard technique for transforming goals based on states into goals based on events. It is defined as follows.

Tactic replace unmonitorable state by events

Motivation: resolve lack of monitorability

Precondition: the agent lacks of monitorability for a predicate P in the definition of G.

Heuristics: the tactic can be used when one can identify two events startP and stopP that occur when P becomes true and false, respectively.

PostCondition:

1. Object model elaboration: the object model is enriched with the two events startP and stopP that occur when P becomes true and false, respectively:

startP ⇔ @ P
stopP ⇔ @ ¬ P

2. Goal model elaboration: Using the above domain properties, the goal G is refined into the subgoal

G{P/ ¬ stopP *Since* startP}

and the initial condition

initially → ¬ P.

(In practice, further simplification of the generated subgoals are sometimes necessary to make the formal definition more readable.)

Example 1. Consider the railroad crossing problem and the goal

(∃ tr: Train): $\blacksquare_{\leq d}$ InRegion(tr,cr) ⇒ cr.Gate = Closed

The predicate InRegion(tr,cr) is not monitorable by any agent in the application domain. However, the entry and exit of trains from the region are monitorable by sensor agents. The goal can then be refined by applying the above goal refinement pattern with the following instantiation:

P: InRegion(tr,cr)

startP: RegionEntry(tr,cr)

stopP: RegionExit(tr,cr)

Example 2. Consider the resource allocation problem and the goal:

Using(u, res) ⇒ Allocation(u, res)

The relationship Allocation(u, res) is not directly monitorable by the User agent, but is defined by the occurrences of AllocationEvent and ReleaseEvent at the interface between the User and the Allocator agents. One can then use the tactic replace unmonitorable state by events to generate the subgoal:

Using(u, res) ⇒ ¬ release_event(u,res) *S* allocation_event(u,res).

6.6. Resolving Lack of Control

We now detail tactics for resolving lack of control. These tactics are symmetric to the tactics for resolving lack of monitorability. Two tactics consist in adding the necessary control links to the agent or refining the goal further. The tactics add control and split lack of control are defined next. We then define further specializations of the tactic split lack of control.

6.6.1. Add control

Tactic add control

Motivation: resolve lack of control

Precondition: the agent ag lacks control of variable c in order to realize $G_{(M, C)}$

Heuristics: the tactic should be applied only if the variable can actually be controlled by the agent. Otherwise, one has to consider the alternative tactic split lack of control defined below.

PostCondition: a new control link is created between ag and c.

Example: Consider the meeting scheduling problem and the goal Achieve[ParticipantsCstrRequested] defined by:

Intended(p,m) ∧ Scheduling(sch,m)
⇒ ◊ (∃ cstr_req: CstrRequest):
cstr_req.Sent
∧ cstr_req.MeetingName = ● m.Name
∧ cstr_req.delivery_address = ● p.address

Let us assume that in the current agent interface model the Scheduler agent lacks of control for the object CstrRequest. The tactic add control resolves this by declaring the following control link:

Control [Scheduler, CstrRequest]
InstDecl: *Ctrl*(sch, cstr_req.Sent)

6.6.2. Split lack of control

The tactic split lack of control is symmetric to the tactic split lack of monitorability. It is defined as follows.

Tactic split lack of control

Motivation: resolve lack of control

Precondition: the agent ag lacks control of c in order to realize $G_{(M, C)}$

Heuristics: the tactic should be applied when m cannot be controlled by ag.

PostCondition:

1. Object model elaboration: The object model is enriched with a new intermediate variable o.

2. Goal model elaboration: The goal G is refined into the subgoals $G2_{(o, c)}$, that constrain the values of c based on the values of o, and $G1_{(M, (C\setminus c)\cup o)}$ (see Figure 6.14). Note that G1 does not refer to the uncontrollable variable c.

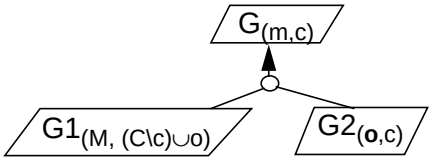


FIGURE 6.14. Split lack of control

(This description is slightly simplified. Tactics for resolving lack of control may sometimes refine G with more than two subgoals.)

3. Agent model elaboration: The new intermediate variables i must be controllable by ag. This ensures that the subgoal G2 refers to less uncontrollable variables than the parent goal G.

A first way to elaborate the agent model consists in identifying a *controlling agent* capable of controlling c and monitoring o, and in assigning the goal $G2_{(\mathbf{o},c)}$ to that agent (Figure 6.15). In the context of control systems, actuators are typical examples of such controlling agents.

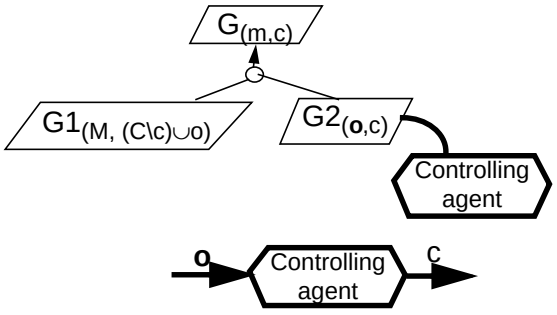


FIGURE 6.15. Split lack of control with controlling agent

Alternatively, the new intermediate variable o can also be declared as an internal variable. This alternative elaboration of the agent model makes the goal refinement graph more robust with respect to changes of controlling agent or agent interfaces (see Section 6.5.2).

Note that the tactics split lack of monitorability and split lack of control are symmetric. Splitting lack of control for one agent often corresponds to splitting lack of monitorability for another agent.

Example. Consider the mine pump control system again and the goal Maintain[PumpOnWhenHighWater]:

HighWaterSignal = 'On' ⇒ PumpMotor = 'On'

The variable PumpMotor is an environment quantity that is not directly controlled by the PumpController software agent.

An application of the tactic split lack of control consists in elaborating the model as follows:

1. Object model elaboration: a new variable PumpSwitch that denotes a switch to command the pump is identified;

2. Goal model elaboration: the goal is refined into the subgoals PumpSwitchOnWhenHighWater and PumpOnWhenPumpSwicthOn, defined as follows:

HighWaterSignal = 'On' ⇒ **PumpSwitch** = 'On'

PumpSwitch = 'On' ⇒ PumpMotor = 'On'

3. Agent model elaboration: a PumpActuator agent controlling the pump motor and monitoring the pump switch is introduced; it is assigned to the goal PumpOnWhenPumpSwicthOn.

Specializations: Specialized tactics for splitting lack of control define specialized patterns for refining a goal so as to resolve lack of control of agents. These tactics vary according to the pattern of the goals G, G1, G2, and the nature of the intermediate variable o. Specialized tactics for resolving lack of control are shown in Figure 6.16. These tactics are symmetric to the tactics for resolving lack of monitorability.

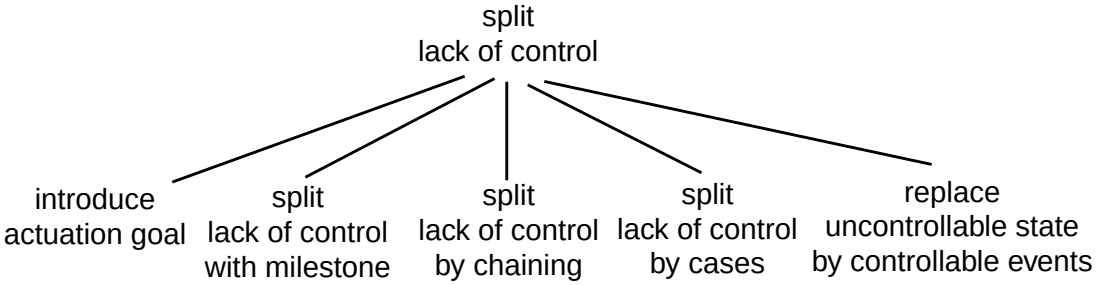


FIGURE 6.16. Specializations of the tactic split lack of control

6.6.3. Introduce Actuation Goals

The tactic introduce actuation goal is symmetric to the tactic introduce accuracy goal. It is defined as follows.

Tactic introduce actuation goal

Motivation: resolve lack of control

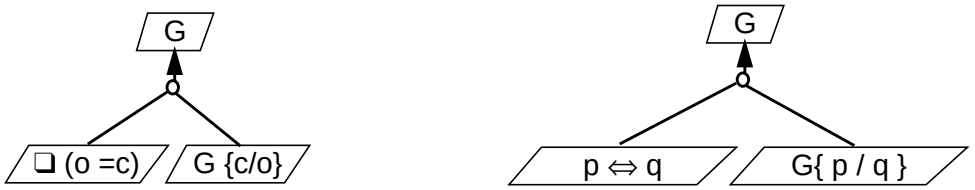
Precondition: the agent ag lacks of control for c in order to realize $G_{(M, C)}$

Heuristics: the tactic should be applied when one can identify an intermediate variable o that can be related to c through some actuation goal. (The property relating o to c can also be a domain property.)

PostCondition:

1. Object model elaboration: The object model is enriched with a new variable o that denotes a quantity used to control c.

2. Goal model elaboration: The unrealizable goal is refined into an actuation goal relating o to c, and a companion subgoal whose definition refers to the variables o instead of c. Two alternative formal goal refinement patterns for the tactic are shown in Figure 6.17.



(a) introduce actuation goal *on variable* (b) introduce actuation goal *on predicate*

FIGURE 6.17. Introduce actuation goal

The patterns introduce actuation goal *on variable* and introduce actuation goal *on predicate* correspond to alternative ways to apply the tactic.

Variants of these patterns dealing with non-ideal actuation goals involving tolerance and delays are discussed in Section 6.6.3.1 below.

3. Agent model elaboration: The elaboration of the agent model is inherited from the parent tactic split lack of control.

Example 1. The refinement of the goal Maintain[PumpOnWhenHighWater] in Section 6.6.2 corresponds to an application of the tactic introduce actuation goal *on predicate* with the following instantiation:

p: PumpMotor = 'On' q: PumpSwitch = 'On'.

Example 2. Consider a train control system and the goal Maintain[SafeAcceleration] requiring the acceleration of a train to be less than some value, the latter being a function of its speed and distance with the preceding train:

Following(tr1, tr2) ⇒ tr1.Acc ≤ F(tr1.Loc, tr2.Loc, tr1.Speed)

This goal is unrealizable by a TrainController agent, because the latter lacks of control for the actual acceleration of the train. An application of the tactic introduce actuation goal can be used to resolve such lack of control by identifying the variable Train.AccCmd denoting the command used to control the acceleration of the train; and refining the goal into:

Following(tr1, tr2) ⇒ tr1.AccCmd ≤ F(tr1.Loc, tr2.Loc, tr1.Speed)

❑ (tr.Acc = tr.AccCmd)

This refinement is of course too ideal. The relation between a train acceleration and its acceleration command is more complex than a simple equality. More precise definitions for these goals will be given for the BART case study in Chapter 9.

6.6.3.1. Deidealizing actuation goals through tolerances and delays

The specification of tolerance and delays for actuation goals is similar to the specification of tolerances and delays for accuracy goals. The formal definition patterns in Table 6.1 can be used to specify such goals (see Section 6.5.3.4).

Example 1. Consider the ideal actuation goal for the mine pump control system:

$\Box$ (PumpMotor = 'On' $\Leftrightarrow$ PumpSwitch = 'On')

The delay pattern can be used to specify the delay needed for the pump motor to be on once the pump switch is set to 'On'. The deidealized goal can be formalized as follows:

$\blacksquare_{\leq pump\_delay}$ PumpSwitch = 'On' $\Rightarrow$ PumpMotor = 'On'

$\wedge$ PumpSwitch = 'Off' $\Rightarrow$ PumpMotor = 'Off'

The first assertion says that the pump motor is 'On' when the pump switch has been 'On' during the last pump_delay time units; the second assertion says that the pump motor goes immediately off when the pump switch is turned off.

Example 2. Consider the railroad crossing problem and the goal Maintain[GateClosedWhenTrainCrossing]:

InCrossing(tr, cr) $\Rightarrow$ cr.Gate = 'Closed'

The goal is not realizable by the GateController software agent because it lacks of control for the variable cr.Gate. Such lack of control, can be resolved by introducing the variable cr.GateSignal and an actuation goal with delay:

$\blacksquare_{\leq gate\_delay}$ cr.GateSignal = 'down' $\Rightarrow$ cr.gate = 'Closed'.

The companion subgoal refining the goal is then defined by:

InCrossing(tr, cr) $\Rightarrow \blacksquare_{\leq gate\_delay}$ cr.GateSignal = 'down'

6.6.4. Split Lack of Control with Milestone

The tactic split lack of control with milestone can be used to refine an Achieve goal so as to resolve the lack of control for a variable appearing in the *consequent* of the goal.

Tactic split lack of control with milestone

Motivation: resolve lack of control

Precondition: the unrealizable goal is an *Achieve* goal of the form C $\Rightarrow \Diamond$ T and the agent lacks of control for a variable appearing in the consequent of that goal.

Heuristics: the tactic is worth being considered when an intermediate milestone M for reaching T from C can be identified.

Postcondition:

1. Object model elaboration: the object model is enriched with the new variables appearing in the definition of the milestone M

2. Goal model elaboration: the Achieve goal is refined according to the following milestone-driven goal refinement pattern [Dar95, Dar96]:

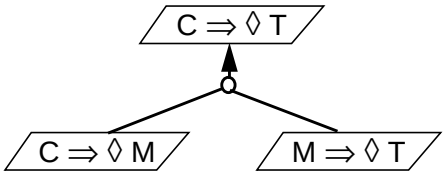


FIGURE 6.18. Split lack of control with milestone

Variants of this basic milestone-driven goal refinement patterns in Table 6.3, given there for splitting lack of monitorability with milestone, can also be used for splitting lack of control with milestone.

3. Agent model elaboration: the elaboration of the agent model is inherited from from the parent tactic split lack of control.

Example 1. Consider the patient monitoring problem and the goal Achieve[NurseInterventionForCriticalPulseRate]:

p.PulseRate ∉ p.SafePulse ⇒ (∃ n: Nurse): Intervention(n,p)

This goal is unrealizable by the PatientMonitoring software agent, because the latter lacks of control for the nurse intervention. The tactic split lack of control with milestone can be applied with the following milestone:

M: (∃ a: Alarm): a.Raised ∧ a.Loc = p.BedNbr

to generate the subgoals

Achieve[AlarmForCriticalPulseRate]

Achieve[NurseIntervetionForAlarm]

Note that the same goal refinement had been generated in Section 6.5.4 by application of the tactic split lack of *monitorability* with milestone; the latter was used for resolving lack of monitorability of Nurse agents on patients' pulse rates.

Example 2. Consider the meeting scheduling problem and the goal Achieve[PrtctsPresenceAtConvenientMeeting]:

Intended(p.m) ∧ m.Planned ∧ Convenient(p,m) ⇒ ◊ Participates(p,m)

This goal is not realizable by the MeetingScheduler agent because the latter lacks of control for the relationship Participates. The tactic split lack of control with milestone can be used to resolve such lack of control with the intermediate milestone

M: Informed(p,m)

thereby generating the subgoals

Achieve[PrtcptsInformed]

Achieve[InformedPrctptsPresenceAtConvenientMeeting].

Again, the same goal refinement can be obtained by resolving lack of monitorability of Participant agent on variables in the antecedent of the goal.

6.6.5. Split Lack of Control by Chaining

The tactic split lack of control by chaining is symmetric to the tactic split lack of monitorability by chaining.

Tactic split lack of control by chaining

Motivation resolve lack of control

Precondition: the unrealizable goal is a Maintain goal of the form P ⇒ Q, P ⇒ ❑ Q, or P ⇒ Q *W* R; and the agent lacks of control on variables referenced in Q. (We assume that constrained variables are in Q).

Heuristics: the tactic is worth being considered for resolving lack of control on a variable in Q when an intermediate condition M can be identified such that M ⇒ Q.

Postcondition:

1. Object model elaboration: the object model is enriched with the new variables appearing in the definition of the new predicate M.

2. Goal model elaboration: the Maintain goal can be refined by using one of the chain-driven goal refinement patterns in Table 6.7 [Dar95, Dar96]. (Note that these patterns are slightly different from those used to split lack of monitorability by chaining.)

Parent Goal	Subgoal	Subgoal
P ⇒ Q	P ⇒ M	M ⇒ Q
P ⇒ ❑ Q	P ⇒ ❑ M	M ⇒ Q
P ⇒ Q *W* R	P ⇒ M *W* R	M ⇒ Q

Table 6.7. Split lack of control by chaining

3. Agent model elaboration: the elaboration of the agent model is inherited from the parent tactic split lack of control.

Example. Consider the railroad crossing problem and the high-level goal Avoid[Train/CarCollisions]:

TrainInCrossing(tr, cr) ⇒ ¬ (∃ c: Car): CarInCrossing(tr, cr)

The goal is not realizable by the GateController agent, because the latter lacks of control on the relationship CarInCrossing. The tactic split lack of control by chaining can applied with the following instantiation:

M: cr.Closed

to generate the subgoals

Maintain[GateClosedWhenTrainInCrossing]

Maintain[NoCarInClosedCrossing].

6.6.6. Split Lack of Control By Cases

The tactic split lack of control by cases is symmetric to the tactic split lack of monitorability by cases.

Tactic split lack of control by cases

Motivation: resolve lack of control

Precondition: the agent ag lacks of control for c in order to realize $G_{(M, C)}$

Heuristics: the tactic is worth being applied when different cases can be identified in which lack of control for c can be resolved in specific ways for each case.

PostCondition:

1. Object model elaboration: the object model is enriched with the new variables appearing in the definition of the different cases.

2. Goal model elaboration: the case-driven goal refinement patterns of Table 6.5 and 6.6, given there for splitting lack of monitorability by cases, can also be used for splitting lack of control by cases.

Example. Consider the ambulance dispatching system and the goal Achieve[AllocatedAmbulanceMobilized]:

$$\text{Allocation(amb, inc)} \Rightarrow \Diamond_{\leq d}\ \text{Mobilization(amb, inc)}$$

The Mobilization relationship is controlled by AmbulanceStaff agents and not directly by an AmbulanceAllocator agent. The goal is therefore not realizable by the AmbulanceAllocator agent because the latter lacks of control on that relationship. Since the mobilization of ambulances has to be handled differently dependent on whether the ambulance is waiting at a station or somewhere on the road, the tactic split lack of control by cases is applied by instantiating the second pattern in Table 6 as follows:

C1: amb.OnRoad C2: $(\exists$ st: Station): AtStation(amb, st)

The following subgoals are thereby generated:

$$\text{Allocation(amb, inc)} \wedge \text{AtStation(amb, st)} \Rightarrow \Diamond_{\leq d}\ \text{Mobilization(amb, inc)}\ ,$$

$$\text{Allocation(amb, inc)} \wedge \text{amb.OnRoad} \Rightarrow \Diamond_{\leq d}\ \text{Mobilization(amb, inc)}\ ,$$

together with the domain property:

$$\Box\ (\ \text{amb.OnRoad} \vee (\exists\ \text{st: Station}): \text{AtStation(amb, st)}\ ).$$

6.6.7. Replace Uncontrollable State by Events

The tactic replace uncontrollable state by events is symmetric the tactic replace unmonitorable state by events. It is defined as follows.

Tactic replace uncontrollable state by events

Motivation: resolve lack of control

Precondition: the agent lacks of control on variables in a predicate P occurring in the definition of G.

Heuristics: the tactic can be used when two events startP and stopP can be identified which occur when P becomes true and false, respectively.

PostCondition:

1. Object model elaboration: the object model is enriched with two events startP and stopP that occur when P becomes true and false, respectively:

$$\text{startP} \Leftrightarrow @\ P$$
$$\text{stopP} \Leftrightarrow @\ \neg\ P$$

2. Goal model elaboration: Using the above domain properties, the goal G is refined into the subgoal

$$G\{P/\ \neg\ \text{stopP}\ \textit{Since}\ \text{startP}\}$$

and the initial condition

$$\text{initially} \rightarrow \neg\ P.$$

(In practice, further simplification of the generated subgoals may sometimes be necessary to make the formal definition more readable.)

6.7. Resolve Goal Unsatisfiability

There are two tactics for resolving goal unsatisfiability: weaken goal with unsatisfiability condition and prevent unsatisfiability. The former consists in weakening the goal so as to cover the unsatisfiability condition; the latter consists in refining the goal by requiring the unsatisfiability condition to be avoided.

6.7.1. Weaken goal with unsatisfiability condition

Tactic weaken goal with unsatisfiability condition

Motivation: resolve goal unsatisfiability

Precondition: $G_{(M,\ C)}$ is unsatisfiable when the unsatisfiability condition B holds.

Heuristics: the tactic is worth being applied when G is not safety-critical and the unsatisfiability condition can be tolerated.

PostCondition: The goal definition is weakened into $G \vee B$. The weakening of the goal has then to be propagated along the goal refinement links.

Example. Consider the meeting scheduling problem and the goal Achieve[ConvenientMeetingPlanned] defined by:

$$\text{m.Requested}$$
$$\Rightarrow \Diamond\ (\exists\ \text{d: Date}):$$
$$(\ \text{m.Date} = \text{d} \wedge \text{d} \in \text{m.DateRange}$$
$$\wedge\ \bullet\ (\forall\ \text{p: Participant}):\ \text{Intended(p,m)} \rightarrow \text{d} \notin \text{Constaint[p,m].exclset}\ )$$

This goal is unsatisfiable if there is no date inside the date range that satisfies all participants' constraints. The unsatisfiability condition for this goal is thus;

◊ (∃ m: Meeting):
m.Requested
∧ ❑ ¬ (∃ d: Date): d ∈ m.DateRange
∧(∀ p: Participant): Intended(p,m) → d ∉ Constaint[p,m].exclset

The tactic weaken unsatisfiability allows the problem to be solved by weakening the goal definition into:

m.Requested ⇒
(◊ (∃ d: Date): m.Date = d ∧ d ∈ m.DateRange
∧ ● (∀ p: Participant): Intended(p,m) → d ∉ Constaint[p,m].exclset)
∨ ❑ ¬ m.Feasible

The predicate m.Feasible in this weakened formula is defined by:

m.Feasible ⇔
¬ (∃ d: Date): d ∈ m.DateRange
∧ (∀ p: Participant): Intended(p,m) → d ∉ Constaint[p,m].exclset

6.7.2. Prevent goal unsatisfiability

Tactic prevent goal unsatisfiability

Motivation: resolve goal unsatisfiability

Precondition: $G_{(M, C)}$ is unsatisfiable when the unsatisfiability condition B holds.

Heuristics: the tactic is worth being applied when G is safety-critical and the unsatisfiability condition cannot be tolerated.

PostCondition: The unsatisfiable goal is refined according to the refinement pattern in Figure 6.19.

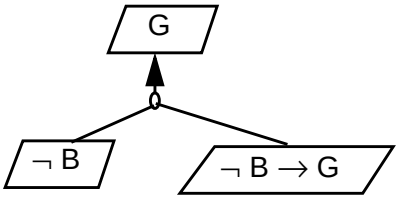


FIGURE 6.19. Prevent goal unsatisfiability

This pattern captures the general idea of the tactic. In practice, the formal definitions generated by the strict application of this pattern may have to be adapted.

Example. Consider an ambulance dispatching system and the goal Achieve[AmbulanceMobilizedInSector] defined by:

inc.Reported ∧ InSector(inc, s)
⇒ $◊_{≤3m}$ (∃ amb: Ambulance):
Mobilization(amb, inc) ∧ ● (amb.Available ∧ InSector(amb, s))

The domain of goal unsatisfiability of the goal is defined by:

◊ (∃ inc: Incident, s: Sector):
inc.Reported ∧ InSector(inc, s)
∧ $❑_{≤3m}$ ¬ (∃ amb: Ambulance): amb.Available ∧ InSector(amb, s)

Since the goal is safety critical, the tactic prevent unsatisfiability is used to refine it. The negation of the unsatisfiability condition is given by

inc.Reported ∧ InSector(inc, s)
⇒ $\Diamond_{\leq 3m}$ (∃ amb: Ambulance):
amb.Available ∧ InSector(amb, s))

This assertion is then strengthened into the goal Maintain[AvailableAmbulanceInSector], requiring that in every sector there is always an ambulance available.

❑ (∃ amb: Ambulance): amb.Available ∧ InSector(amb, s)

The companion subgoal generated by the tactic is:

inc.Reported ∧ InSector(inc, s)
⇒ $\Diamond_{\leq 3m}$
(∃ amb: Ambulance): Mobilization(amb, inc) ∧ ● (amb.Available ∧ InSector(amb, s))
∨
¬ (∃ amb: Ambulance): amb.Available ∧ InSector(amb, s)

This goal requires an available ambulance to be mobilized from the sector in which the incident occurred, except if there is no ambulance available in that sector.

6.8. Resolve References to the Future

We now consider tactics for resolving references to the future. There are tactics for resolving references to the strict future and tactics for resolving synchronization problems.

6.8.1. Resolve References to Strict Future

6.8.1.1. Apply anticipation pattern

References to the future can be resolved by identifying a condition that anticipates the future values of monitored variables.

Tactic apply anticipation pattern

Motivation: resolve references to the future

Precondition: the goal constrains variables in terms of future values of monitored variables

Heuristics: the tactic should be considered when a condition A can be identified that anticipates future values of monitored variables (this condition will be called anticipation condition).

Postcondition:

1. Object model elaboration: the object model is enriched with the new variables appearing in the definition of the anticipation condition A.

2. Goal model elaboration: the goal is refined into a first subgoal (or domain property) that relates the condition on future states of monitored variables to the anticipation condition, and a second subgoal that constrains controlled variables based on the anticipation condition.

The following (semantically equivalent) 'anticipation' formulas can be used to refine the goal by anticipating the condition M on future states of monitored variables:

(i) $M \Rightarrow \blacksquare_{\leq d'} A$ (ii) $\blacklozenge_{\leq d'} \neg A \Rightarrow \neg M$
(iii) $\Diamond_{\leq d'} M \Rightarrow A$ (iv) $\neg A \Rightarrow \square_{\leq d'} \neg M$

The first formula says that if M holds then the anticipation condition must have been true for a least d' time unit. The fourth formula says that if the anticipation condition does not hold currently, then M will not hold during the next d' time units. These assertions are all semantically equivalent; they define the same set of histories.

Table 6.8 shows some anticipation-driven refinement patterns that use these anticipation formulas.

Parent Goal	Subgoal	Subgoal	Subgoal
$\square_{\leq d} M \Rightarrow C$	$M \Rightarrow \blacksquare_{\leq d'} A$	$\blacksquare_{\leq d'-d} A \Rightarrow C$	$\square (d' \geq d)$
$\Diamond_{\leq d} M \Rightarrow C$	$M \Rightarrow \blacksquare_{\leq d'} A$	$\blacksquare_{\leq d'-d} A \Rightarrow C$	$\square (d' \geq d)$
$M \Rightarrow \blacksquare_{\leq d} C$	$M \Rightarrow \blacksquare_{\leq d'} A$	$\blacksquare_{\leq d'-d} A \Rightarrow C$	$\square (d' \geq d)$

Table 6.8. Anticipation-driven refinement patterns

Example. Consider the railroad crossing problem and the 'utility' goal Maintain[GateOpenWhenNoTrain]:

$\square_{\leq d} \neg$ (∃ tr: Train): InCrossing(tr,cr) ⇒ cr.Gate = 'opened'

This goal constrains the value of the variable cr.Gate based on future values of the variable InCrossing(tr,cr).

Such a reference to the future is resolved by using the property that if a train is in the crossing, it must have been in the region of the crossing during some time d' that depends on the maximum speed of trains. One can apply the tactic apply anticipation pattern by instantiating the first pattern in Table 6.8 as follows:

A: InRegion(tr,cr)

The two subgoals generated by the pattern are then:

InCrossing(tr,cr) ⇒ $\blacksquare_{\leq d'}$ InRegion(tr,cr)

$\blacksquare_{\leq d'-d}$ InRegion(tr,cr) ⇒ cr.Gate = 'opened'

6.8.2. Resolve Synchronization problems

We now consider tactics for resolving synchronization problems (that is, a goal constrains the values of controlled variables based on the *current* values of monitored variables). The alternative tactics are:

- replace current by previous;
- introduce reactiveness hypothesis;
- introduce mutual exclusion assumption;
- apply mutual exclusion refinement pattern;
- apply anticipation refinement pattern.

These tactics are described successively.

6.8.2.1. Replace current by previous

Tactic replace current by previous

Motivation: resolve synchronization problem

Precondition: the goal constrains the values of controlled variables based on the current values of monitored variables m.

Heuristics: this tactic should be considered when the definition of the goal is temporally too strong.

Postcondition: The tactic consists in weakening the goal definition by replacing every occurrence of the problematic monitored variable m by the value of this variable in the previous state. The weakened goal definition is then given by G{m/ ● m}. This weakening of goal definition must then be propagated along the refinement links of the goal graph.

Example 1. Consider the mine pump control system and the goal Maintain[PumpSwitchOnWhenHighWaterDetected]:

HighWaterSignal = 'On' ⇒ PumpSwitch = 'On'

An application of the tactic replace current by previous yields the following goal definition:

● HighWaterSignal = 'On' ⇒ PumpSwitch = 'On'

Example 2. Consider the meeting scheduling problem and the goal Achieve[ConvenientMeetingPlanned] whose definition is given by:

m.Requested
⇒ ◊ (∃ d: Date):
(m.Date = d ∧ (∀ p: Participant): Intended(p,m) → d ∉ Constaint[p,m].exclset)

An application of the tactic replace current by previous yields the following goal definition:

m.Requested
⇒ ◊ (∃ d: Date):
(m.Date = d ∧ ● (∀ p: Participant): Intended(p,m) → d ∉ Constaint[p,m].exclset)

6.8.2.2. Introduce reactiveness hypothesis

The tactic introduce reactiveness hypothesis corresponds to a standard technique for resolving synchronization problems. It consists in assuming that an agent, usually a software one can react infinitely fast to changes of variables in its environment. Such hypothesis is a built-in one in the semantics of specification languages such as SCR [Heit96].

Such hypothesis is inconsistent with our formal model of agents, and can therefore not be modelled in our framework. Therefore, the tactic introduce reactivity hypothesis consists in ignoring the synchronization problem by assuming that the agent can react infinitely fast to changes of monitored variables.

For example, consider again the goal Maintain[PumpSwitchOnWhenHighWater] defined by

HighWaterSignal = 'On' ⇒ PumpSwitch = 'On'.

The tactic introduce reactiveness hypothesis consists here in assuming implicitly that the PumpController agent can react infinitely fast to changes of the HighWaterSignal variable.

6.8.2.3. Introduce mutual exclusion hypothesis

The tactic introduce mutual exclusion hypothesis consists in refining the goal by assuming that changes of monitored and controlled variables cannot occur simultaneously.

Tactic introduce mutual exclusion hypothesis

Motivation: resolve synchronization problem

Precondition: the goal constrains the values of controlled variables based on the current values of monitored variables m.

Heuristics: this tactic should be considered when it can be *assumed* that states transitions for monitored and controlled variables never occur simultaneously.

Postcondition: For a goal of the form $P \Rightarrow Q$, this tactic consists in applying the formal goal refinement pattern in Table 6.9.

Parent Goal	Subgoal	Subgoal	Subgoal	DomProp/Ass
$P \Rightarrow Q$	$P \rightarrow Q$	$@P \Rightarrow \bullet Q$	$@\neg Q \Rightarrow \bullet \neg P$	$\Box \neg (@P \wedge @\neg Q)$

Table 6.9. Introduce mutual exclusion hypothesis

Example. Consider a system to control the opening of train doors, and the safety goal Maintain[DoorsClosedWhileMoving]:

$tr.Moving \Rightarrow tr.DoorsState = \text{‘Closed’}$

Suppose the variable tr.Moving is controlled by some agent, say the TrainDriver agent, whereas the variable tr.DoorsState is controlled by some other agent, say the DoorCtrler agent. The DoorCtrler agent cannot realize the goal because the latter constrains the variable tr.DoorState based on the current value of the variable tr.Moving.

The tactic introduce mutual exclusion hypothesis consists in making the assumption that the opening of doors cannot occur simultaneously with the start of trains:

$\Box \neg (@\ tr.Moving \wedge @\ tr.DoorsState \neq \text{‘Closed’})$

thereby generating the two subgoals

$@\ tr.Moving \Rightarrow \bullet\ tr.DoorsState = \text{‘Closed’}$

$@\ tr.DoorsState \neq \text{‘Closed’} \Rightarrow \bullet \neg\ tr.Moving$

together with an initial condition:

$tr.Moving \rightarrow tr.DoorsState = \text{‘Closed’}$

The first subgoal is then realizable by the TrainDriver agent, whereas the second one by the DoorCtrler agent.

Note: If we apply the alternative tactic introduce reactiveness hypotheses, the goal Maintain[DoorsClosedWhileMoving] could be assigned as responsibility of the DoorCtrler agent alone; one should then be assume that this agent can close the doors infinitely

fast when the train starts moving. Alternatively, the goal could be assigned as responsibility of the TrainDriver agent only; one should then assume that this agent can stop the train infinitely fast when the doors get open. In both cases, the assumptions are highly unrealistic.

6.8.2.4. Apply mutual exclusion refinement pattern

The tactic apply mutual exclusion pattern is to be used to resolve synchronization problems by refining the goal so as to prevent simultaneous state transitions of monitored and controlled variables.

Tactic apply mutual exclusion pattern

Motivation: resolving synchronization problem

Heuristics: this tactic should be considered for refining the goal by *requiring* (instead of assuming) state transition of monitored and controlled variables to never occur simultaneously.

Precondition: the goal constrains the values of controlled variables based on the current values of monitored variables m.

Postcondition: for a goal of the form $P \Rightarrow Q$, this tactic consists in applying the formal goal refinement pattern in Table 6.10. This pattern is reminiscent of standard mutual exclusion schemes, with the predicate T playing the role of the variable 'turn' indicating which processor is allowed to enter its critical section.

Parent Goal	Subgoal	Subgoal	Subgoal
$P \Rightarrow Q$	$P \rightarrow Q$	$@P \Rightarrow \bullet (Q \wedge T)$	$@\neg Q \Rightarrow \bullet (\neg P \wedge \neg T)$

Table 6.10. Mutual-exclusion refinement pattern

Example. Consider again the goal Maintain[DoorsClosedWhileMoving]. The tactic apply mutual exclusion pattern can be used to resolve the synchronization problem discussed before by introducing the following variable:

T: GoSignal

The variable GoSignal would correspond to a signal between the TrainDriver and DoorCtrler agents or to a signal controlled by some other agent at the station where the train is stopped. The following subgoals are thereby generated:

@ tr.Moving $\Rightarrow \bullet$ (tr.DoorsState = 'Closed' $\wedge$ GoSignal)

@ tr.DoorsState $\neq$ 'Closed' $\Rightarrow \bullet$ ($\neg$tr.Moving $\wedge \neg$ GoSignal)

together with the initial condition:

tr.Moving $\rightarrow$ tr.DoorsState = 'Closed'

If the new variable GoSignal is monitorable by the TrainDriver and DoorCtrler agents, the first subgoal is realizable by the TrainDriver, and the second one by the DoorCtrler.

6.8.2.5. Apply anticipation pattern

The tactic apply anticipation pattern defined to resolve reference to strict future can also be used to resolve synchronization problems. Two anticipation-driven goal refinement patterns for goals of the form $P \Rightarrow Q$ are defined in Table 6.11.

Parent Goal	Subgoal	Subgoal
$P \Rightarrow Q$	$P \Rightarrow \bullet M$	$\bullet M \Rightarrow Q$
$P \Rightarrow Q$	$P \Rightarrow \bullet \blacksquare_{\leq d} M$	$\bullet \blacksquare_{\leq d} M \Rightarrow Q$

Table 6.11. Anticipation-driven refinement pattern

Example. Consider the railroad crossing problem and the goal Maintain[GateClosed-WhenTrainCrossing]:

$Crossing(tr, cr) \Rightarrow cr.Gate = \text{‘closed’}$

The goal has a synchronization problem because it requires the gate to be closed simultaneously with a train entering the crossing. The tactic apply anticipation pattern can be used to resolve that problem by instantiating the first pattern in Table 6.11 as follows:

A: InRegion(tr,cr)

The two subgoals generated by the pattern are then:

$InCrossing(tr,cr) \Rightarrow \bullet \blacksquare_{\leq d'} InRegion(tr,cr)$

$\bullet \blacksquare_{\leq d'} InRegion(tr,cr) \Rightarrow cr.Gate = \text{‘opened’}$

6.9. Resolve Unbounded Achieve Goal

Consider a goal taking the form

$C \Rightarrow \Diamond T$

This goal is not realizable because it does not constrain the finite runs of an agent assigned to it. The tactic resolve unbounded achieve goal is to be applied to strengthen the goal temporally into

$C \Rightarrow \Diamond_{\leq d} T$ or $C \Rightarrow \bigcirc T$.

This tactic is generally applied at the last step of the goal-refinement process.

6.10. Summary

This chapter has proposed various specification elaboration tactics for resolving each kind of realizability problem studied in Chapter 5. These tactics provide systematic guidance for recursively refining goals into subgoals until the latter are realizable by single agents, the tactics drive the identification of agents and objects during the goal refinement process.

The systematic application of these tactics on two real case studies will be described in Chapter 9.

Chapter 7
Formal Patterns for Goal Operationalization

Once goals have been refined into subgoals that are realizable by single agents, the next step of the goal-oriented process consists in deriving (a) the operations that are relevant to the goals, and (b) the requirements on these operations so that the goals are satisfied.

This chapter proposes a formal technique to support this derivation. The general principle is to reuse generic patterns. The use of generic patterns has already been studied and applied for the upstream step of goal refinement [Dar95, Dar96]. Here, we explore the use of specific patterns for the later operationalization step.

The definition of operationalization patterns is based on a formal semantics of operationalization defined in Section 7.1. Section 7.2 describes the basic idea of using operationalization patterns; Section 7.3 discusses the benefits of such patterns for goal-oriented requirements engineering. Section 7.4 explains how patterns can be identified and organized for retrievability. Section 7.5 describes a few operationalization patterns that have been defined so far.

7.1. Semantics of the KAOS operation model

The KAOS operation model has two semantics. The first semantics is defined in terms of the transition systems introduced in Chapter 4. The second semantics is defined by translating fragments of operation specifications into temporal logic.

The two semantics are equivalent: the set of histories generated by the transition system is equal to the set of histories covered by the temporal formulae. The temporal semantics of operations is convenient for defining the semantics operationalization links that relate operations to goals.

7.1.1. Temporal semantics of operations

An operation defines a relation over states; this relation is defined by the domain pre- and post conditions of the operation. Formally, we introduce for every operation op whose arguments are variables arg_1, ..., arg_n and results are variables res_1, ..., res_n, a predicate [| op |] defined as follows:

$$[|\ op\ |](arg_1, ..., arg_n, res_1, ..., res_n) =_{def} \bullet\ DomPre(op) \wedge DomPost(op)$$

For example, consider the following operation:

Operation SwitchPumpOn
  Input PumpController {**arg** c}/ HighWaterSignal
  Output PumpController / PumpSwitch
  DomPre c.PumpSwitch = ‘Off’
  DomPost c.PumpSwitch = ‘On’

The predicate associated with this operation is:

$$[|\ \text{SwitchPumpOn}\ |]\ (c) =_{def} \bullet\ c.\text{PumpSwitch} = \text{'Off'} \wedge c.\text{PumpSwitch} = \text{'On'}$$

In the sequel, we will drop the brackets around such predicates and write SwitchPumpOn (c) instead of [| SwitchPumpOn |] (c).

The semantics of required pre-, trigger- and post- conditions is then defined as follows. Let P be a requirement on an operation op; the temporal formula associated with it is noted [|P|], and is defined as follows:

$$\text{if } P \in \text{ReqPre(op)}, \text{ then } [|\ P\ |] =_{def} (\forall *)\ [|\ op\ |] \Rightarrow \bullet\ P$$

$$\text{if } P \in \text{ReqTrig(op)}, \text{ then } [|\ P\ |] =_{def} (\forall *)\ \bullet\ P \wedge \bullet\ \text{DomPre(op)} \Rightarrow [|\ op\ |]$$

$$\text{if } P \in \text{ReqPost(op)}, \text{ then } [|\ P\ |] =_{def} (\forall *)\ [|\ op\ |] \Rightarrow P$$

By unfolding the definition of predicate [|op|], the semantics of required pre-, trigger- and post- conditions can be rephrased as follows:

$$\text{if } P \in \text{ReqPre(op)}, \text{ then } [|\ P\ |] = (\forall *)\ \bullet\ \text{DomPre(op)} \wedge \text{DomPost(op)} \Rightarrow \bullet\ P$$

$$\text{if } P \in \text{ReqTrig(op)}, \text{ then } [|\ P\ |] = (\forall *)\ \bullet\ P \wedge \bullet\ \text{DomPre(op)} \Rightarrow \text{DomPost(op)}$$

$$\text{if } P \in \text{ReqPost(op)}, \text{ then } [|\ P\ |] = (\forall *)\ \bullet\ \text{DomPre(op)} \wedge \text{DomPost(op)} \Rightarrow P$$

For example, consider the following required trigger condition on the operation introduced above:

Operation SwitchPumpOn
....
ReqTrigFor Maintain[PumpSwitchOnWhenHighWaterDetected]
c.HighWaterSignal = 'On'

The semantics of the required trigger condition is expressed by the following temporal assertion:

$$\bullet\ c.\text{HighWaterFlag} = \text{'On'} \wedge \bullet\ c.\text{PumpSwitch} = \text{'Off'} \Rightarrow \text{SwitchPumpOn}(c)$$

7.1.2. Semantics of Operationalization

Operationalization links relate realizable goals assigned to some agent to requirements on operations the agent has to perform (see Section 3.2.7.5).

A set of required pre, trigger, and post conditions operationalizes a goal if the satisfaction of the required conditions on the corresponding operations guarantees the satisfaction of the goal.

Formally, a set {P1, ..., Pn} of requirements on operations operationalizes a goal G iff the following conditions holds:

- $[|\ P1\ |], ..., [|\ Pn\ |] \models G$ **(Completeness)**
- $[|\ P1\ |], ..., [|\ Pn\ |] \not\models \text{false}$ **(Consistency)**
- $G \models [|\ P1\ |], ..., [|\ Pn\ |]$ **(Minimality)**

The completeness condition for operationalization is similar to the corresponding condition for goal refinement [Dar95, Dar96]. A first important difference however is that the semantics of goal operationalization does not rely on domain properties to guarantee the

satisfaction of the goal. This is due to the fact that the agent responsible for the goal may not rely on domain properties to realize the goal. A second important difference is in the definition of the minimality condition. The minimality condition for operationalization requires that the requirements operationalizing the goal are not stronger than required by the goal.

The formal specification of goals and operations allows the completeness, consistency and minimality of operationalization to be formally verified.

For example, consider the goal Maintain[PumpSwitchOnWhenHighWaterDetected] defined as follows:

$$\forall\ c: \text{PumpController}$$
$$c.\text{HighWaterSignal} = \text{`On'} \Rightarrow \bigcirc\ c.\text{PumpSwitch} = \text{`On'}$$

A complete, minimal, and consistent operationalization of the goal is given by the following requirements on the operation SwitchPumpOn and SwitchPumpOff:

Operation SwitchPumpOn
Input PumpController {**arg** c}/ HighWaterSignal
Output PumpController / PumpSwitch
DomPre c.PumpSwitch = 'Off'
DomPost c.PumpSwitch = 'On'
ReqTrigFor Maintain[PumpSwitchOnWhenHighWaterDetected]
c.HighWaterSignal = 'On'

Operation SwitchPumpOff
Input PumpController {**arg** c}/ HighWaterSignal
Output PumpController / PumpSwitch
DomPre c.PumpSwitch = 'On'
DomPost c.PumpSwitch = 'Off'
ReqPreFor Maintain[PumpSwitchOnWhenHighWaterDetected]
$\neg$ c.HighWaterSignal = 'On'

The required trigger condition on the operation SwitchPumpOn requires that the pump *must be* switched on when the HighWaterSignal is On; the required precondition on the operation SwitchPumpOff requires that the pump *may be* switched off only if the HighWaterSignal is not On.

The completeness of this operationalization is established by proving the following assertion:

$$\begin{aligned}
&\text{SwitchPumpOn}(c) \Leftrightarrow \bullet\ c.\text{PumpSwitch} = \text{`Off'} \wedge c.\text{PumpSwitch} = \text{`On'}\ ,\\
&\text{SwitchPumpOff}(c) \Leftrightarrow \bullet\ c.\text{PumpSwitch} = \text{`On'} \wedge c.\text{PumpSwitch} = \text{`Off'}\ ,\\
&\bullet\ c.\text{HighWaterSignal} = \text{`On'} \wedge \bullet\ c.\text{PumpSwitch} = \text{`Off'} \Rightarrow \text{SwitchPumpOn}(c)\ ,\\
&\text{SwitchPumpOff}(c) \Rightarrow \bullet\ \neg\ c.\text{HighWaterSignal} = \text{`On'}\\
\models\\
&c.\text{HighWaterSignal} = \text{`On'} \Rightarrow \bigcirc\ c.\text{PumpSwitch} = \text{`On'}
\end{aligned}$$

Note that the required precondition on the SwitchPumpOff operation is necessary for the operationalization to be complete.

The minimality condition is obtained by inverting the antecedent and the consequent in the above formula, and can also be established formally.

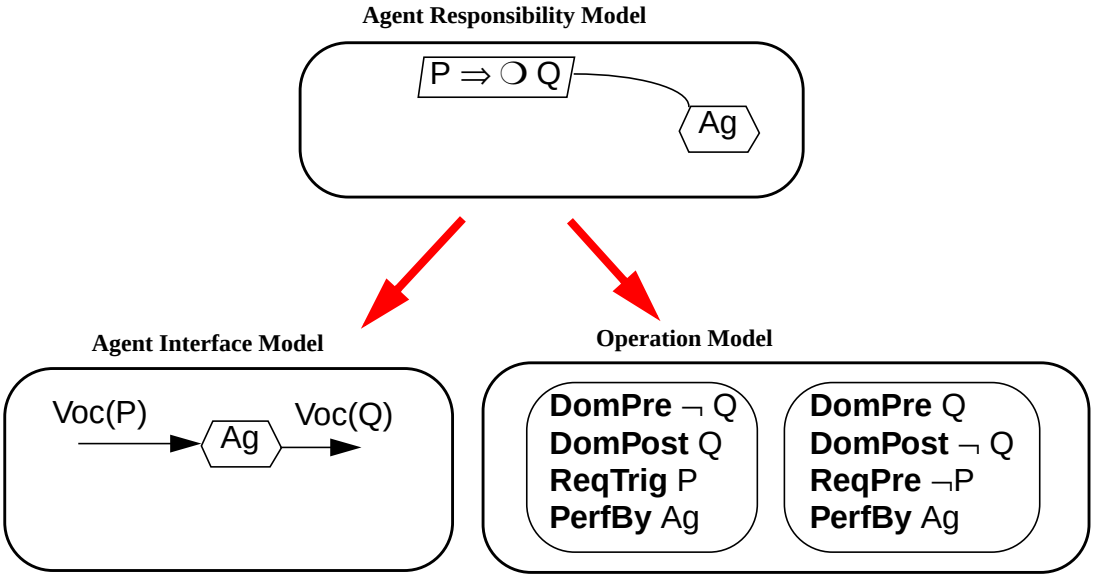


FIGURE 7.1. Operationalization pattern for P ⇒ ❍ Q

7.2. Operationalization Patterns

Operationalization patterns capture operationalization links between an abstract goal specification and abstract operation specifications. They are proved correct with respect to the above semantics of operations. They can be used to derive required pre-, trigger-, and post-conditions on operations from the formal definition of a terminal goal.

For each operationalization pattern, we also give the variables that need to be monitored and controlled by an agent for the goal to be realizable by that agent.

As a first example, the operationalization pattern in Figure 7.1 provides a way to operationalize Achieve goals of the form P ⇒ ❍ Q, where Q is a state formula and P a past formula.

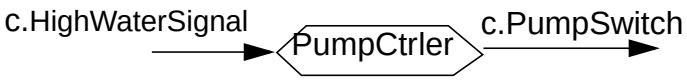
In the agent interface model of that pattern, Voc(P) and Voc(Q) denote the set of state variables appearing in the assertions P and Q, respectively.

The operation model in that pattern states that in order to operationalize a goal P ⇒ ❍ Q, one has to define two operations: a first operation with domain pre/post conditions given by the pair [¬ Q, Q] and a *required trigger condition* stating that the operation *must* be applied when P holds; and a second operation with domain pre/post conditions given by the pair [Q, ¬ Q] and a *required precondition* stating that the operation *may* be applied only when P does not hold.

This pattern is generic. It can be instantiated to completely different situations. For example, the goal Maintain[PumpSwitchOnWhenHighWaterDetected] can be operationalized by instantiating the operationalization pattern in Figure 7.1 as follows:

P : c.HighWaterSignal Q : c.PumpSwitch = ‘On’

The specifications for the operations SwitchPumpOn and SwitchPumpOff in Section 7.1 are thereby automatically derived, together with the following agent interface model:



Our operationalization patterns generate a minimal and complete set of operations and requirements on these operations to ensure the goal considered. In the example above, if the operation SwitchPumpOff had been left out of the specification, the operation model would not have been complete with respect to the goal.

Use of operationalization patterns shortcuts tedious proofs of consistency, completeness, and minimality. All patterns presented in this chapter were proved formally correct. Patterns involving propositional qualitative temporal logic were proved correct using the SteP verification tool [Man96]; whereas the others were proved “by-hand”.

A few operationalization patterns are proposed in [Dar93]. The patterns defined there are not based on a formal semantics of operations, and do not ensure the completeness, consistency and minimality of the operationalization links as defined in this thesis.

7.3. Benefits of Operationalization Patterns

Operationalization patterns are useful for the following reasons.

- They allow low-level formal reasoning to be hidden from requirements engineers.
- They provide constructive guidance for deriving operational requirements from goals.
- They enable one to detect incomplete operational requirements and provide guidance on how to make the specification complete with respect to stated goals [Yue87].
- Operationalization patterns applied backwards allow formal goal specifications to be inferred bottom-up from operational requirements.

Each point is discussed in turn.

7.3.1. Hiding low-level proofs

Formal languages allow properties of interest to be proved formally by use of inference rules of the language. Proving properties using such inference rules is generally a tedious, complex and error-prone activity that requires high levels of expertise. Patterns are intended to relieve specifiers of such low-level formal reasoning. Patterns amount to *high-level inference rules* that are proved correct once and for all. They can be reused many times in many different contexts to solve commonly occurring problems.

An alternative technique to formal operationalization patterns would be a fully automatic technique for deriving operational requirements from goals. Fully automatic techniques, such as model checking, can be used to *verify* formal requirements models a posteriori; however they do not provide automatic guidance to constructively elaborate the model. Furthermore, such techniques generally require restricting the expressive power of the language.

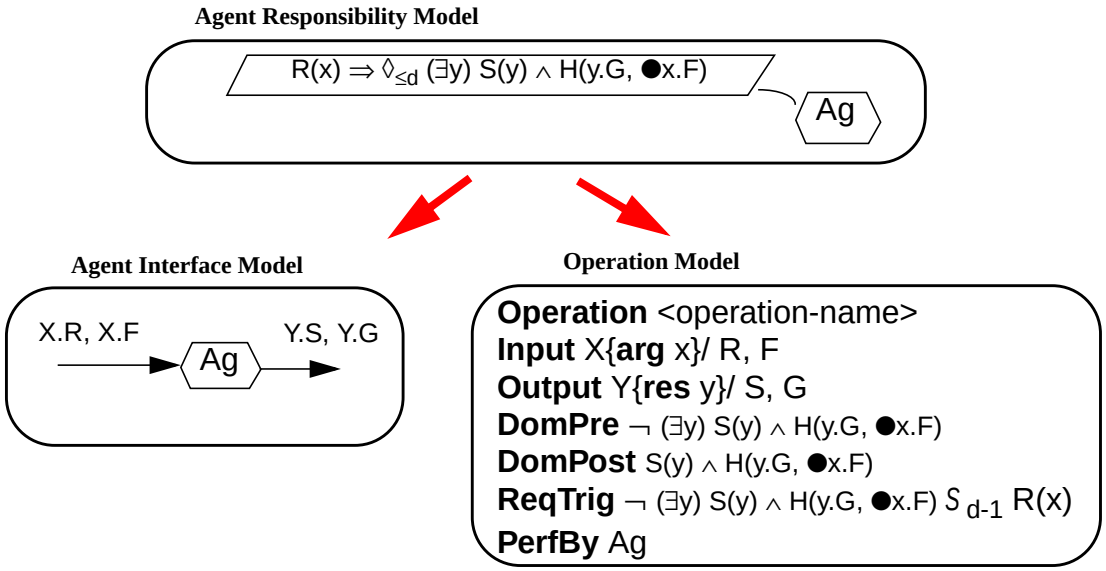


FIGURE 7.2. Operationalization pattern for R(x) ⇒ ◊≤d (∃y) S(y) ∧ H(y.G, ●x.F)

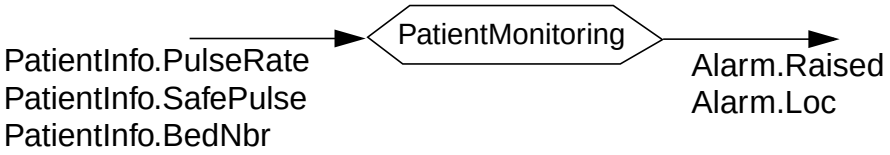
7.3.2. Deriving operational requirements from goals

As mentioned before, operationalization patterns can be used constructively to (i) identify the operations relevant to the goals, and (ii) derive requirements on these operations ensuring that the goals are satisfied.

A first example of this has been shown in Section 7.2. As another example, consider the patient monitoring problem and the goal Achieve[AlarmForCrticalPulseRateInfo] assigned as the responsibility of the PatientMonitoring software agent. This goal is specified as follows:

pi.PulseRate ∉ pi.SafePulse ⇒ ◊≤delay (∃ a: Alarm): a.Raised ∧ a.Loc = ● pi.BedNbr

This goal is an Achieve goal involving real-time delays. An operationalization pattern matching this goal definition is shown in Figure 7.2. (Note that this operationalization pattern is an example of a first-order pattern involving real-time delays). The pattern yields the following interface declaration for the PatientMonitoring agent.:



The operational requirement resulting from the application of this pattern is:

Operation RaiseAlarm
Input PatientInfo {**arg** pi}/ PulseRate, SafePulse, BedNbr
Output Alarm {**res** a}/ Raised, Loc
DomPre ¬ (∃ a: Alarm): a.Raised ∧ a.Loc = ● pi.BedNbr
DomPost a.Raised ∧ a.Loc = ● pi.BedNbr
ReqTrig For AlarmForCrticalPulseRateInfo
¬ (∃ a: Alarm): a.Raised ∧ a.Loc = ● pi.BedNbr
S delay -1 pi.PulseRate ∉ pi.SafePulse
PerfBy PatientMonitoring

7.3.3. Checking operational requirements for completeness

Operationalization patterns can be used to check whether some given operation specifications are complete operationalizations of given goals. This use of operationalization patterns corresponds to the similar use of refinement patterns in [Dar96]; it is important as intuitive operationalizations of goals produced by hand tend to be incomplete.

For example, consider the goal Maintain[PumpSwitchOnWhenHighWaterDetected] again. An intuitive operationalization of that goal might be given by the operation SwitchPumpOn together with the required trigger condition:

ReqTrig For PumpSwitchOnWhenHighWaterDetected
c.HighWaterSignal

The goal definition and the operation match the operationalization pattern in Figure 7.1. One can therefore derive that, in order to be complete, the operationalization should also include the operation SwitchPumpOff with the following required precondition:

ReqPre For PumpSwitchOnWhenHighWaterDetected
¬ c.HighWater

7.3.4. Inferring goals from operations

Operationalization patterns can also be used backwards to elicit goals underlying some operation specifications. This use of patterns is important as initial descriptions of requirements tend to be given in very operational terms. Eliciting goals underlying such operational specifications allows for various goal-level analysis such as checking that the operations are complete with respect to the goals [Yue87], identifying and resolving conflicts at the goal level [Lam98b], identifying and resolving obstacles to goals [Lam98a, Lam2Ka], and exploring alternative system proposals.

Consider for instance the specification of a simple autopilot [But96]. The initial problem statement defines the following informal requirements.

> *"If the pilot dials in an altitude that is more than 1,200 feet above the current altitude and then presses the alt_eng button, the altitude mode will not directly engage. Instead, the altitude engage mode will change to "armed" and the flight-path angle select mode is engaged."*

The informal requirements refer to the operations EngageALTmode, ArmALTmode, and EngageFPAmode together with required conditions on their applications. For instance, the operation EngageFPAmode can be specified as follows:

Operation EngageFPAmode
Input AutoPilot {**arg** a}
ALTengageEvent {**arg** alt_eng}
Output Autopilot / FPAmode
DomPre a.FPAmode = 'off'
DomPost a.FPAmode = 'on'
ReqTrig For <*unknown goal*>
alt_eng. occurs ∧ ALTtarget - ALTactual > 1200
"if the pilot presses the alt_eng button when the target altitude is more than 1,200 feet above the current altitude, the FPA mode is engaged"

One can then apply the operationalization pattern in Figure 7 below matching the domain pre- and post conditions and the required trigger condition to infer the goal that justifies this trigger condition:

Goal Achieve[FPAModeEngagedWhenHighTargetAltitutde]
FormalDef ∀ a: AutoPilot, alt_eng: ALTengageEvent
@ alt_eng. occurs ∧ ALTtarget - ALTactual > 1200 ∧ a.FPAMode = 'off'
⇒ ❍ a.FPAmode = 'on'

Higher-level goals can then be identified by asking WHY questions. The resulting goal graph will provide the rationale for the operations described in the initial problem statement. Goal-level analysis can then be performed on the derived goal structure.

A formal technique for inferring goals from scenarios is described in [Lam98b]. The starting point of the goal inference procedure discussed there and the use of patterns presented in this section are different. [Lam98b] starts from concrete scenarios of interaction between agents, represented as instance-level trace diagrams. The inference of goals with operationalization patterns starts from operational specifications.

7.4. Building a Library of Patterns

We now describe how relevant operationalization patterns can be identified and organized for retrievability. Currently, only a few representative patterns have been identified. Further extension of the library is subject to further work.

7.4.1. Identifying Patterns

One way to identify operationalization patterns is to abstract them from concrete examples of goal operationalizations. Unfortunately, there is no large set of specifications available from which patterns could be inferred. This is due to the fact that the constructive elaboration of operational requirements from goals is not widely adopted yet; and previous derivations of operational requirements from goals were done by hand without a fully precise semantics for operationalization links.

We decided to explore the space of operationalizations based on the pattern of the goal to be operationalized. For each pattern of goal specification, we derived the corresponding operationalization pattern. In order to get a rich set of goal patterns, we extended and specialized the high-level Achieve/Maintain patterns of the KAOS language with patterns adapted from [Dwy99].

Currently, our library is composed of propositional patterns only. Further work is required to extend it with first-order patterns.

Figure 7.3 shows the taxonomy of goal patterns that have been considered so far. Each name in the hierarchy may have several variants of goal patterns. The figure shows a typical propositional goal pattern for each node. Further extensions of this taxonomy of goal patterns will trigger the identification of further operationalization patterns in our library.

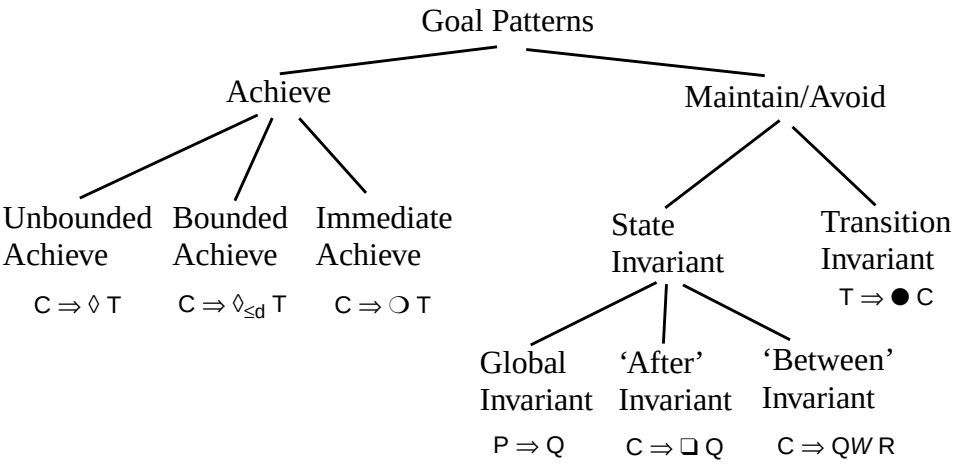


FIGURE 7.3. A taxonomy of goal patterns

7.4.2. Coverage of the Library

Since operationalization patterns are identified from the pattern of the goal, *the coverage of the library of operationalization patterns is relative to the coverage of the taxonomy of goal patterns*. The effectiveness of our approach is based on the assumption that most properties that occur in practice can be specified using a small set of goal patterns. This assumption is partly supported by an empirical study reported in [Dwy99].

7.5. A Library of Operationalization Patterns

7.5.1. Achieve Goals

We first describe operationalization patterns for Achieve goals. Propositional patterns for specifications of Achieve goals include:

Unbounded Achieve: $C \Rightarrow \Diamond T$

Bounded Achieve: $C \Rightarrow \Diamond_{\leq d} T$

Immediate Achieve: $C \Rightarrow \bigcirc T$, $\bullet C \Rightarrow T$, $C \wedge \neg T \Rightarrow \bigcirc T$

Since Unbounded Achieve goals are not realizable, there is no operationalization pattern for such goals. Operationalization patterns for Bounded Achieve goals and Immediate Achieve goals are given in Figures 7.4 to 7.7. As mentioned before, further work is required to define operationalization patterns for first-order variants of these goal patterns. (An example of a first-order pattern for Bounded Achieve goals was given in Figure 7.2.)

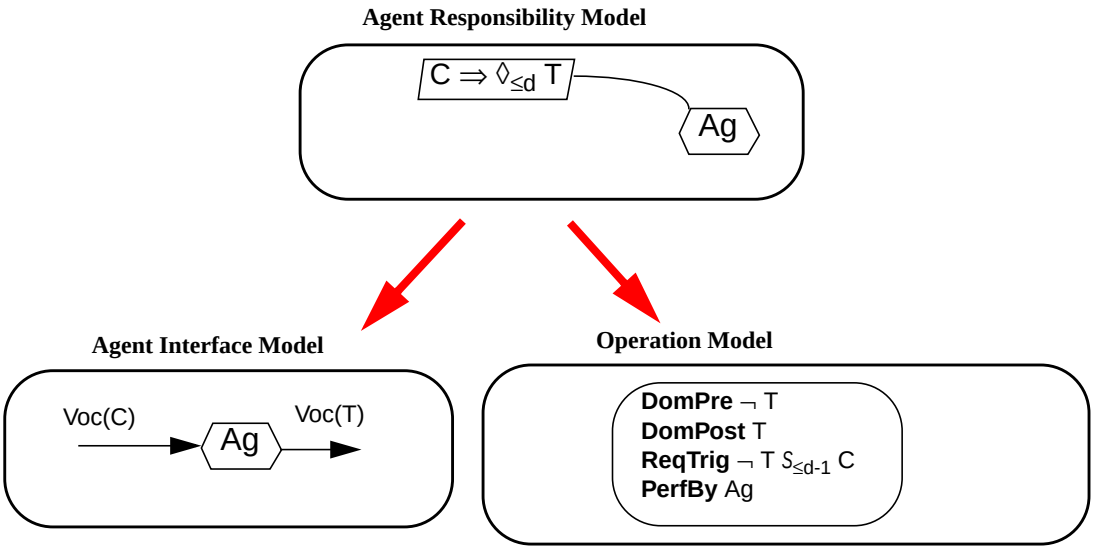


FIGURE 7.4. Operationalization pattern for $C \Rightarrow \Diamond_{\leq d} T$

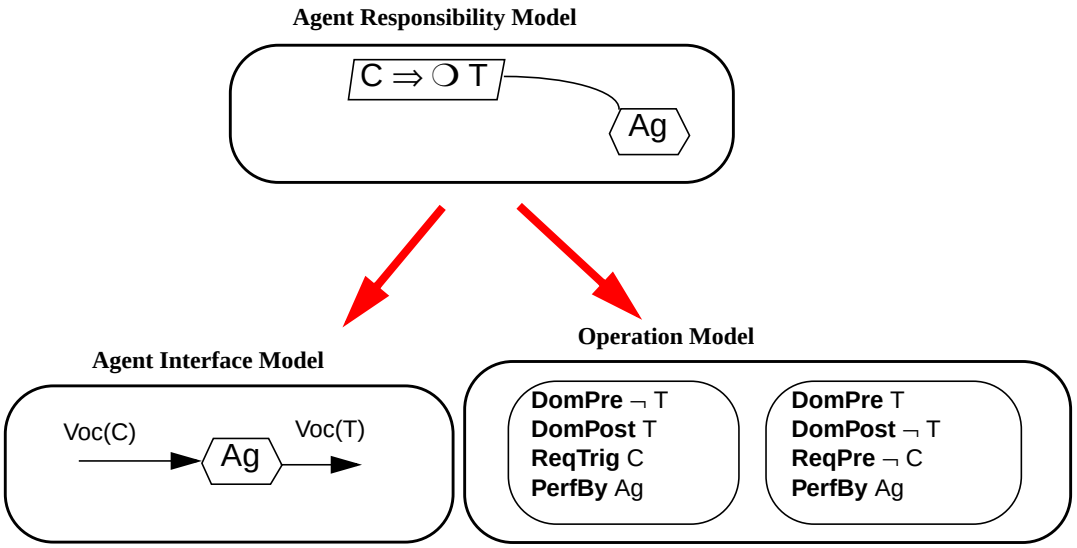


FIGURE 7.5. Operationalization pattern for P ⇒ ❍ Q

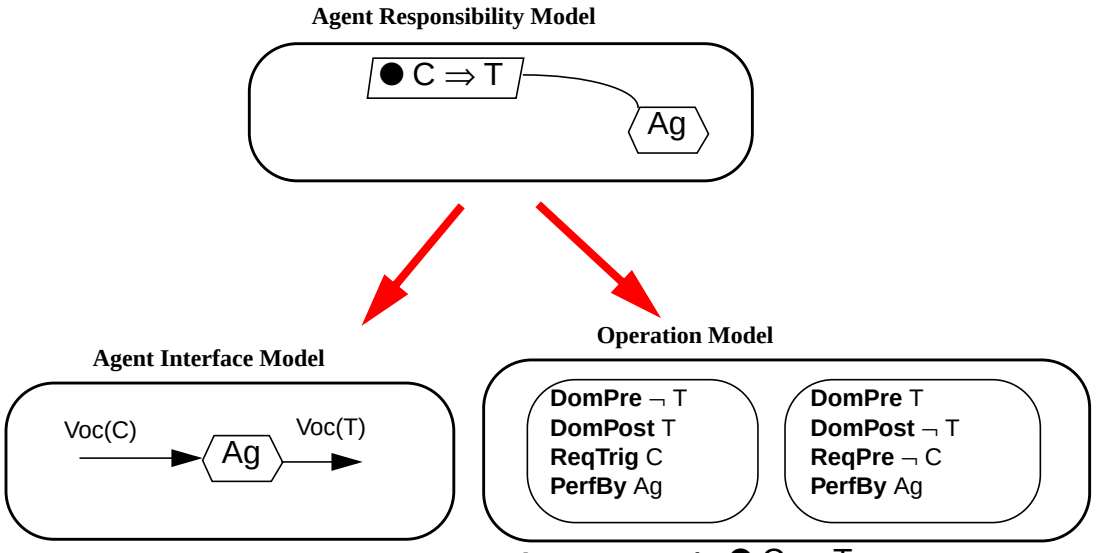


FIGURE 7.6. Operationalization pattern for ● C ⇒ T

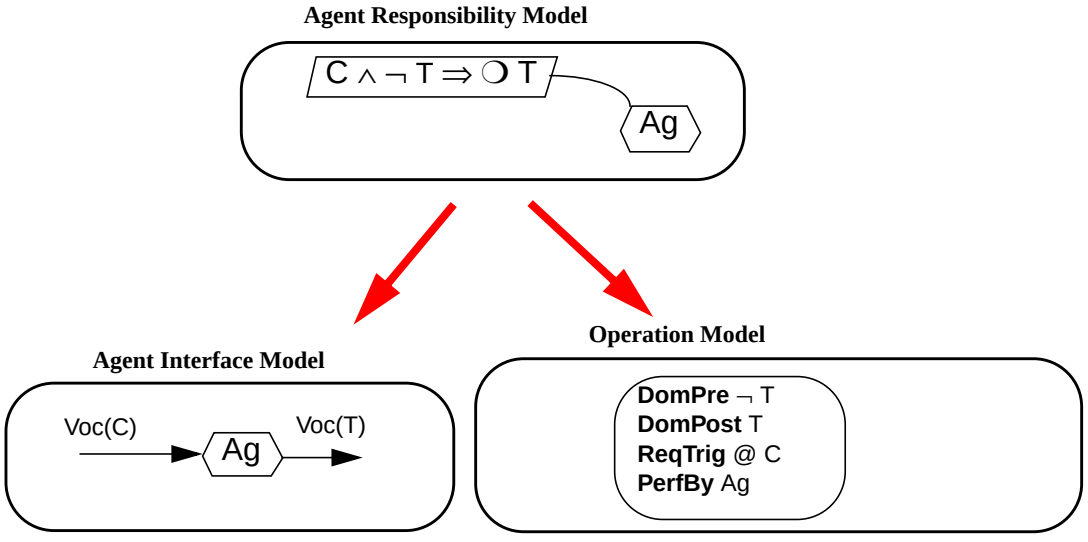


FIGURE 7.7. Operationalization pattern for C ∧ ¬ T ⇒ ❍ T

7.5.2. Maintain Goals

For Maintain goals, we distinguish between state invariants that constrain system states, and transition invariants that constrain system transitions. In the spirit of [Dwy99], we also consider different temporal scopes of the invariant: a *global* invariant is required to hold over all system states, an '*after*' invariant is required to hold only after some condition C, and a '*between*' invariant is required to hold between states in which C holds and states in which R holds.

For *state invariants*, we consider the following goal patterns:

Global Invariant: $P \Rightarrow Q$

After Invariant: $C \Rightarrow \boxminus Q$, $C \Rightarrow \boxminus_{\leq d} Q$

Between Invariant: $C \Rightarrow Q \; W \; R$, $\bullet C \Rightarrow Q \; W (Q \wedge R)$

For a *global invariant* of the form $P \Rightarrow Q$ to be realizable by an agent, the agent has to control all variables appearing in P and Q; the operationalization of such goal is shown in Figure 7.8. Note that state invariants are operationalized by *required post-conditions*.

Similarly, for an *'after' invariant* of the form $C \Rightarrow \boxminus Q$ or $C \Rightarrow \boxminus_{\leq d} Q$ to be realizable by an agent, the agent has to control all variables appearing in C and Q; operationalization patterns for such goals are given in Figures 7.9 and 7.10. These goals are operationalized by (i) a required post-condition on an operation that makes C true, and (ii) a required pre-condition on the operation that makes Q false.

For a *'between' invariant* of the form $C \Rightarrow Q \; W \; R$ to be realizable by an agent, that agent also has to control all variables appearing in P, Q and R; the operationalization of such a goal is given in Figure 7.11.

A 'between' invariant of the form $\bullet C \Rightarrow Q \; W (Q \wedge R)$ is realizable by an agent controlling Q and monitoring C and R. The operationalization of such goal is given in Figure 7.12. The operationalization consists of (i) a required trigger condition on the operation that makes Q true, and (ii) a required precondition on the operation that makes Q false.

For *transition invariants*, we consider the following patterns:

required transition: $C \Rightarrow \bigcirc T$, $\bullet C \Rightarrow T$, $C \wedge \neg T \Rightarrow \bigcirc T$

allowed transition: $@ T \Rightarrow \bullet C$, $T \Rightarrow \bullet C$

These patterns are realizable by an agent that controls the variables appearing in T and monitors the variables appearing in C. The required transition patterns are the same as the immediate Achieve patterns. Operationalization patterns for the allowed transition goal patterns are given in Figure 7.13 and 7.14. The definition of first-order variants of these patterns is subject to further work.

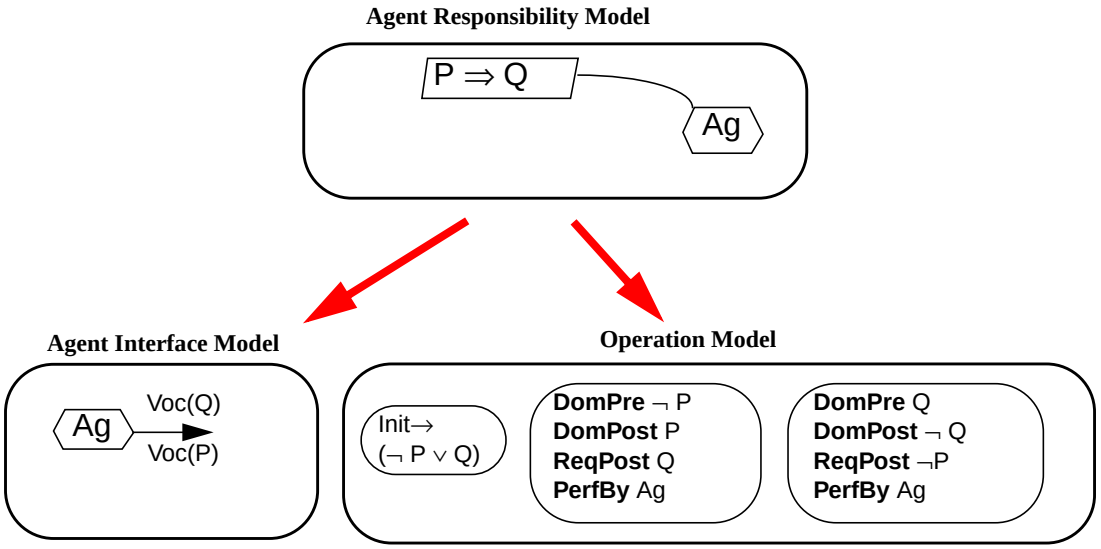


FIGURE 7.8. Operationalization pattern for $P \Rightarrow Q$

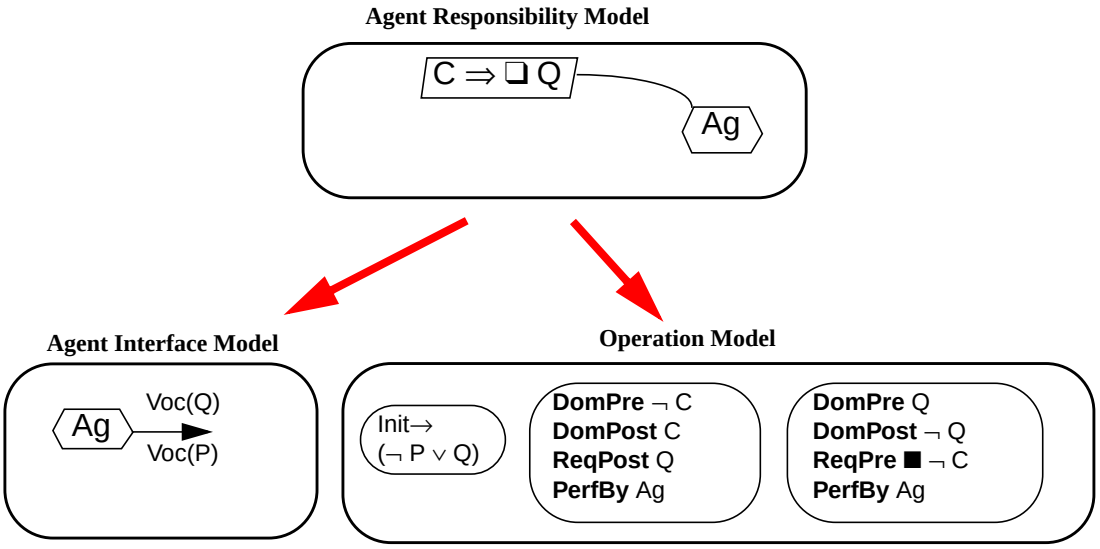


FIGURE 7.9. Operationalization pattern for C ⇒ ❑ Q

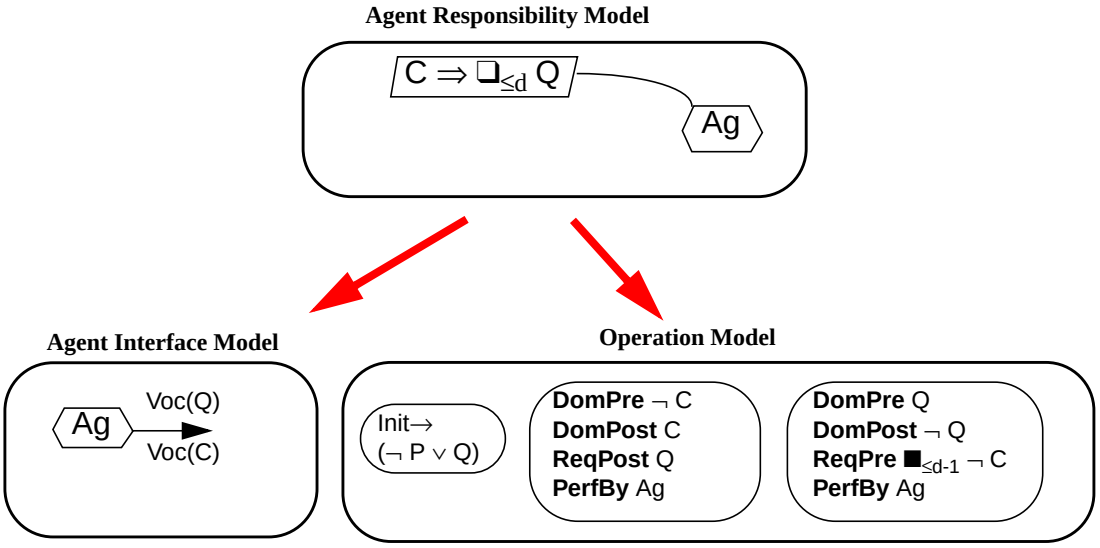


FIGURE 7.10. Operationalization pattern for C ⇒ ❑≤d Q

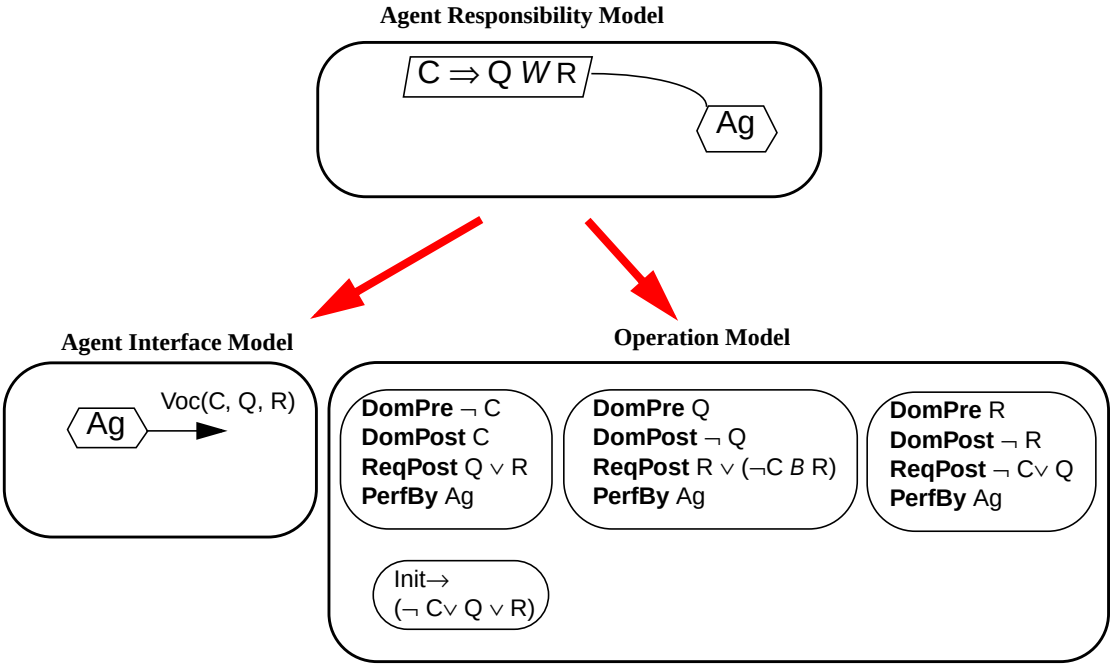


FIGURE 7.11. Operationalization pattern for C ⇒ Q *W* R

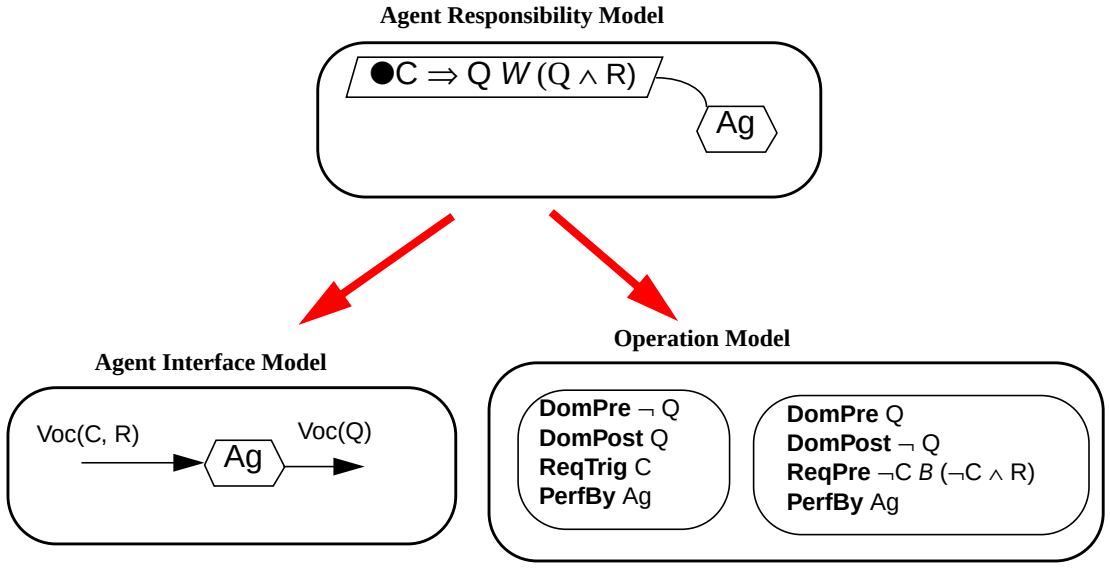


FIGURE 7.12. Operationalization pattern for ● P ⇒ Q *W* (Q ∧ R)

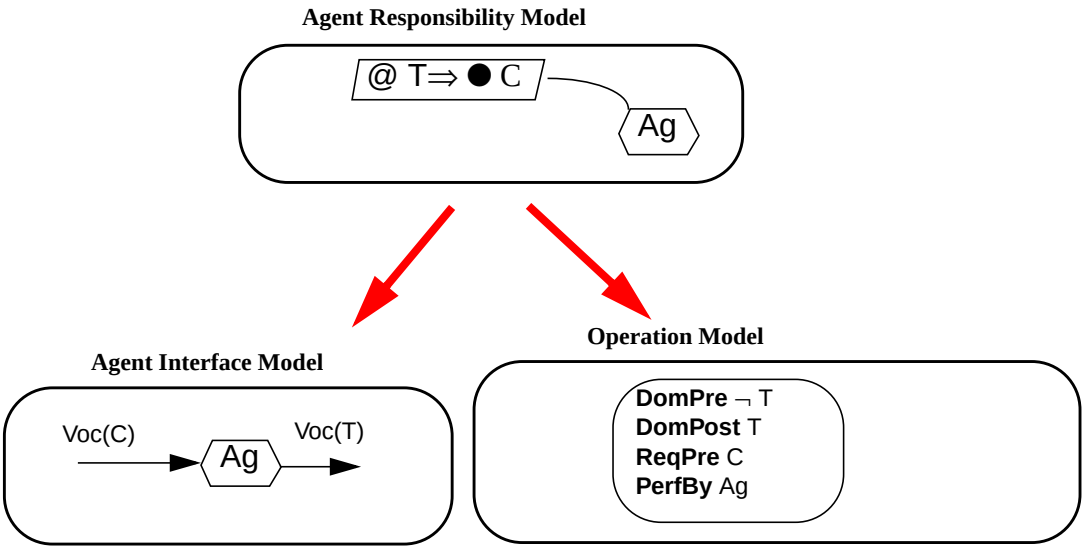


FIGURE 7.13. Operationalization pattern for @ T⇒ ● C

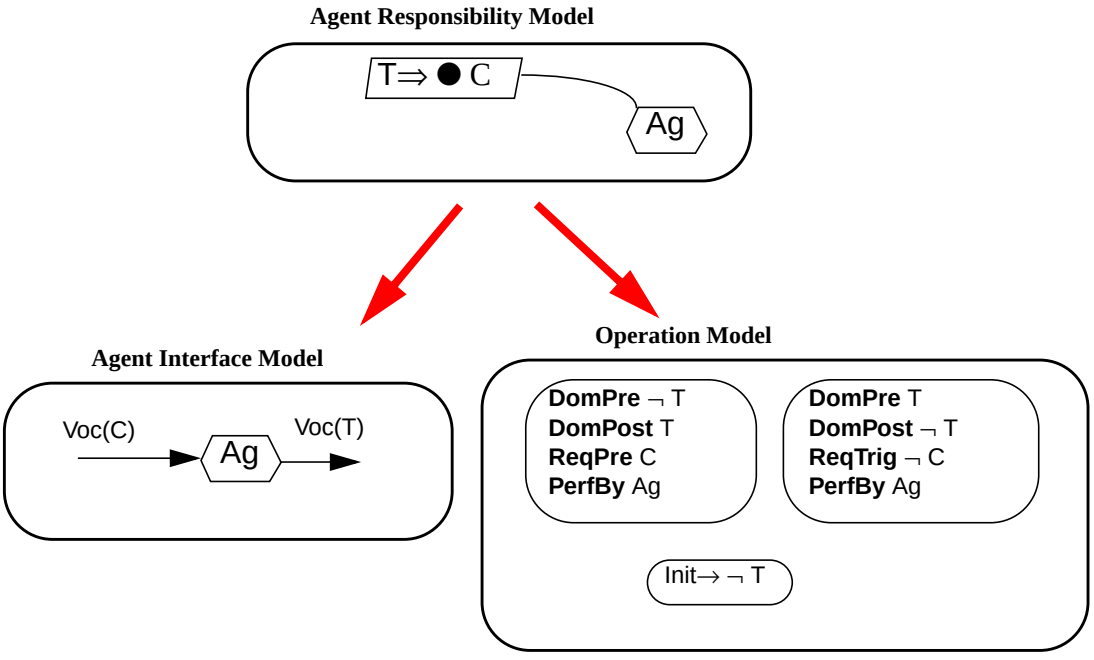


FIGURE 7.14. Operationalization pattern for T⇒ ● C

Chapter 8
Obstacle Analysis

This chapter describes the anticipation and handling of exceptional agent behaviours during the requirements elaboration process. It is largely based on [Lam2Ka].

8. 1. Introduction

One major problem requirements engineers are faced with is that first-sketch specifications of goals, requirements and assumptions tend to be too ideal; such assertions are likely to be occasionally violated in the running system due to unexpected behavior of agents like humans, devices, or software components [Lam95, Pot95, Fea98]. This general problem is not really handled by current requirements elaboration methods.

Consider the mine pump system, for example; a first-sketch goal such as Maintain[PumpOnWhenHighWater] is overideal and likely to be violated from time to time -- because, e.g., the water sensor may fail to correctly detect a high water level; the pump may refuse to start; the pump controller may fail to produce correct input in time; etc. In an ambulance dispatching system, a first-sketch goal such as Achieve[MobilizedAmbulancePromptlyAtIncident] is overideal and likely to be violated because of, e.g., allocation of a vehicle not close enough to the incident location; or too long allocation time; or imprecise or confused location; etc. In an electronic reviewing system for a scientific journal, a first-sketch goal such as Achieve[ReviewReturnedInFourWeeks] or an assumption such as ReviewerReliable are straightforward examples of overideal statements that are likely to be violated on occasion; the same might be true for a security goal such as Maintain[ReviewerAnonymity]. In a resource management system, a goal such as Achieve[RequestedResourceUsed] or an assumption such as RequestPendingUntilUse are also overideal as requesting agents may change their mind and no longer wish to use the requested resource even if the latter becomes available. In a meeting scheduler system, a goal such as Achieve[ParticipantsTimeConstraintsProvided] is likely to be violated, e.g., for participants that do not check their email regularly thereby missing invitations to meetings and requests for providing their time constraints. In a control system, a goal such as Maintain[AlarmIssuedWhenAbnormalCondition] might be violated sometimes due to unavailable data, device failure or deactivation by malicious agents.

Overidealization of goals, requirements and assumptions results in run-time inconsistencies between the specification of the system and its actual behavior. The lack of anticipation of exceptional circumstances may thus lead to unrealistic, unachievable and/or incomplete requirements. As a consequence, the software developed from those requirements will inevitably result in failures, sometimes with critical consequences for the environment.

The purpose of this chapter is to introduce systematic techniques for deidealizing goals, assumptions and requirements, and to integrate such techniques in the goal-oriented requirements elaboration method in order to derive more complete and realistic requirements, from which more robust and flexible systems can be built.

Our approach is based on the concept of *obstacle* first introduced in [Pot95]. Obstacles are a dual notion to goals; while goals capture desired conditions, obstacles capture undesirable (but nevertheless possible) ones. An obstacle obstructs some goal, that is, when the obstacle becomes true the goal may not be achieved. The term “obstacle” is thus introduced here to denote a *goal-oriented* abstraction, at the requirements engineering level, of various notions that have been studied extensively in specific areas - such as *hazards* that may obstruct safety goals [Lev95] or *threats* that may obstruct security goals [Amo94] -, or in later phases of the software lifecycle - such as *faults* that may prevent a program from achieving its specification [Cri95, Gar99].

The chapter presents a formalization of this notion of obstacle; a set of techniques for systematic generation of obstacles from goal specifications and domain properties; and a set of alternative specification elaboration tactics that transform goal specifications so as to resolve the obstacles generated.

Back to the example of the ideal goal named Achieve[ReviewReturnedInFourWeeks], our aim is to derive obstacle specifications from a precise specification of this goal and from properties of the domain; one would thereby expect to obtain obstacles such as, e.g., WrongBeliefAboutDeadline or ReviewRequestLost; UnprocessablePostscriptFile; and so on. From there one would like to resolve those obstacles, e.g., by weakening the original goal formulation and propagating the weakened version in the goal refinement graph; by introducing new goals and operationalizations to overcome or mitigate the obstacles; by changing agent assignments so that the obstacle is less likely occur; and so on.

The rest of the chapter is organized as follows. Section 8.2 introduces obstacles to goals and provides a formal characterization of this concept, including the notion of completeness of a set of obstacles. Section 8.3 discusses a modified goal-oriented requirements elaboration process that integrates obstacle analysis. Section 8.4 presents techniques for generating obstacles from goal formulations. Section 8.5 then presents techniques for transforming goals, requirements and/or assumptions so as to resolve the obstacles generated.

8. 2. Goal Obstruction by Obstacles

This section formally defines obstacles, their relationship to goals, and their refinement links; a criterion is provided for a set of obstacles to be complete; a general taxonomy of obstacles is then suggested. In the sequel, the general term “goal” will be used indifferently for a high-level goal, a requirement assigned to an agent in the software-to-be, or an assumption assigned to an agent in the environment.

8. 2. 1. Obstacles to goals

Semantically speaking, a goal defines a set of desired behaviors, where a behavior is a temporal sequence of states.Goal refinement yields sufficient subgoals for the goal to be achieved.

Likewise, an obstacle defines a set of undesirable behaviors. Goal obstruction yields sufficient obstacles for the goal to be violated; the negation of such obstacles yields necessary preconditions for the goal to be achieved.

Let *G* be a goal and *Dom* a set of domain properties. An assertion *O* is said to be an *obstacle* to *G* in *Dom* iff the following conditions hold:

1. {O, Dom} $\models \neg$ G (*obstruction*)
2. {O, Dom} $\not\models$ **false** (*domain-consistency*)

Condition (1) states that the negation of the goal is a logical consequence of the theory comprising the obstacle specification and the set of domain properties available; condition (2) states that the obstacle may not be logically inconsistent with the domain theory. Clearly, it makes no sense to reason about obstacles that are inconsistent with the domain.

As a first simple example, consider a library system and the following high-level goal stating that every book request should eventually be satisfied:

Goal *Achieve* [BookRequestSatisfied]
FormalDef $\forall$ bor: Borrower, b: Book
Requesting (bor, b)
$\Rightarrow \Diamond$ ($\exists$ bc: BookCopy) [Copy (bc, b) $\wedge$ Gets (bor, bc)]

An obstructing obstacle to that goal might be specified by the following assertion:

$\exists$ bor: Borrower, b: Book
$\Diamond$ { Requesting (bor, b)
$\wedge \Box \neg$ ($\exists$ bc: BookCpy) [Copy (bc, b) $\wedge$ Gets (bor, bc)] }

Condition (1) trivially holds as the assertion amounts to the negation of the goal (remember that $P \Rightarrow Q$ iff $\Box (P \rightarrow Q)$, and $\neg \Box (P \rightarrow Q)$, iff $\Diamond (P \wedge \neg Q)$,). This obstructing assertion covers the classical starvation scenario [Dij71] in which, each time a copy of a requested book becomes available, this copy gets borrowed in the next state by a borrower different from the requesting agent.

To further illustrate the need for condition (2), consider the following goal for some device control system (expressed in propositional terms for simplicity):

Running $\wedge$ PressureTooLow $\Rightarrow$ AlarmRaised

Considering the domain property:

PressureTooLow $\wedge$ Startup $\Rightarrow \neg$ AlarmRaised,

it is easy to see that condition (1) would be satisfied by the candidate obstacle

$\wedge \Diamond$ [Running $\wedge$ PressureTooLow $\wedge$ Startup]

which taken with the above domain property logically entails the negation of the goal; however this candidate is inconsistent with another domain property stating that the device cannot be both in startup and running modes:

Running $\Rightarrow \neg$ Startup

Note that the above definition of an obstructing obstacle allows for the same obstacle to obstruct several different goals; examples of this will be seen later on.

It is also worth noticing that, since *Achieve/Cease* and *Maintain/Avoid* goals all have the general form $\Box$ GC, an obstacle to such goals will always have the general form $\Diamond$ OC; in the sequel, GC and OC will be called goal and obstacle condition, respectively.

8. 2. 2. Completeness of a set of obstacles

Given some goal formulation, defensive requirements specification would require as many meaningful obstacles as possible to be identified for that goal; completeness is desirable -at least for high-priority goals such as, e.g., Safety goals.

A set of obstacles O_1, ..., O_n to goal G in Dom is *domain-complete* with respect to G iff the following condition holds:

$\{\neg O_1, ..., \neg O_n, Dom\} \models G$ (*domain-completeness*)

This condition intuitively means that if none of the obstacles in the set may occur then the goal is satisfied.

It is most important to note that completeness is a notion relative to what is known about the domain. To make this clear, let us consider the following example introduced in [Jac95] after a real plane incident. The goal

$MovingOnRunway \Rightarrow ReverseThrustEnabled$

can be AND-refined, using the milestone refinement pattern [Dar96], into two subgoals:

$MovingOnRunway \Rightarrow WheelsTurning$ (Ass)

$WheelsTurning \Rightarrow ReverseThrustEnabled$ (Rq)

The second subgoal is a requirement assigned to a software agent; the first subgoal is an assumption assigned to an environment agent. Assumption *Ass* will be violated iff

$\Diamond (MovingOnRunway \wedge \neg WheelsTurning)$ (N-Ass)

Assume now that the following necessary conditions for wheels to be turning are known in the domain:

$WheelsTurning \Rightarrow WheelsOut$ (D1)
$WheelsTurning \Rightarrow \neg WheelsBlocked$ (D2)
$WheelsTurning \Rightarrow \neg Aquaplaning$ (D3)

The following obstacles can then be seen to obstruct *Ass* in that domain since each of them then entails *N-Ass*:

$\Diamond (MovingOnRunway \wedge \neg WheelsOut)$ (O1)
$\Diamond (MovingOnRunway \wedge WheelsBlocked)$ (O2)
$\Diamond (MovingOnRunway \wedge Aquaplaning)$ (O3)

In order to check the domain completeness of these obstacles we take their negation:

$MovingOnRunway \Rightarrow WheelsOut$ (N-O1)
$MovingOnRunway \Rightarrow \neg WheelsBlocked$ (N-O2)
$MovingOnRunway \Rightarrow \neg Aquaplaning$ (N-O3)

Back to the definition of domain-completeness, one can see that the set of obstacles {O1, O2, O3} will be complete or not depending on whether or not the following property is known in the domain:

$MovingOnRunway$
$\wedge WheelsOut \wedge \neg WheelsBlocked \wedge \neg Aquaplaning$ (D4)
$\Rightarrow WheelsTurning$

Obstacle completeness thus really depends on what valid properties are known in the domain.

Note that if D4 is not a valid property in the domain, the negation of this property, i.e,

◊ (MovingOnRunway
∧ WheelsOut ∧ ¬ WheelsBlocked ∧ ¬ Aquaplaning (O4)
∧ ¬ WheelsTurning)

is a further obstacle to the above assumption Ass. Furthermore, the set {O1, O2, O3, O4} is now complete wrt. Ass. In this case, no domain property is used to show the completeness of the obstacles.

8. 2. 3. Obstacle refinement

Like goals, obstacles may be refined. *AND-refinement* links may relate an obstacle to a set of subobstacles (called refinement); this means that satisfying the subobstacles in combination is a sufficient condition in the domain for satisfying the obstacle. *OR-refinement* links may relate an obstacle to an alternative set of refinements; this means that satisfying one of the refinements is a sufficient condition in the domain for satisfying the obstacle. The obstacle refinement structure for a given goal may thus be represented by an AND/OR directed acyclic graph.

A set of obstacles O_1, ..., O_n is an *AND-refinement* of an obstacle O iff the following conditions hold:

1. $\{O_1 \wedge O_2 \wedge ... \wedge O_n, Dom\} \models O$ (*entailment*)
2. $\{O_1 \wedge O_2 \wedge ... \wedge O_n, Dom\} \not\models$ **false** *(consistency)*

In general one is interested in minimal AND-refinements, in which case the following condition has to be added:

2. for all i: $\{\wedge_{j \neq i} O_j, Dom\} \not\models O$ (*minimality*)

A set of obstacles O_1, ..., O_n is an *OR-refinement* of an obstacle O iff the following conditions hold:

1. for all i: $\{O_i, Dom\} \models O$ (*entailment*)
2. for all i: $\{O_i, Dom\} \not\models$ **false** *(consistency)*

In general one is interested in complete OR-refinements in which case the domain-completeness condition has to be added:

3. $\{\neg O_1 \wedge .. \wedge \neg O_n, Dom\} \models \neg O$ *(completeness)*

In the plane landing example above, the set {O1, O2, O3} is a complete OR-refinement of the higher-level obstacle *N-Ass* in a domain comprising property D4.

One may sometimes wish to consider all disjoint alternative subobstacles of an obstacle; the following additional condition has to be added in such cases:

4. for all $i \neq j$: $\{ O_i, O_j, Dom\} \models$ **false** *(disjointness)*

Section 8.4.3 will present a rich set of complete and disjoint obstacle refinement patterns.

Chaining the definitions in Sections 8.2.1 and 8.2.3 leads to the following straightforward proposition:

If O' is a subobstacle within an OR-refinement of an obstacle O that obstructs some goal G, then O' obstructs G as well.

8. 2. 4. Classifying obstacles

As mentioned in Section 3.2.4, goals are classified by the type of requirements they will drive about the agents concerned. For each goal category, corresponding obstacle categories may be defined. For example,

- Non-satisfaction obstacles are obstacles that obstruct the satisfaction of agent wishes (that is, Satisfaction goals);
- Non-information obstacles are obstacles that obstruct the generic goal of making agents informed about object states (that is, Information goals);
- Inaccuracy obstacles are obstacles that obstruct the consistency between the state of objects in the environment and the state of their representation in the software (that is, Accuracy goals);
- Hazard obstacles are obstacles that obstruct Safety goals;
- Threat obstacles are obstacles that obstruct Security goals.

Such obstacle categories may be further specialized into subcategories --e.g.,Indiscretion and Corruption obstacles are subcategories of Threat obstacles that obstruct goals in the Confidentiality and Integrity subcategories of Security goals, respectively [Amo94]; WrongBelief obstacles form a subcategory of Inaccuracy obstacles; and so on.

Knowing the (sub)category of a goal may prompt a search for obstructing obstacles in the corresponding category. More specific goal subcategories will of course result in more focussed search for corresponding obstacles. This provides the basis for heuristic identification of obstacles, as discussed in Section 8.4.4.

8. 2. 5. Goal obstruction vs. goals divergence

In the context of handling conflicts between multiple goals, [Lam98b] introduced the notion of divergent goals. Goals G1, G2, ..., Gn are said to be *divergent* iff there exists a boundary condition that makes them logically inconsistent with each other in the domain considered. We have shown that an obstacle corresponds to a boundary condition for the degenerate case where n=1. As a consequence, there are generic principles common to obstacle identification/resolution and divergence identification/resolution. However, handling exceptions to the achievement of a single goal and handling conflicts between multiple stakeholders' goals correspond to different problems and foci of concern for the requirements engineer. For example, the above notions of completeness and refinement are specifically introduced for obstacle analysis. The classification of obstacles and the heuristic rules for their identification is specific to obstacle analysis (see Section 8.4.4). As will be seen below, the common generic principles for identification/resolution yield specific instantiations and specializations for obstacle analysis. For example, the goal regression procedure can be simplified (see Section 8.4.1); the completion procedure is specific to obstacle analysis (see Section 8.4.2); obstruction refinement patterns are different from divergence patterns (see Section 8.4.3).

8. 3. Integrating Obstacles in the RE Process

First-sketch specifications of goals, requirements and assumptions tend to be too ideal; they are likely to be occasionally violated in the running system due to unexpected agent behavior [Lam95, Pot95]. The objective of obstacle analysis is to anticipate exceptional behaviors in order to derive more complete and realistic goals, requirements and assumptions.

A defensive extension of the goal-oriented process model described in Section 3.3 is depicted in Figure 8.15. (the arrows indicate data dependencies.) The main difference is the *obstacle analysis loop* introduced in the upper right part.

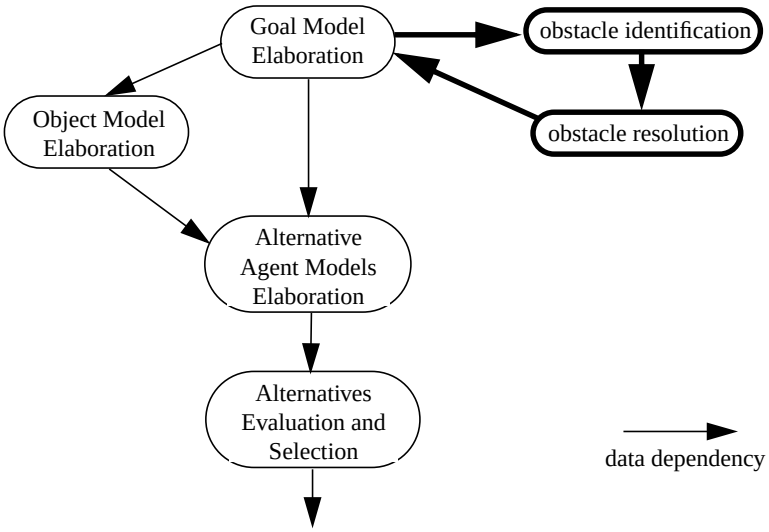


FIGURE 8.15. Obstacle analysis in goal-oriented requirements elaboration

During elaboration of the goal graph by elicitation and by refinement, obstacles are generated from goal specifications. Such obstacles may be recursively refined. (Section 8.4 will discuss techniques for supporting the obstacle identification/refinement process.)

The generated obstacles are resolved which results in a goal structure updated with new goals and/or transformed versions of existing ones. The resolution of an obstacle may be subdivided into two steps [Eas94]: the generation of alternative resolutions, and the selection of one among the alternatives considered. (Section 8.5 will discuss different tactics for resolution generation.)

The new goal specifications obtained by resolution may in turn trigger a new iteration of goal elaboration and obstacle analysis. Goals obtained from obstacle resolution may also refer to new objects/operations and require specific operationalizations.

A number of questions arise from this process model.

- *Obstacle identification:* From which goals in the goal graph should obstacles be generated? For some given goal, how extensive should obstacle generation be?

– The more specific the goal is, the more specific its obstructing obstacles will be. A high-level goal will produce high-level obstacles which will need to be refined significantly into sub-obstacles in order to identify precise circumstances that lead to the violation of the goal. It is much easier and preferable to elicit/refine what is wanted than what is *not* wanted. We therefore recommend that obstacles be identified from *terminal* goals assignable to individual agents.

– The extensiveness of obstacle identification will depend on the category and priority of the goal being obstructed. For example, obstacle identification should be exhaustive for Safety or Security goals; higher-priority goals deserve more extensive identification than lower-priority ones. Domain-specific cost-benefit analysis needs to be carried out to decide when the obstacle identification process should terminate.

- *Obstacle resolution:* For some given obstacle, how extensive should the generation of alternative resolutions be? For some set of alternative resolutions, how and when should a specific resolution be selected?

  As will be seen in Section 8.5, the generation of alternative resolutions correspond to the application of different tactics for resolving obstacles. The tactics include obstacle elimination, with subtactics such as obstacle prevention, goal substitution, agent substitution, goal deidealization, or object transformation; obstacle reduction; and obstacle tolerance, with subtactics such as obstacle mitigation or goal restoration. (Some of these tactics have been studied in other contexts of handling problematic situations -- e.g., deadlocks in parallel systems [Cof71]; exceptions and faults in fault-tolerant systems [And81, Cri91, Jal94, Gar99]; feature interaction in telecommunication systems [Kec98]; inconsistencies in software development [Nus96]; or conflicts between requirements [Rob97, Lam98b]).

– The range of tactics to consider and the selection of a specific tactics to apply will depend on the likelihood of occurrence of the obstacle, on the impact of such an occurrence (in number of goals being obstructed by the obstacle), and on the severity of the consequences of such an occurrence (in terms of priority of the goals being obstructed). Risk analysis and domain-specific cost-benefit analysis need to be deployed in order to provide a definite answer. Such analysis is outside the scope of this thesis.
– The selection of a specific resolution should not be done too early in the goal/obstacle analysis process. An obstacle identified at some point may turn out to be more severe later on (e.g., because it then appears to also obstruct new important goals being elicited). Premature decisions may stifle the consideration of alternatives that may appear to be more appropriate later on in the process [Eas94].

- *Goal-obstacle analysis iteration:* When should the intertwined processes of goal elaboration and obstacle analysis stop?

  The goal-obstacle analysis loop in Figure 8.15 may terminate as soon as the obstacles that remain are considered acceptable without any resolution. Risk analysis needs again to be carried out together with cost-benefit analysis in order to determine acceptability thresholds.

8. 4. Generating Obstacles

According to the definition in Section 8.2.1, the identification of obstacles obstructing some given goal in the considered domain proceeds by iteration of two steps:

(1) Given the goal specification, find some assertion that may obstruct it;

(2) Check that the candidate obstacle thereby obtained is consistent with the domain theory available.

Note that checking that the obstacle is consistent with the domain allows one to discard obstacles that are known to be physically impossible in the domain. It does not allow one to identify further obstacles.

We therefore concentrate on step (1) and present techniques for deriving candidate obstacles whose domain consistency/feasibility can be checked subsequently. We successively discuss:

- a formal calculus of preconditions for obstruction,
- the use of formal obstruction patterns to shortcut formal derivations,
- the use of identification heuristics based on obstacle classifications as a cheap, informal alternative to formal techniques.

8. 4. 1. Regressing goal negations

The first technique is based on the obstruction condition defining an obstacle in Section 8.2.1. Given the goal assertion *G*, it consists of calculating preconditions for obtaining the negation $\neg G$ from the domain theory. Every precondition obtained defines a candidate obstacle. This may be achieved using a regression procedure which can be seen as a counterpart of Dijkstra's precondition calculus [Gri81] for declarative representations. Variants of this procedure have been used in AI planning [Wal77], in explanation-based learning [Lam91], and in requirements engineering to identify divergent goals [Lam98b]. We first explain the general procedure before showing how it can be specialized and simplified for obstacle generation.

Consider a meeting scheduler system and the goal stating that intended people should participate in meetings they are aware of and which fit their constraints:

Goal *Achieve* [InformedParticipantsAttendance]
FormalDef $\forall$ m: Meeting, p: Participant
$\quad$ Intended (p, m) $\wedge$ Informed (p, m) $\wedge$ Convenient (p, m)
$\quad \Rightarrow \Diamond$ Participates(p, m)

The initialization step of the regression procedure consists of taking the negation of this goal, which yields

(NG) $\Diamond \exists$ m: Meeting, p: Participant
Intended (p, m) $\wedge$ Informed (p, m) $\wedge$ Convenient (p, m)
$\wedge \Box \neg$ Participates(p, m)

(Such initialization may already produce precise, feasible obstacles in some cases; see other examples below.)

Suppose now that the domain theory contains the following property:

$\forall$ m: Meeting, p: Participant
Participates(p, m) $\Rightarrow$ Holds (m) $\wedge$ Convenient (p, m)

This domain property states that a necessary condition for a person to participate in a meeting is that the meeting is being held and its date/location is convenient to her. A logically equivalent formulation is obtained by contraposition:

(D) $\forall$ m: Meeting, p: Participant
$\neg$ [Holds (m) $\wedge$ Convenient (p, m)] $\Rightarrow \neg$ Participates(p, m)

The consequent in (D) unifies with a litteral in (NG); regressing (NG) through (D) then amounts to replacing in (NG) the matching consequent in (D) by the corresponding antecedent. We have thereby formally derived the following potential obstacle:

(O1) $\Diamond \exists$m: Meeting, p: Participant
Intended (p, m) $\wedge$ Informed (p, m) $\wedge$ Convenient (p, m)
$\wedge \Box$ [$\neg$ Holds (m) $\vee \neg$ Convenient (p, m)]

This obstacle covers two situations, namely, one where some meeting never takes place and the other where a participant invited to a meeting whose date/location was first convenient to her is no longer convenient when the meeting takes place. Using the OR-refinement techniques described in Section 8.4.3 we will thereby obtain two subobstacles that could be named MeetingPostponedIndefinitely and LastMinuteImpediment, respectively.

Assuming the domain theory takes the form of a set of rules A $\Rightarrow$ C, a temporal logic variant of the regression procedure found in [Lam91] can be described as follows.

Initial step:
take O := $\neg$ G
Inductive step:
let A $\Rightarrow$ C be the domain rule selected,
with *C* matching some subformula *L* in *O* whose
occurrences in *O* are all positive;
then μ := mgu (L, C);
O := O [L / A.μ]

This procedure relies on the following definitions and notations:

– for a formula scheme $\varphi(u)$ with one or more occurrences of the sentence symbol u, an occurrence of u is said to be positive in φ if it does not occur in a subformula of the form $p \leftrightarrow q$ and it is embedded in an even (explicit or implicit) number of negations;
– mgu (F1, F2) denotes the most general unifier of F1 and F2;
– F.μ denotes the result of applying the substitutions from unifier μ to F;
– F [F1 / F2] denotes the result of replacing every occurrence of F1 in formula F by F2.

The soundness of the regression procedure follows from a monotonicity property of temporal logic [Man92, p.203]:

If all occurrences of u in $\varphi(u)$ are positive, then
$(p \Rightarrow q) \rightarrow (\varphi(p) \Rightarrow \varphi(q))$
is valid.

Every iteration in the regression procedure produces potentially finer obstacles to the goal under consideration; it is up to the specifier to decide when to stop, depending on whether the obstacles obtained are meaningful and precise enough (i) to be able to asses their probability of occurrence, and (ii) to see appropriate ways of resolving them through tactics discussed in Section 8.5.

In the example above only one iteration was performed. Regressing obstacle (O1) above further through a domain property like

Convenient (p, m) ⇒ m.Date **in** p.Constraints
∧ m.Location **in** p.Constraints

would have produced finer sub-obstacles to the goal

Achieve [InformedParticipantsAttendance],

namely, the date being no longer convenient or the location being no longer convenient when the meeting takes place.

Exploring the space of potential obstacles derivable from the domain theory is achieved by *backtracking* on each domain rule applied to select another applicable one. After having selected rule (D) in the example above, one could select the following other domain rule stating that another necessary condition for participation is that the meeting date the participant has in mind corresponds to the actual date of the meeting:

(D') ∀ m: Meeting, p: Participant
Participates(p, m) ⇒ ∃ M: $Belief_p$(m.Date = M) ∧ m.Date = M

The $Belief_{ag}$ construct in this formalization is sometimes used to capture Accuracy goals and Inaccuracy obstacles; it is linked to the $Knows_{ag}$ construct by the following property:

$$Knows_{ag}(P) \equiv Belief_{ag}(P) \wedge P$$

where ag denotes an agent instance, P a fact, and the KAOS built-in predicate $Knows_{ag}(P)$ means that the truth value of P in ag's local memory coincides with the actual truth value of P.

Regressing the goal negation (NG) above through property (D') now yields the following new obstacle:

(O2) ◊ ∃m: Meeting, p: Participant
Intended (p, m) ∧ Informed (p, m) ∧ Convenient (p, m)
∧ ❑ ∀M: ¬ [$Belief_p$(m.Date = M) ∧ m.Date = M]

This obstacle, in the Inaccuracy category, could be named

ParticipantBelievesWrongDate.

Further backtracking on other applicable rules would generate other obstacles obstructing the goal Achieve[InformedParticipantsAttendance] such as, e.g., ParticipantNotInformedInTime, InvitationNotKnown, etc.

The examples above exhibit a **simplified procedure** for generating obstacles to *Achieve* goals of the form C ⇒ ◊ T:

1. Negate the goal, which yields a pattern ◊ (C ∧ ❑ ¬ T);
2. Find *necessary* conditions for the target condition T in the domain theory;
3. Replace the negated target condition in the pattern resulting from step 1 by the negated necessary conditions found; each such replacement yields a potential obstacle. If needed, apply steps 2, 3 recursively.

A dual version of this simplified procedure can be used for goals having the *Maintain* patterns C ⇒ T, C ⇒ ❑ T, or C ⇒ T *W* N. For the plane landing example in Section 8.2.2, it generates the obstacles O1, O2, and O3 to the assumption *Ass* in a straightforward way.

In practice, the domain theory does not necessarily need to be very rich at the beginning. Given a target condition T in a goal such as C ⇒ ◊ T, the requirements engineer may *incrementally elicit necessary conditions* for T by interaction with domain experts and clients.

To give a more extensive idea of the space of obstacles that can be generated systematically using this technique, Figure 8.16 shows a goal AND-refinement tree, derived by instantiation of a frequent refinement pattern from [Dar96], together with corresponding obstacles that were generated by regression (universal quantifiers have been left implicit).

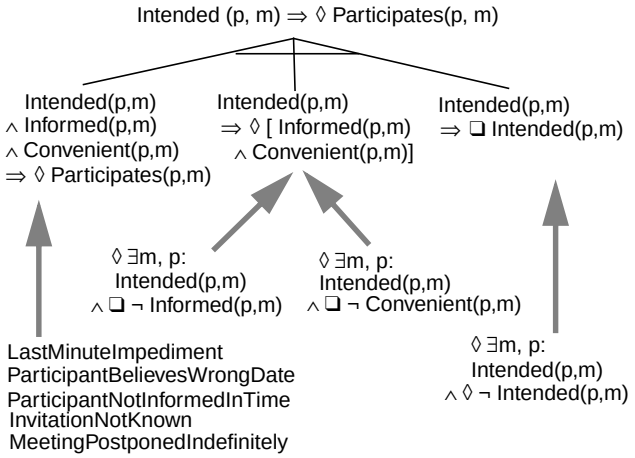


FIGURE 8.16. Goal refinement and obstacles derived by regression

8. 4. 2. Completing a set of obstacles

The domain-completeness condition in Section 8.2.2 suggests a procedure for completing a set of obstacles O_1, ..., O_k already identified for some goal G.

As noted in Section 8.2.1, G has the general form ❑ GC whereas O_i has the general form ◊ OC_i. The completion procedure can be described as follows.

1. Form the complementary assertion
   $O^* = \Diamond (\neg GC \wedge \neg OC_1 \wedge ... \wedge \neg OC_k)$;
2. Check the consistency of O* with Dom;
3. If O* is domain-consistent and too unspecific, regress it through Dom or generate subobstacles using refinement patterns, to yield finer obstacles SO* ;
4. If needed, apply steps 1-3 recursively to the SO*'s.

It is easy to check that the set {O*, O_1, ..., O_k} obtained by Step 1 satisfies the domain-completeness condition in Section 8.2.2 in which the domain is temporarily not considered. Considering the domain in the next steps allows O* to be checked for consistency and refined if necessary. A frequent simplification arises from Step 3 when O* has the form P ∧ P1 and a domain property is found having the form P ⇒ P1. A one-step regression then yields O = P.

Back to the plane landing example in Section 8.2.2, Step 1 of the completion procedure applied to the assumption

$$\text{MovingOnRunway} \Rightarrow \text{WheelsTurning} \quad \text{(Ass)}$$

and the obstructing obstacles

$$\Diamond\,(\text{MovingOnRunway} \wedge \neg\, \text{WheelsOut})\text{(O1)}$$
$$\Diamond\,(\text{MovingOnRunway} \wedge\wedge\, \text{WheelsBlocked})\text{(O2)}$$
$$\Diamond\,(\text{MovingOnRunway} \wedge\wedge\, \text{Aquaplaning})\text{(O3)}$$

yields

$$O* = \Diamond\,(\text{MovingOnRunway} \wedge \neg\, \text{WheelsTurning} \wedge \text{WheelsOut} \wedge \neg\, \text{WheelsBlocked} \wedge \neg\, \text{Aquaplaning})$$

This candidate obstacle is inconsistent with the domain if property (D4) is found in *Dom* (see Section 8.2.2). If not, further regression/refinement through *Dom* should be undertaken to find out more specific causes/subobstacles of O* in order to complete the set (O1)-(O3). Such refinement may be driven by patterns as we discuss now.

8. 4. 3. Using obstruction refinement patterns

As introduced in Section 8.2.3, obstacles may be AND/OR-refined into subobstacles. AND-refinements yield more "primitive" obstacles, that is, obstacles for which (i) the probability of their occurrences can be assessed, and (ii) effective ways of resolving them can be envisioned more easily. On the other hand, domain-complete OR-refinements are in general desirable for critical goals; they yield a domain-complete set of alternative subobstacles that can be made disjoint if necessary.

Section 8.4.1 already contained examples of obstacle refinements. The obstacle LastMinuteImpediment was in fact OR-refined into two alternative subobstacles using the domain theory, namely, the date being no longer convenient *or* the location being no longer convenient. Figure 8.16 also shows an example of OR-refinement of the obstacle obstructing the goal in the middle of the goal tree; this obstacle, not explicitly represented there, has been formally OR-refined into the two subobstacles in the middle (which could be named MeetingNeverNotified and MeetingNeverConvenient, respectively). The latter subobstacles may be refined in turn. Similarly, the obstacle ParticipantBelievesWrongDate that was derived in Section 8.4.1 could be OR-refined into alternative subobstacles like WrongDateCommunicated, ParticipantConfusesDates, etc.

The AND/OR refinement of obstacles may be seen as a formal, *goal-oriented* form of fault-tree analysis [Lev95] or threat-tree analysis [Amo94]. Such analysis is usually done in an informal way through interaction with domain experts and clients; our aim here is to derive complete fault/threat-trees formally.

The regression procedure in Section 8.4.1 is a first technique to achieve this; alternatively, one may use obstacle refinement patterns to shortcut the formal derivations involved in the regression procedure.

The general principle is similar to goal refinement patterns [Dar96] and divergence detection patterns [Lam98b]. A library of generic refinement patterns is built; each pattern is a refinement tree where the root is a generic assertion to be refined and the leaves are generic refining assertions. The correctness of each pattern is proved formally *once and for all*.

The patterns for goal obstruction are specific in that the roots of refinement trees are negated goals. The generation of (sub)obstacles to some goal then proceeds by selecting patterns whose root matches the negation of that goal, and by instantiating the leaves accordingly. The requirements engineer is thus relieved of the technical task of doing the formal derivations required in Section 8.4.1. The patterns can be seen as high-level inference rules for deriving finer obstacles.

All obstruction patterns in this paper were proved formally correct using the STeP verification tool [Man96]. As we will see, the notion of correctness is different for AND- and OR-refinement patterns. We discuss them successively.

8. 4. 3. 1. *AND*-refinement patterns

Figures 8.17 to 8.19 show a sample of frequent AND-refinement patterns for obstacles that obstruct *Achieve* and *Maintain* goals, respectively.

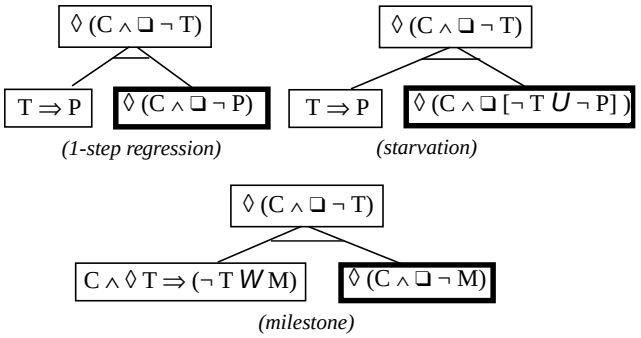


FIGURE 8.17. AND-refinement patterns for obstacles to the goal C ⇒ ◊

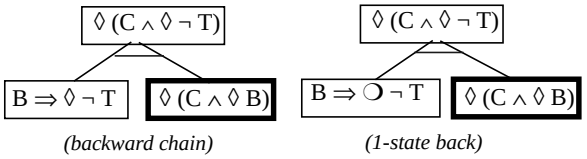


FIGURE 8.18. AND-refinement patterns for obstacles to the goal C ⇒ ❑

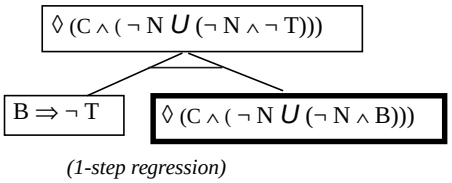


FIGURE 8.19. AND-refinement patterns for obstacles to the goal
C ⇒ T W N

The *root* assertion in each AND-tree corresponds to the negation of the goal being obstructed. (Remember that there is an implicit outer ❑-operator in every strong implication; this causes the outer ◊-operator to appear there.) The *left* child assertion may correspond to a domain property, to another requirement/assumption, or to a companion subobstacle. In the 1-step regression and starvation patterns, it will typically correspond

to a domain rule T ⇒ P. In the milestone pattern, it defines a necessary milestone *M* for reaching the target predicate *T*. The left child assertion often guides the identification of the subobstacle captured by the *right* child assertion.

Obstacle refinement patterns may thus help identifying both *subobstacles* and *domain properties*. Also note that the 1-step regression pattern in Figures 8.17 and 8.19 correspond to the regression procedure in Section 8.4.1 where only one iteration is performed.

As an example of using the starvation pattern in Figure 8.17, consider a general resource management system and the goal

∀ u: User, r: Resource
Requesting (u, r) ⇒ ◊ Allocated (r, u)

The domain property

Allocated (r, u) ⇒ ¬ ∃ u' ≠ u: Allocated (r, u')

suggests reusing the starvation pattern with instantiations

C: Requesting (u, r)

T: Allocated (r, u) , *P*: ¬ ∃ u'≠ u: Allocated (r, u')

The following starvation obstacle has been thereby derived:

◊ ∃ u: User, r: Resource
Requesting (u, r) ∧ ❑ [¬ Allocated (r, u) *U* ¬ ∃ u'≠ u: Allocated (r, u')]

As an example of using the 1-step regression pattern in Figure 8.19, consider the LAS ambulance dispatching system [LAS93] and the goal stating that an ambulance allocated to an incident should remain allocated to that incident until it has arrived at the incident scene. This goal may be formalized by

∀ a: Ambulance, inc: Incident
Allocation (a, inc) ⇒ Allocation (a, inc) *W* Intervention (a, inc)

We know from the domain that an ambulance can be allocated to at most one incident at a time:

Allocation (a, inc) ⇒ ¬ ∃ inc' ≠ inc: Allocation (a, inc')

This property suggests using the 1-step regression pattern with the following instantiations

C: Allocation (a, inc) , *T:* Allocation (a, inc)
N: Intervention (a, inc) , *B*: ∃ inc' ≠ inc: Allocation (a, inc')

The following subobstacle is thereby derived:

◊ ∃ a: Ambulance, inc: Incident
Allocation (a, inc)
∧ ¬ Intervention (a, inc) *U*
¬ Intervention (a, inc) ∧ ∃ inc' ≠ inc: Allocation (a, inc')

This obstacle captures a situation in which an ambulance allocated to an incident becomes allocated to another incident before its intervention at the first one.

A more extensive set of obstacle AND-refinement patterns is given in Tables 8.1-8.4. Each table corresponds to a specific kind of goal. Each row in a table represents an AND-refinement of the negation of the goal associated with the table. The lower a row is in a

	assertion	subobstacle
1-step regress	$S \Rightarrow P$	$\Diamond [R \wedge \Box \neg P]$
	$S \Rightarrow P$	$\Diamond [R \wedge (\neg S \mathcal{U} \Box \neg P)]$
starvation	$S \Rightarrow P$	$\Diamond [R \wedge \Box (\neg S \mathcal{U} \neg P)]$
missing source	$R \wedge \Diamond S \Rightarrow P$	$\Diamond [R \wedge \neg P]$
non-persistence	$R \wedge \Diamond S \Rightarrow P \mathcal{W} S$	$\Diamond [R \wedge \neg S \mathcal{U} (\neg P \wedge \neg S)]$
non-persistence	$R \wedge \Diamond S \Rightarrow P \mathcal{W} (P \wedge S)$	$\Diamond [R \wedge (\neg S \mathcal{U} \neg P)]$
milestone	$R \wedge \Diamond S \Rightarrow \neg S \mathcal{W} M$	$\Diamond [R \wedge \Box \neg M]$
blocking	$B \Rightarrow \Box \neg S$	$\Diamond [R \wedge (\neg S \mathcal{U} B)]$
substitution	$S' \Rightarrow \Box \neg S \wedge \blacksquare \neg S$	$\Diamond [R \wedge \Diamond S']$
strengthening	$R \wedge \Diamond S \Rightarrow \Diamond [P \wedge (P \mathcal{W} S)]$	$\Diamond [R \wedge \Box \neg P]$
starvation	$R \wedge \Diamond S \Rightarrow \Diamond [P \wedge (P \mathcal{W} S)]$	$\Diamond [R \wedge (\neg S \mathcal{U} \Box \neg P])]$
	$R \wedge \Diamond S \Rightarrow \Diamond [P \wedge (P \mathcal{W} S)]$	$\Diamond [R \wedge (\neg S \mathcal{U} (\neg S \wedge \Box \neg P))]$

TABLE 8.1. Patterns of obstacles to the goal $R \Rightarrow \Diamond S$

	assertion	subobstacle
1-step regress	$Q \Rightarrow C$	$\Diamond [P \wedge \Diamond \neg C]$
backward	$C \Rightarrow \Diamond \neg Q$	$\Diamond [P \wedge \Diamond C]$
1-state back	$C \Rightarrow o \neg Q$	$\Diamond [P \wedge \Diamond C]$

TABLE 8.2. Patterns of obstacles to the goal $P \Rightarrow \Box Q$

	assertions	subobstacle
	$Q \wedge C \Rightarrow o \neg Q$, $P \wedge C \Rightarrow o P$	$\Diamond [P \wedge Q \wedge C]$
	$Q \wedge C \Rightarrow o \neg Q$, $\neg P \wedge C \Rightarrow o P$	$\Diamond [\neg P \wedge Q \wedge C]$
	$\neg P \wedge C \Rightarrow o P$, $\neg Q \wedge C \Rightarrow o \neg Q$	$\Diamond [\neg P \wedge \neg Q \wedge C]$

TABLE 8.3. Patterns of obstacles to the goal $\Box (P \rightarrow Q)$

	assertion	subobstacle
back state	$p \Rightarrow o \neg q$	$\Diamond p$

TABLE 8.4. Patterns of obstacles to the goal $\Box q$

table, the more specific the corresponding assertion and subobstacle are. The assertions in the first column may represent a domain property, a requirement or a companion subobstacle. Table 8.4 may be seen to correspond to the backward construction of a fault-tree from a state machine [Rat96]; *p* and *q* are intended to be state predicates there.

All AND-refinement patterns in Tables 8.1-8.4 were proved correct using STeP [Man96] --by this we mean that the entailment and consistency conditions in Section 8.2.3 were formally verified.

8. 4. 3. 2. Complete OR-refinement patterns

Figures 8.20 shows a pattern for refining the obstruction of an *Achieve* goal $C \Rightarrow \Diamond T$ into a complete set of disjoint alternative subobstacles (see Section 8.2.3 for the definition of completeness and disjointness). The goal negation $\Diamond (R \wedge \Box \neg S)$ is AND-refined into two child nodes; the left child assertion may be a domain property, an assumption or a requirement (in this case it defines what a milestone is); the right child node is an OR-node refined into two alternative subobstacles.

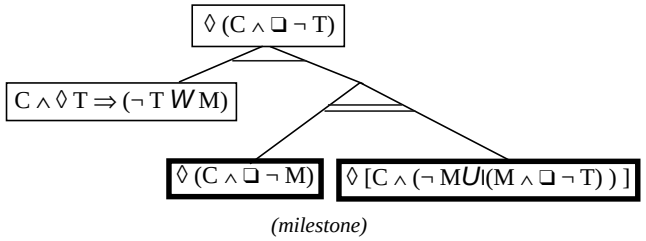


FIGURE 8.20. OR-refinement pattern for obstacles to the goal $C \Rightarrow \Diamond T$

As an example of using this pattern, consider the meeting scheduler system again and the goal stating that participants' time/location constraints should be provided if requested [Lam95]:

$\forall$ m: Meeting, p: Participant
ConstraintsRequested (p, m) $\Rightarrow \Diamond$ ConstraintsProvided (p, m)

An obvious milestone condition for a participant to provide her constraints is that a request for constraints is reaching her. This suggests using the milestone pattern in Figure 8.20 with the following instantiations:

C: ConstraintsRequested (p, m) *T:* ConstraintsProvided (p, m)
M: RequestReached (p, m)

The milestone pattern then generates the formalized domain property

$\forall$ m: Meeting, p: Participant
ConstraintsRequested (p, m) $\wedge \Diamond$ ConstraintsProvided (p, m)
$\Rightarrow$ [$\neg$ ConstraintsProvided (p, m) *W* RequestReached (p, m)]

together with a complete set of alternative subobstacles to the goal above:

$\Diamond \exists$ m: Meeting, p: Participant
ConstraintsRequested (p, m) $\wedge \Box \neg$ RequestReached (p, m)

or

$\Diamond \exists$ m: Meeting, p: Participant
ConstraintsRequested (p, m) $\wedge$
$\neg$ RequestReached (p, m) *U*
(RequestReached (p, m) $\wedge \Box \neg$ ConstraintsProvided (p, m)

The refinement may then proceed further to find out finer subobstacles in each alternative; this will yield causes for a request not reaching an invited participant and causes for a participant not providing her constraints in spite of the request having reached her, respectively.

assertion	obstacle	obstacle	obstacle
$S \Leftrightarrow P \wedge Q$	$\Diamond [R \wedge \Box \neg P]$	$\Diamond [R \wedge \Box \neg Q]$	$\Diamond [R \wedge \Diamond P \wedge \Diamond Q \wedge \Box \neg (P \wedge Q)]$
$S \Rightarrow P$	$\Diamond [R \wedge \Box \neg P]$	$\Diamond [R \wedge \Diamond P \wedge \Box \neg S]$	
$S \Rightarrow P$	$\Diamond [R \wedge \neg S \mathcal{U} \Box \neg P]$	$\Diamond [R \wedge \Diamond P \wedge \Box \neg S]$	
$S \Rightarrow P$	$\Diamond [R \wedge \Box (\neg S \mathcal{U} \neg P)]$	$\Diamond [R \wedge \Diamond (P \mathcal{W} (P \wedge S)) \wedge \Box \neg S]$	
$R \wedge \Diamond S \Rightarrow P$	$\Diamond [R \wedge \neg P]$	$\Diamond [R \wedge P \wedge \Box \neg S]$	
$R \wedge \Diamond S \Rightarrow P \mathcal{W} S$	$\Diamond [R \wedge \neg S \mathcal{U} (\neg P \wedge \neg S)]$	$\Diamond [R \wedge P \mathcal{W} S \wedge \Box \neg S]$	
$R \wedge \Diamond S \Rightarrow P \mathcal{W} (P \wedge S)$	$\Diamond [R \wedge \neg S \mathcal{U} \neg P]$	$\Diamond [R \wedge \Box \neg S \wedge P \mathcal{W} (P \wedge S)]$	
$R \wedge \Diamond S \Rightarrow \neg S \mathcal{W} M$	$\Diamond [R \wedge \Box \neg M]$	$\Diamond [R \wedge \neg M \mathcal{U} (M \wedge \Box \neg S)]$	
$B \Rightarrow \Box \neg S$	$\Diamond [R \wedge \neg S \mathcal{U} B]$	$\Diamond [R \wedge \Box \neg S \wedge \neg B \mathcal{W} S]$	
$P \Rightarrow \Box \neg S \wedge \blacksquare \neg S$	$\Diamond [R \wedge \Diamond P]$	$\Diamond [R \wedge \Box \neg S \wedge \Box \neg P]$	

TABLE 8.5. Obstacle OR-refinement for the goal $R \Rightarrow \Diamond S$

assertion	obstacle	obstacle
$Q \Leftrightarrow Q1 \wedge Q2$	$\Diamond [R \wedge \neg Q1]$	$\Diamond [R \wedge \neg Q2]$

TABLE 8.6. Obstacle OR-refinement for the goal $\Box (P \rightarrow Q)$

A more extensive set of complete and disjoint OR-refinement patterns is given in Tables 8.5-8.6. Each table corresponds to a specific kind of goal. Each row in a table represents a refinement of the negation of the goal associated with the table; the thick vertical line separator represents an AND whereas the double line separators represent an OR. Some of the patterns in these tables will be used in the obstacle analysis for the London Ambulance System in Chapter 9.

All OR-refinement patterns in Tables 8.5-8.6 were proved correct using STeP [Man96] -- by this we mean that the entailment, consistency, disjointness, and domain-completeness conditions in Section 8.2.3 were formally verified. In the latter case, the formulas in the assertion column were taken as the generic domain property forming *Dom*.

8. 4. 4. Informal obstacle identification

Informal heuristics may be used to help identify obstacles without necessarily having to go through formal techniques every time. Although they are easier to deploy, the result will be much less accurate, and not guaranteed to be formally correct and complete.

Such heuristics are rules of thumb taking the form: "**if** the specification has such or such characteristics **then** consider such or such type of obstacle to it". The general principle is somewhat similar in spirit to the use of HAZOP-like guidewords for eliciting hazards [Lev95] or, more generally, to the use of safety checklists [Jaf91, Som97].

Our heuristics are based on goal/obstacle classifications (see Section 3.4), on formal obstruction patterns we have identified, and on past experience in identifying obstacles. General heuristics are independent of any particular class of goals; more specific heuristics are associated with some specific class.

General heuristics refer to the KAOS meta-model only (see the concepts defined in Section 3.2). Here are a few examples to illustrate the approach.

- **If** an agent has to monitor some object in order to guarantee the goal it is assigned to **then** consider the following types of obstacles:

– InfoUnavailable: the necessary information about the object state is not available to the agent;
– InfoNotInTime: the necessary information about the object state is available too late;
– WrongBelief: the necessary information about the object state as recorded in the agent's memory is different from the actual state of this object. (In the meeting scheduler example, this heuristic might have helped identifying obstacles like ParticipantBelievesWrongDate --see Section 8.4.1; for an electronic reviewing process an obstacle like ReviewerBelievesWrongDeadline could be identified in a similar way.)

The WrongBelief obstacle class can be further refined into subclasses such as:

– InfoOutDated: the information provided to the agent is no longer correct at the time of use;
– InfoForgotten: the information provided to the agent is no longer available at the time of use;
– WrongInference: the agent has made a wrong inference from the information available;
– InfoConfusion: the agent confuses the necessary information about the object state with some other information.

InfoConfusion obstacles can be refined in turn, e.g.,

– InstanceConfusion: the agent confuses the necessary information about the object state with information about *another* instance of object within the same class [Pot95] (instance confusion is also related to the notion of 'identity' concern in [Jac2K]);
– ValueConfusion: the agent confuses different values for an attribute of the same object;
– UnitConfusion: the agent confuses different units in terms of which values of an object attribute are expressed.

In the meeting scheduler example, these heuristics might have helped identify several obstacles among those derived formally, e.g., participants confusing meetings or dates, meeting initiators confusing participants which results in wrong people being invited, confusion in constraints, etc. In an ambulance dispatching system, an obstacle like an ambulance going to a wrong place could be identified thereby.

An important specialization of InfoConfusion obstacles in the aviation domain is ModeConfusion where pilot agents become confused about what the cockpit software agent is doing; obstacles in this category receive increasing attention as they have been recognized to be responsible for a significant number of critical incidents [But98].

- **If** an agent requires some resource in order to guarantee the goal it is assigned to **then** consider obstacles in the following categories: ResourceUnavailable, ResourceTooLate, ResourceOutOfOrder, WrongResource, ResourceConfusion, and so on.
- **If** a persistent condition is necessary to reach the target condition from the source condition in an *Achieve* goal, **then** consider an obstacle in which the persistent condition becomes false before reaching the target condition.

The latter heuristic rule corresponds to a natural language rephrasing of the *missing persistence* pattern in Table 8.1; it suggests how similar heuristics can be formulated from the other patterns.

More *specific heuristics* refer to goal classifications. Here are a few examples.

- **If** a *MessageDelivered* goal in the Information goal category is considered, **then** consider obstacles like *MessageUndelivered*, *MessageDeliveredAtWrongPlace*, *MessageDeliveredAtWrongTime*, *MessageCorrupted*.
- **If** a goal being considered is in the StimulusResponse category, **then** consider the following types of obstacles:

– StimulusIgnored, TooLatePickUp, IncorrectValue, or StimuliConfused obstacles to the abstract goal StimulusPickedUp;

– NoResponse, ResponseTooLate, ResponseIgnored, or WrongResponse obstacles to the abstract companion goal ResponseProvided.

Obstacles can also be identified by analogy with obstacles in similar systems, using analogical reuse techniques [Mas97].

8. 5. Resolving Obstacles

The generated obstacles need to be resolved in some way or another. As discussed in Section 8.3, the resolution process covers two aspects: the *generation* of alternative resolutions and the *selection* of one resolution among those identified. Which resolution to apply and when to apply it will depend on risk/cost-benefit analysis based on the likelihood of occurrence of the obstacle and on the severity of its consequences. We will not discuss selection tactics here; we concentrate on the generation of alternative resolutions.

Such resolutions correspond to different *specification elaboration tactics* that may be applied. They can be classified into three broad classes depending on whether the obstacle is eliminated (Section 8.5.1), reduced (Section 8.5.2), or tolerated (Section 8.5.3). Some of these tactics have been studied in other contexts of handling problematic situations --e.g., deadlocks in parallel systems [Cof71]; exceptions and faults in fault-tolerant systems [And81, Cri91, Jal94, Gar99]; feature interaction in telecommunication systems

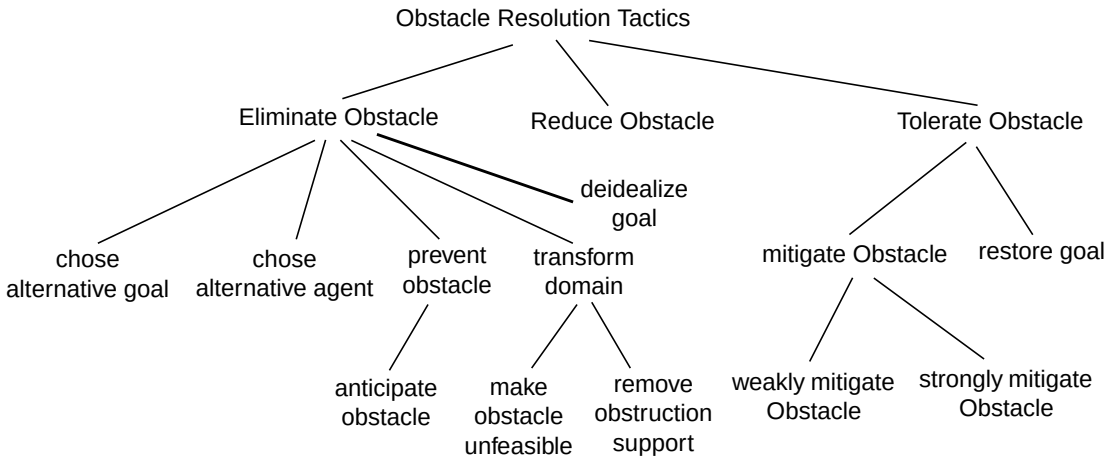


FIGURE 8.21. The library of obstacle resolution tactics

[Kec98]; inconsistencies in software development [Nus96]; or conflicts between requirements [Rob97, Lam98b]. The objective here is to specialize such tactics to the resolution of obstacles to goals during requirements engineering, and to make them explicit in terms of specification transformation rules in the formal framework of temporal logic.

The obstacle resolution process will result in a transformed goal structure, transformed requirements specifications, and transformed domain properties in some cases.

The library of obstacle resolution tactics is shown in Figure 8.21.

8. 5. 1. Obstacle Elimination

Eliminating an obstacle requires one among the conditions defining an obstructing obstacle in Section 8.2.1 to be inhibited; the obstruction should be avoided or the obstacle should be made inconsistent/infeasible within the domain. The strategies below address one of the conditions or the other.

8. 5. 1. 1. Goal substitution

A most effective way of resolving an obstacle is to identify an *alternative goal refinement* for some higher-level goal, in which the obstructed goal and obstructing obstacle are no longer present. In the meeting scheduler example, one may eliminate the obstacle ElectronicAgendaNotMaintained that obstructs the goal ElectronicAgendaUpToDate by choosing an alternative refinement for the father goal ParticipantsConstraintsKnown (see Figure 8.22); the alternative goal refinement consists in introducing the two companion goals ConstraintsRequested (under responsibility of the meeting scheduling software) and ConstraintsProvided (still under joint responsibility of participants and the email system).

Choosing an alternative goal refinement will in general result in a different design for the composite system.

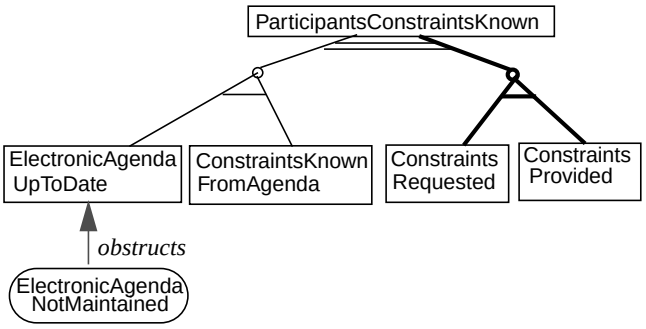


FIGURE 8.22. choose alternative goal

8. 5. 1. 2. Agent substitution

Another way of overcoming the obstacle is to consider *alternative agent assignments* so that the obstacle scenario may no longer occur. This will in general result in different system proposals, in which more or less functionality is automated and in which the interaction between the software and its environment may be quite different.

Back to our meeting scheduler example, one might overcome the obstacle ParticipantNotResponsive to the goal ConstraintsProvided by assigning the responsibility for that goal to the participant's Secretary instead (to overcome subobstacles such as EmailNotCheckedRegularly or ParticipantTooBusy), or by assigning the responsibility for the goal ParticipantsConstraintsRequested to the meeting initiator (rather than the meeting scheduling software) --through email, phone calls, etc.

In the electronic reviewing example, one could introduce a software agent for checking that no occurrences of the reviewer's name are found in the review (to overcome the obstacle NonAnonymousReview); a software agent for checking destination tables (to overcome the obstacle MessageSentToWrongPerson); and so on.

Agent substitution may entail goal substitution and vice-versa.

8. 5. 1. 3. Obstacle prevention

The tactic prevent obstacle resolves the obstruction by *adding a new goal* requiring that the obstacle be avoided.

Remember that a goal G has the general form ❑ GC whereas an obstacle O to G has the general form ◊ OC. To prevent O from being ever satisfied, the following *Avoid* goal is thus introduced:

G*: ❑ ¬ OC

AND/OR refinement and obstacle analysis may then be applied to the new goal in turn.

Back to our meeting scheduler example, consider the obstacle MeetingForgotten that obstructs the goal Achieve [InformedParticipantsAttendance] in Figure 8.16. The tactics prevent obstacle yields the new goal Avoid [MeetingForgotten]. The latter may then be refined into a requirement Achieve [MeetingReminded] under responsibility of the meet-

ing scheduling software. Another example of obstacle prevention in a train control system is the introduction of an automatic brake facility (with corresponding goals and agents) to prevent trains from exceeding their speed limit.

It may turn out, after checking with domain experts, that the assertion $\Box \neg OC$ introduced for obstacle prevention is not a goal/requirement but a domain property that was missing from the domain theory *Dom*, making the obstacle unfeasible in the domain (see the domain consistency condition in Section 8.2.1). In such cases the domain theory will be updated instead of the goal structure, and the obstacle will be discarded.

The tactic anticipate obstacle is a subtactic for refining obstacle prevention goals. It is applicable when some persistent condition *P* can be found such that *P* must persist during some time interval for the obstacle condition OC to become true:

$$OC \Rightarrow \blacksquare_{\leq d} P$$

In such a case, the obstacle prevention goal may be refined by introducing the subgoal

$$G^*: \quad P \Rightarrow \Diamond_{\leq d} \neg P$$

For obstacles to Security goals, for example, one might have the following instantiations:

OC: InformationCorruptedByAgent
P: IntrusionUndetected

Obstacle anticipation patterns may be used when an event can be identified that necessarily precedes the truth of the obstacle condition.

8. 5. 1. 4. Goal Deidealization

It is often the case that obstacles are found to obstruct first-sketch goal formulations because the latter are too ideal. Such goal formulations should then be deidealized so that they cover the behaviors captured by the obstacle. The principle is to *transform the goal* being obstructed in order to make the obstruction disappear.

Let us suggest the technique via an example first.

Consider the obstacle ParticipantNotInformedInTime in Figure 8.16 which obstructs the goal

$$Intended(p, m) \wedge Informed(p, m) \wedge Convenient(p, m) \Rightarrow \Diamond Participates(p, m)$$

The idea is to make the obstructed goal more liberal, that is, to weaken it so that it covers the obstacle. In this case the goal weakening is achieved by strengthening its antecedent:

$$Intended(p, m) \wedge Informed\mathbf{InTime}(p, m) \wedge Convenient(p, m) \Rightarrow \Diamond Participates(p, m)$$

The predicate Informed**InTime** (p, m) is derived from the corresponding obstacle; it requires participants to be kept informed during a time period starting at least *N* days before the meeting date:

$$Informed\mathbf{InTime}(p, m) \equiv \blacksquare_{\leq (m.Date - Nd)} Informed(p, m)$$

Once this more liberal goal is obtained, the predicates that were transformed to weaken the goal are to be propagated in the goal tree to replace their older version everywhere; this generally results in strengthened brother goals and weakened higher-level goals. The result of the change propagation in the tree shown in Figure 8.16 will produce a strengthened goal in the middle of the tree, namely,

Intended(p, m) $\Rightarrow \Diamond$ [Informed**InTime**(p, m) $\wedge$ Convenient(p, m)]

The deidealization procedure is similar to the one used for weakening divergent goals [Lam98b]. It is simpler here as only one goal assertion has to be considered for weakening. The procedure has two steps:

(1) *Weaken* the goal specification to obtain a more liberal version that covers the obstacle. Syntactics generalization operators can be used here such as adding a disjunct, removing a conjunct, or adding a conjunct in the antecedent of an implication.

(2) Propagate the predicate changes in the goal AND-tree in which the weakened goal is involved, by replacing every occurrence of the old predicates by the new ones.

The cardinality transformations in [Fea93] may be seen as a particular form of syntactics generalization in step 1 of this simplified procedure. Step 2 can be done simply by updating the instantiations of the goal refinement patterns used to build the goal graph, when such patterns have been used [Dar96].

Goal deidealization patterns may also be used as formal support for the deidealization process. Given the obstructed goal and the obstructing obstacle, they yield deidealized versions of the goal. To illustrate the approach, Table 8.7 gives some patterns for some of the obstacles from Table 8.1.

goal	obstacle	deidealized goal
R $\Rightarrow \Diamond$ S	$\Diamond$ [R $\wedge \neg$ P]	R $\wedge$ P $\Rightarrow \Diamond$ S
R $\Rightarrow \Diamond$ S	$\Diamond$ [R $\wedge \Box \neg$ P]	R $\wedge$ (P *W* S) $\Rightarrow \Diamond$ S
R $\Rightarrow \Diamond$ S	$\Diamond$ [R $\wedge$ ($\neg$ S *U* $\neg$ P)]	R $\wedge$ (P *W* S) $\Rightarrow \Diamond$ S
R $\Rightarrow \Diamond$ S	$\Diamond$ [R $\wedge$ ($\neg$ S *U* $\Box \neg$ P)]	R $\wedge \Box \Diamond$ P $\Rightarrow \Diamond$ S
R $\Rightarrow \Diamond$ S	$\Diamond$ [R $\wedge \Box$ ($\neg$ S *U* $\neg$ P)]	R $\wedge \Diamond$ (P *W* (P$\wedge$S)) $\Rightarrow \Diamond$ S

TABLE 8.7. Deidealization patterns for *Achieve* goals

At the end of Section 8.4.3.1 we considered the resource management *Achieve* goal

$\forall$ u: User, r: Resource
Requesting (u, r) $\Rightarrow \Diamond$ Allocated (r, u),

and generated the starvation obstacle

$\Diamond \exists$ u: User, r: Resource
Requesting (u, r) $\wedge \Box$ [$\neg$ Allocated (r, u) *U* $\exists$u' $\neq$ u: Allocated (r, u')]

The goal and starvation obstacle match the last row of Table 8.7; we thereby generate the deidealized goal specification

$\forall$ u: User, r: Resource
Requesting (u, r) $\wedge$ ($\neg \exists$u $\neq$ u: Allocated(ru')) *W* Allocated(r,u)
$\Rightarrow \Diamond$ Allocated (r, u)

The new goal version states that *if* the user requests the resource and the resource is subsequently kept unallocated unless allocated to her/it, *the*n the resource is eventually allocated to her/it. The new condition P *W* S that strengthens the antecedent has to be propagated into the goal AND-tree. The goals that refer to this new predicate as target condition might be operationalized through a reservation procedure.

8. 5. 1. 5. Domain transformation

This strategy consists in transforming the domain within which the software-to-be operates so as to make the obstruction disappear. The set of domain properties is modified so as to make the obstacle either inconsistent with the domain (see the domain-consistency condition in Section 8.2.1) or no longer obstructing the goal (see the obstruction condition in Section 8.2.1).

As an illustration of the first case, consider the goal Achieve[AllocatedAmbulanceMobilized] in an ambulance dispatching system. One obstacle to this goal corresponds to the situation where an ambulance crew decides to mobilize another ambulance than the one allocated by the system. The domain property making this possible is that mobilization orders received by crews at ambulance stations mention the incident location. The obstacle can then be eliminated by transforming the mobilization order so that it does no longer mention the incident location; the latter information would then be provided by a mobile data terminal inside the ambulance.

As an illustration of the second case, we can prevent the obstacle InconvenientLocation from obstructing the goal InformedParticipantsAttendance in the meeting scheduler system by transforming the domain so that video conferencing is made possible; the conjunct m.Location **in** p.Constraints would then be dropped from the domain property stating necessary conditions for meetings to be convenient (see Section 8.4.1).

8. 5. 2. Obstacle Reduction

The difference between tactics for reducing obstacles and the previous one is that here one tries to *reduce the occurrences of the obstacle* instead of eliminating them completely.

Tactics that act on the motivation of human agents are instances of this class. The principle is to reduce the situations in which an agent acts abnormally or irresponsibly either by dissuasion or by providing rewards. For instance, many library systems issue fines to dissuade borrowers from late returns; insurance systems provide premium reduction for good customers; some transportation companies issue rewards for crews arriving on time; and so on.

8. 5. 3. Obstacle Tolerance

In cases where the obstacle cannot be thoroughly avoided, or where avoiding it is simply too costly or not worthwhile, one may specify which behaviors will be admissible or tolerated in the presence of the obstacle.

8. 5. 3. 1. Goal restoration

The tactic restore goal consists of *adding a new goal* stating that if the obstacle condition OC becomes true then the obstructed goal assertion G should be satisfied again in some reasonably near future. This new goal thus takes the *Achieve* form

G*: $OC \Rightarrow \Diamond G$

This strategy could be followed for the obstacle PaperLost that obstructs the goal Achieve[ReviewReturned]. A subgoal refining the restoration goal will be Achieve[Lost-PaperResent].

8. 5. 3. 2. Obstacle mitigation

Another alternative tactic to obstacle elimination is to seek effective ways of mitigating the consequences of the obstacle. The principle is to *add a new goal* to attenuate the effects of obstacle occurrences. Two forms of mitigation can be distinguished.

The tactic mitigate obstacle weakly consists in ensuring some weakened version G' of the obstructed goal G whenever the obstacle condition OC becomes true. A weak mitigation goal thus has the form

G*: $OC \Rightarrow G'$

where G' is a deidealized version of G obtained using the specification transformations described in Section 8.5.1.4.

To illustrate this, consider the obstacle LastMinuteImpediment generated in Section 8.4.1. The introduction of the weak mitigation goal

Achieve [ImpedimentNotified]

will ensure a weaker version of the goal InformedParticipantsAttendance in Section 8.4.1, namely,

$Intended(p, m) \wedge Informed(p, m) \wedge Convenient(p, m)$
$\Rightarrow \Diamond [\, Participates(p, m) \vee \textit{Excused (p, m)} \,]$

(Note that in this case an obstacle prevention alternative to such weak mitigation would yield a goal like Achieve [MeetingReplanned].)

The tactic mitigate obstacle strongly consists in ensuring some parent goal G' of G whenever the obstacle condition OC becomes true, in spite of G being obstructed. A strong mitigation goal thus has the form

G*: $OC \Rightarrow G'$

where the obstructed goal G is a subgoal of G'.

Figure 8.23 illustrates this on a mine pump system example [Jos95]. The goal

Avoid[MinerInOverfloodedMine]

strongly mitigates the obstacle ExcessiveWaterFlow that obstructs the goal WaterFlow-Limited by guaranteeing that the parent goal Avoid[MinerDrowning] will be satisfied.

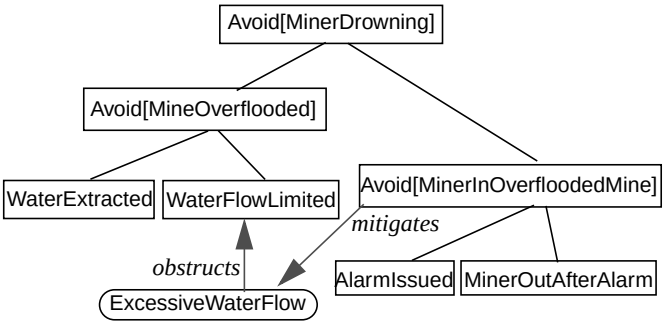


FIGURE 8.23. Obstacle mitigation

The distinction between strong and weak mitigation somewhat corresponds, at the requirements engineering level, to two different, sometimes confused notions of fault tolerance [Cri91]: one where the program meets its specification in spite of faults, and the other where the program meets a weaker version of the specification.

8. 5. 3. 3. Do-nothing

For non-critical obstacles whose consequences have no significant impact on the performance of the system a last strategy is of course to tolerate its occurrences without any resolution action.

8. 6. Summary

In order to get high-quality software, it is of upmost importance to reason about exceptional agent behavior during requirements elaboration --not only software agents, but also the agents in the environment like devices, operators, users, etc.

The key principle underlying this chapter is that obstacle analysis needs to be done as early as possible in the requirements engineering process, that is, at the *goal* level. The earlier such analysis is started, the more freedom is left for resolving the obstacles. Moreover, goals provide a precise entry point for starting analysis in a more focussed way like, e.g., the construction of fault-trees or threat-trees from negated goals.

Various formal and heuristic techniques were presented for obstacle generation and refinement from goal specifications and domain properties; the generation of obstacle resolutions is achieved through various tactics to eliminate, reduce, or tolerate the obstacle.

When to apply such or such identification/resolution technique may depend on the domain, on the application in this domain, on the kind of obstacle, on the severity of its consequences, on the likelihood of its occurrence, and on the cost of its resolution. Much exciting work remains to be done with those respects.

Chapter 9
Case Studies

This chapter illustrates and assesses the various techniques described in the preceding chapters on real case studies of significant sizes. Two case studies have been performed.

The first one concerns the ambulance despatching system of the London Ambulance Service (LAS) for which failure stories have been published [LAS93, Fin96]. The LAS system was proposed as a common case-study for the 8th International Workshop on Software Specification and Design (IWSSD-8). The initial document for this case study is the "Report on the Inquiry into the London Ambulance Service" [LAS93].

The second one concerns an automated train control system for the San Francisco Bay Area Rapid Transit (BART) system. The BART system is a recent benchmark proposed to the formal specification community. The initial document was provided by an independent source involved in the development [Win99].

These case studies are appealing for a number of reasons; they are real, safety-critical, and distributed systems involving real-time, accuracy, and fault tolerance requirements.

For both case studies, the agent-driven tactics of Chapter 6 are used to generate alternative goal refinements, agent responsibilities and agent interfaces from high-level goals. Obstacle Analysis is then performed on the resulting goal models. Various obstacle identification techniques are used to systematically generate obstacles from goals. Alternative obstacle resolutions are then explored through the systematic application of obstacles resolution tactics. For the LAS case study, the list of potential obstacles that have been generated are compared to the scenarios that actually occurred during the two system failures in October-November 1992.

9. 1. The London Ambulance Service System

9. 1. 1. Introduction

The London Ambulance Service (LAS) has two main functions: responding to emergency calls requiring the rapid intervention of an ambulance, and dealing with non-urgent patient journeys. The case study is only concerned with the handling of urgent calls.

The UK Government imposes performance standards (called ORCON) for accident and emergency calls upon ambulance services. In 1992, when the first automated system for LAS was put into use, the performance standard for the LAS was:

> *An ambulance must arrive at the scene within 14 minutes for 95% of the calls.*

It was the difficulty of meeting that standard that motivated the need for a new system.

Our elaboration of the goal model for the LAS is mostly based on a section of the Inquiry Report that describes the rationale for a Computer Aided Despatch (CAD) system. Excerpts from that section are reproduced below:

> In order to understand the rationale behind the development of the CAD system it is essential to understand the manual system that it would replace and its shortcomings.
>
> The Manual system operates as follows:
>
> *Call Taking*
>
> When a 999 or urgent call is received in the Central Ambulance Control (CAC) room, the Control Assistant (CA) writes down the call details on a pre-printed form. The incident location is identified from a map book, together with the map reference co-ordinates. On completion of the call the incident form is placed into a conveyor belt system with other forms from fellow CA's. The conveyor belt then transports the forms to a central collection point within CAC.
>
> *Resource Identification*
>
> Another CAC staff member collects the form from the central collection point and, through reviewing the details on the form, decides which resource allocator should deal with it (based on the three London Division - North East, North West, and South). At this point, potential duplicate calls are also identified. The resource allocator then examines the form for his/her sector and, using status and location information provided through the radio operator and noted on forms maintained in the "activation box" for each vehicle, decides which resource should be mobilized. This resource is then also recorded on the form which is passed to a despatcher.
>
> *Resource Mobilisation*
>
> The despatcher will telephone the relevant ambulance station (if that is where the resource is) or will pass mobilisation instructions to the radio operator if the ambulance is already away from the station.
>
> According to the ORCON standards this whole process should take no more than 3 minutes.
>
> There are some clear deficiencies with a totally automated manual system including:
>
> a) identification of the precise location can be time consuming due to often incomplete or inaccurate details from the caller and the consequent need to explore a number of alternatives through the map books;
>
> b) the physical movement of paper forms around the Control Room is inefficient;
>
> c) maintaining up to date vehicle status and location from allocators' intuition and reports from ambulances as relayed to and through the radio operators is a slow and laborious process;

d) communicating with ambulances via voice is time consuming and, at peak times, can lead to mobilization queues;

e) identifying duplicate calls relies on human judgement and memory. This is error prone;

f) dealing with call backs is a labour intensive process as it often involves CA's leaving their posts to talk to the allocators;

g) identification of special incidents needing a Rapid Response Unit or the helicopter (or a major incident team) relies totally on human judgement.

A computer aided despatch system is intended to overcome most of these deficiencies through such features as:

a) a computer based gazetteer with public telephone box identification;

b) elimination of the need to move paper around the control room;

c) timely and (in the case of location information) automated update of resource availability information;

d) computer based intelligence to help identify duplicates and major incidents;

e) direct mobilization to the ambulance on the completion of the call thus potentially, in simple cases, achieving mobilization inside one minute.

9. 1. 2. Elaborating the Goal Model

9. 1. 2. 1. Identifying preliminary goals

1. Identifying high-level performance goal

Form the ORCON standards, we extract the first-sketch idealized goal Achieve[AmbulanceIntervention] that requires that an ambulance must arrive at the scene of an incident within 14 minutes *for every call*.

Goal Achieve[AmbulanceIntervention]
InformalDef For every urgent call reporting an incident, there should be an ambulance at the scene of the incident within 14 minutes.
FormalDef $\forall$ c: UrgentCall, inc: Incident
@ Reporting(c, inc) $\Rightarrow \Diamond_{\leq 14'}$ ($\exists$ amb: Ambulance) Intervention(amb, inc)

This goal is idealized. Sometimes, an ambulance will not arrive on time. During obstacle analysis, we will identify various obstacles to the satisfaction of this goal; and alternative strategies for preventing, reducing or mitigating the consequences of these obstacles.

The fact that 95% of the calls must be responded to within 14 minutes defines the degree to which this goal has to be satisfied. This degree of satisfaction is to be used to evaluate alternative responsibility assignments and obstacle resolution strategies. This evaluation step is currently not supported by the KAOS method.

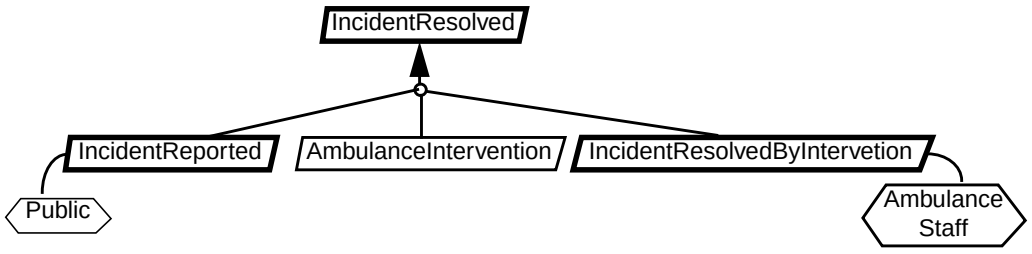


FIGURE 9.24. Asking WHY AmbulanceIntervention and uncovering assumptions

2. Asking WHY questions and uncovering assumptions

Asking a WHY question about the goal Achieve[AmbulanceIntervention] drives the identification of the higher-level goal Achieve[IncidentResolved]:

Goal Achieve[IncidentResolved]
InformalDef Every incident requiring emergency service is eventually resolved.
FormalDef ∀ inc: Incident
inc.Happened ⇒ ◊ inc.Resolved

At this point, we remain voluntarily ambiguous about what is meant for an incident to be resolved. We should however point out that 'resolved' here does not mean that no human life is lost. It is a predicate that we assume to be given by government standards defining the required services to be provided by emergency services such as the LAS.

The identification of this higher-level goal drives the identification of assumptions shown in Figure 9.24. In this goal-graph, the assumptions Achieve[IncidentReported] and Achieve[IncidentResolvedByIntervention] are elicited formally by matching a milestone-driven refinement pattern to the formalization of the parent goal Achieve[IncidentResolved], and to the formalization of the initial goal Achieve[AmbulanceIntervention]. These new goals are defined as follows:

Assumption Achieve[IncidentReported]
InformalDef Every incident requiring emergency service is eventually reported to the LAS.
FormalDef ∀ inc: Incident
inc.Happened ⇒ ◊ (∃ c: UrgentCall) Reporting(c, inc)

Assumption Achieve[IncidentResolvedByIntervention]
InformalDef An incident is resolved by the intervention of a single ambulance.
FormalDef ∀ amb: Ambulance, inc: Incident
Intervention(amb, inc) ⇒ ◊ inc.Resolved

The assumption that an incident is resolved by a single ambulance is of course highly idealized; some incidents may require the intervention of several ambulances. This assumption is idealized also because the resolution of incidents depends on the medical resources carried in the ambulances, and on the availability of medical resources at hospitals where patients are transported. Possible violations of this assumption will be systematically identified during obstacle analysis.

In the sequel, we begin by developing a requirement model that is based on this idealized assumption, i.e. we assume that an incident is resolved by the intervention of a single ambulance, and we are not concerned by the materials that need to be carried in ambu-

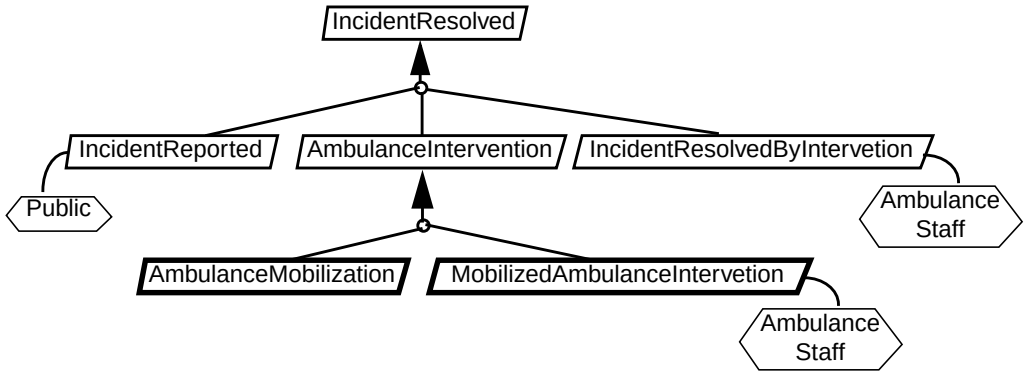


FIGURE 9.25. Refining the goal Achieve[AmbulanceIntervention]

lances. During obstacle analysis, we will identify different obstacles to the satisfaction of that assumption, and show how the idealized model can be transformed by weakening that assumption, modifying previously identified goals, and identifying further goals.

Note that with a goal-oriented approach, such simplifying assumptions have to be described explicitly in the model, and can be systematically identified by checking the completeness of goal refinements.

3. Refining the goal Achieve[AmbulanceIntervention]

The Inquiry Report describes a further performance standard that requires that for every reported incident an ambulance must be mobilized for that incident within 3 minutes. The following first-sketch goal definition is thereby identified:

Goal Achieve[AmbulanceMobilization]
InformalDef For every reported incident, there should eventually be an ambulance mobilized for that incident; the mobilization delay should be less than 3 minutes.
FormalDef $\forall$ c: UrgentCall, inc: Incident
@ Reporting(c, inc) $\Rightarrow \Diamond_{\leq 3'}$ ($\exists$ amb: Ambulance) Mobilization(amb, inc)

In Figure 9.25, this goal is declared as a subgoal of the above goal Achieve[AmbulanceIntervention]. In this goal refinement, the companion subgoal Achieve[MobilizedAmbulanceIntervention] is elicited formally by matching a milestone-driven refinement pattern to the formal definitions of the previously identified goals. The generated formal definition for the goal Achieve[MobilizedAmbulanceIntervention] is given by:

Mobilization(amb, inc) $\Rightarrow \Diamond_{\leq 11'}$ Intervention(amb, inc).

Note that this goal refinement corresponds to an application of the tactic split lack of monitorability with milestone that resolves lack of monitorability of AmbulanceStaff agents for urgent calls.

Figure 9.25 also shows that the goal Achieve[MobilizedAmbulanceIntervention] is assigned as the responsibility of the AmbulanceStaff agents. However, this goal is not realizable by AmbulanceStaff agents, because they cannot monitor for which incident they are mobilized, they can only know for which *location* they are mobilized. As a first step to solving this problem, we declare the relationships Mobilization and Intervention as derived relationships that are defined as follows:

Mobilization(amb, inc) $\Leftrightarrow$ amb.Mobilized $\wedge$ amb.Destination = inc.Location

Intervention(amb, inc) $\Leftrightarrow$ amb.Intervention $\wedge$ amb.Location = inc.Location

That is, an ambulance is mobilized for an incident iff it is mobilized and its destination is the location of the incident; and an ambulance makes an intervention for an incident iff it is in intervention mode and its location is the location of the incident. The attributes Mobilized, Destination, and Intervention of the Ambulance entity are controlled by AmbulanceStaff agents. Using these definitions, the goal

Achieve[MobilizedAmbulanceIntervention]

can now be rewritten into:

$\forall$ amb: Ambulance, loc: Location
amb.Mobilized $\wedge$ amb.Destination = loc
$\Rightarrow \Diamond_{\leq 11'}$ amb.Intervention $\wedge$ amb.Location = loc

Note that this goal definition does not refer to the Incident entity any more. This goal definition is now realizable by AmbulanceStaff agents, because they monitor the attributes Mobilized and Destination of ambulances, and control the attributes Intervention and Location of ambulances.

However, the goal definition is too strong to be satisfiable in the domain. If the ambulance is too far from the incident for which it is allocated, it may be physically impossible for ambulance crews to achieve the goal in time. Also, the goal is not satisfiable if the ambulance is not available at the time it becomes mobilized. The availability of an ambulance is defined as follows:

amb.Available $\Leftrightarrow$ amb.InService $\wedge \neg$ amb.Mobilized

Therefore, the goal is weakened so that it must be satisfied only if the ambulance is available at the time of mobilization, and if the distance between the current location of the ambulance and its destination is small enough so that it can be covered in less than 11 minutes. By applying the tactic weaken goal with unsatisfiability condition, we obtain the following goal definition:

Goal Achieve[MobilizedAmbulanceIntervention]
InformalDef An ambulance mobilized for an incident location, and able to arrive at the incident scene within 11 minutes, should be at the location of the incident within 11 minutes.
FormalDef $\forall$ amb: Ambulance, loc: Location
amb.Mobilized $\wedge$ amb.Destination = loc
$\wedge$ ● amb.Available
$\wedge$ ● TimeDist(amb.Location, amb.Destination) $\leq$ 11'
$\Rightarrow \Diamond_{\leq 11'}$ amb.Intervention $\wedge$ amb.Location = loc

In that formula, the function

TimeDist: Location $\times$ Location $\rightarrow$ TimeUnit

denotes an estimation of the time needed to go from one location to another. We assume this function to be given as part of the domain knowledge. (Maybe that function has been negotiated by representatives of the LAS and the ambulance crews.) This function is here assumed to be time-independent. In the actual system, it is likely that this function will be time-dependent to take into account traffic conditions at different moments of the day.

As a result of weakening the goal Achieve[MobilizedAmbulanceIntervention], one must now strengthen the goal Achieve[AmbulanceMobilization] so that the parent goal in Figure 9.25, Achieve[AmbulanceIntervention], is still satisfied. The strengthening of the goal

is formally derived by applying the same milestone-driven refinement pattern instantiated with a different milestone obtained from the antecedent of the goal Achieve[MobilizedAmbulanceIntervention].

After further minor simplification of the generated goal definition, we obtain the following definition:

Goal Achieve[AmbulanceMobilization]
InformalDef For every urgent call reporting an incident, an available ambulance able to arrive at the incident scene within 11 minutes should be mobilized. The ambulance mobilization time should be less that 3 minutes.
FormalDef $\forall$ c: UrgentCall, inc: Incident
@ Reporting(c, inc)
$\Rightarrow \Diamond_{\leq 3'}$ ($\exists$ amb: Ambulance)
amb.Mobilized $\wedge$ amb.Destination = inc.Location
$\wedge$ $\bullet$ amb.Available
$\wedge$ $\bullet$TimeDist(amb.Location, inc.Location) $\leq$ 11'

It is this goal that will be further refined in the following sections.

In order to have a complete refinement of the goal Achieve[AmbulanceIntervention], the following assumption was also identified:

inc.Reported $\wedge$ inc.Location = loc $\Rightarrow$ $\Box$ inc.Location = loc

That is, it is assumed that the location of an incident does not change. The validity of this assumption needs to be checked with domain experts. (This assumption may not be valid if someone uses a mobile phone to report an incident that happened in a train for instance.)

4. Deriving the object model

As described in Chapter 3, objects are gradually derived from the formal definition of goals. Figure 9.26 shows the object model derived from the definition of the goals identified so far. This model will be enriched with further objects and attributes during the goal refinement process.

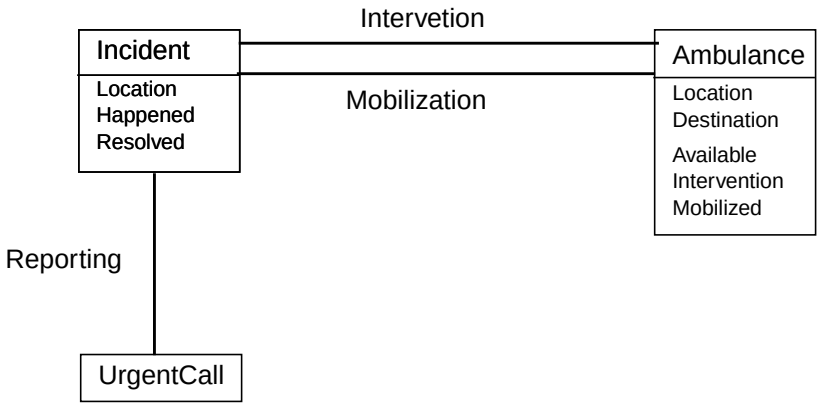


FIGURE 9.26. Partial object model derived from the goals in Figure 9.25

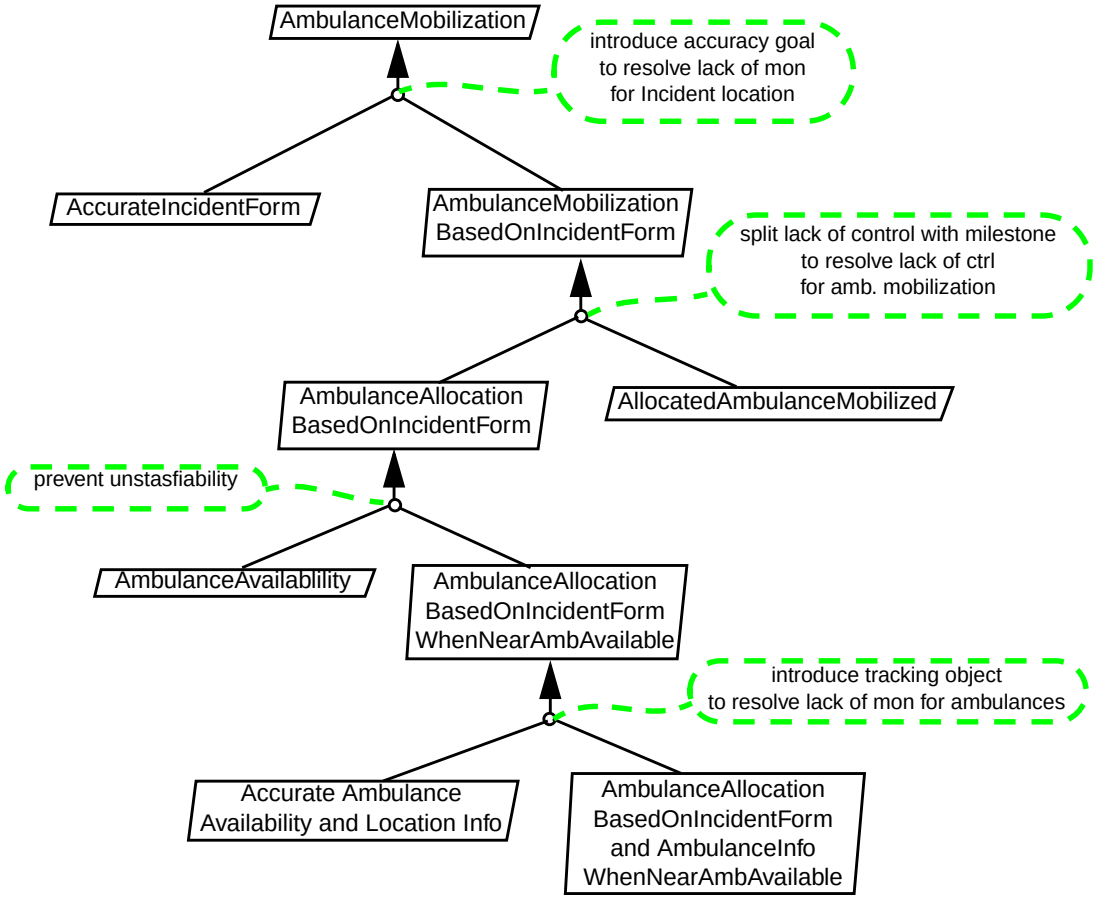


FIGURE 9.27. Applying agent-driven tactics to recursively refine the goal Achieve[AmbulanceMobilization]

9. 1. 2. 2. Refining the goal Achieve[AmbulanceMobilization]

We now consider the refinement of the goal Achieve[AmbulanceMobilization]. This goal is unrealizable by LAS agents operating in the Central Ambulance Control (CAC) room. In the sequel, we call such agents CACAgent. Specializations of CACAgent are, among others, the ControlAssistant, the ResourceAllocator, and the Computer Aided Despatch software. Ambulance staff are not CAC agents.

The goal Achieve[AmbulanceMobilization] is unrealizable by CAC agents for several reasons:

- **lack of monitorability:** CAC agents lack of monitorability for the attributes Incident.Location and Ambulance.Location;
- **lack of control:** CAC agents lack of control for the attributes Ambulance.Mobilzed and Ambulance.Destination (these attributes are controlled by AmbulanceStaff agents);
- **unsatisfiability:** the goal is unsatisfiable if there is no available ambulance that is able to reach the incident location within 11 minutes. That is, the domain of unsatisfiability for the goal is given by:
  $\Diamond$ ($\exists$ inc: Incident, c: UrgentCall):
  (@Reporting(c, inc) $\wedge$ $\square_{\leq 3'}$ $\neg$ ($\exists$ amb: Ambulance): (amb.Available
  $\wedge$ TimeDist(amb.Location, inc.Location) $\leq$ 11'))

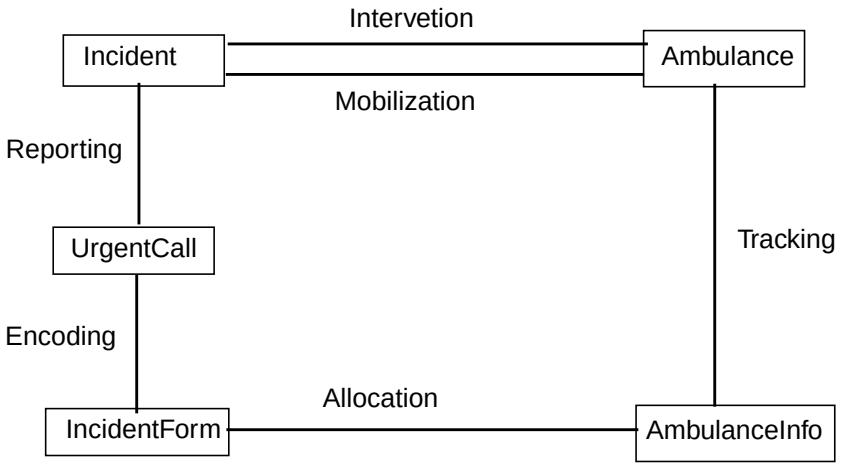


FIGURE 9.28. Partial object model derived from the goals in Figures 9.25 and 9.27 (attributes of objects are not shown)

Figure 9.27 shows a portion of the refinement graph that is obtained by recursively applying agent-driven tactics so as to resolve these realizability problems. Figure 9.28 shows the portion of the object model derived from the goals in Figure 9.25 and 9.27.

Each goal refinement step is now described in turn.

1. Resolve lack of monitorability for incident location

We first resolve lack of monitorability for incident location. The tactic introduce tracking object is used to generate the subgoals:

Achieve[AccurateIncidentForm]

Achieve[AmbulanceMobilizationBasedOnIncidentForm].

(These goals are formally defined below.)

The object model is enriched with the new object IncidentForm that is used to record details about incidents. In the manual system, such an object corresponds to a paper form recording incident details noted by the Control Assistant (CA) agents who handle emergency calls. In an automated system, this object corresponds to an electronic version of the paper form.

The agent model is elaborated by declaring the Control Assistant (CA) agent, and assigning the goal Achieve[AccurateIncidentForm] to that agent. This is a simplification. Actually, the goal Achieve[AccurateIncidentForm] needs to be further refined into subgoals whose responsibilities will be assigned to the Public agent, the CA agents, and possibly software agents such as a Map Gazetteer used to help in locating incidents. In this case study, we will not be concerned with further refinement of this goal. Note however that failures to meet this goal - among others, due to the slow performance of the user interface - contributed to the disaster at the LAS.

The two goals generated by the tactic are formally defined as follows:

Goal Achieve[AccurateIncidentForm]
InformalDef For every urgent call reporting an incident, there is an incident form recording details about the incident. The incident form should record the accurate location of the incident and the time at which the call was taken. (Further details about the incident such as the number of injured persons and the kind of emergency services needed are ignored for the moment. We will need to include them later when the model will be deidealized.)
The time needed to handle the call and fill the incident form should take no more than "call_taking_delay" time units.
FormalDef $\forall$ c: UrgentCall, inc: Incident
Reporting(c, inc) $\Rightarrow \Diamond_{\leq call\_taking\_delay}$ ($\exists$ if: IncidentForm): if.Encoded $\wedge$ Encoding(if, c)
$\wedge$
$\forall$ inc: Incident, c: UrgentCall, if: IncidentForm
Reporting(c, inc) $\wedge$ Encoding(if, c) $\Rightarrow$ if.Location = inc.Location $\wedge$ if.CallTime = c.Time

Goal Achieve[AmbulanceMobilizationBasedOnIncidentForm]
InformalDef For every incident form, an ambulance able to arrive at the incident scene within 11 minutes should be mobilized to the corresponding location. An ambulance should be mobilized less that 3 minutes after the reception of the call.
(Note that if, due to duplicate calls, different incident forms refer to the same location, a single ambulance can be mobilized in response to these different incident forms.)
FormalDef $\forall$ c: UrgentCall, if: IncidentForm
@ if.Encoded
$\Rightarrow \Diamond_{\leq if.CallTime+3'}$ ($\exists$ amb: Ambulance)
amb.Mobilized $\wedge$ amb.Destination = if.Location
$\wedge$ $\bullet$ amb.Available $\wedge$ $\bullet$TimeDist(amb.Location, if.Location) $\leq$ 11'

2. Resolve lack of control for ambulance mobilization

The goal Achieve[AmbulanceMobilizationBasedOnIncidentForm] is not realizable by CAC agents because they lack control of the mobilization of ambulances. (The mobilization of ambulances is controlled by ambulance staff and not directly by CAC agents.)

The tactic split lack of control with milestone is then used to generate the subgoals:

Achieve[AmbulanceAllocationBasedOnIncidentForm]

Achieve[AllocatedAmbulanceMobilized].

(The formal definitions of these goals are shown below.)

The first goal requires an ambulance to be *allocated* to the incident location; the second goal requires the allocated ambulance to eventually be effectively mobilized.

Note that CAC agents cannot allocate ambulances by referring directly to the ambulance instances (this is what Michael Jackson calls an 'identity concern' [Jac2k]); they can only allocate ambulances by referring to some domain-level identifier such as their license plate number, or some other identifier.

We therefore introduce for the Ambulance entity, an attribute, AmbID, that is used by agents in the domain to uniquely refer to a particular ambulance instance. We also introduce the new entity AmbulanceInfo with an attribute AmbId, and a Tracking relationship relating Ambulance and AmbulanceInfo. This relationship is defined as follows:

Tracking(ai, amb) $\Leftrightarrow$ ai.AmbId = amb.AmbID

The Allocation relationship is then defined as a relationship between IncidentForm and AmbulanceInfo (as opposed to the relationship Mobilization and Intervention that link Incident and Ambulance). This relationship is defined in terms of the new attributes Allocated and All_Dest of the AmbulanceInfo entity:

Allocation(ai, if) $\Leftrightarrow$ ai.Allocated $\wedge$ ai.All_Dest = if.Location

At this point, alternative refinement of the goal

Achieve[AmbulanceMobilizationBasedOnIncidentForm]

can be envisaged. These alternatives are generated by the application of alternative milestone-driven refinement patterns in Table 6.2. The two subgoals mentioned above are generated by applying the third pattern in that table. The subgoal

Achieve[AllocatedAmbulanceMobilized]

requires the mobilized ambulance to be exactly the one that has been allocated. By applying the alternative fourth pattern in that table, one generates the alternative refinement:

Achieve[AmbulanceAllocationBasedOnIncidentForm]

Achieve[AmbulanceMobilizedForAllocation]

The first subgoal is the same as the one generated by the first pattern. The second subgoal in this refinement allows ambulance staff to mobilize another ambulance than the one allocated by the system provided that the mobilized and allocated ambulances are at the same location -for instance, at the same ambulance station. This second alternative was actually the one that was in use before the automated system was introduced. When an ambulance waiting in an ambulance station was allocated, ambulance staff were allowed to take some other ambulance from that station. With the introduction of the automated system, ambulance staff were required to abandon this practice. This contributed significantly to the disaster at the LAS since ambulance staff would not or even could not obey this stronger requirement. This problem will be further discussed during obstacle analysis.

In the first alternative, the goals generated by the application of the formal refinement pattern are after some further simplification defined as follows:

Goal Achieve[AmbulanceAllocationBasedOnIncidentForm]
InformalDef For every incident form, an available ambulance able to arrive at the incident scene within 11 minutes should be allocated to the corresponding location. The ambulance allocation time should take no more than “allocation_delay” time units.
FormalDef $\forall$ c: UrgentCall, if: IncidentForm
@ if.Encoded
$\Rightarrow \Diamond_{\leq allocation\_delay}$ ($\exists$ ai: AmbulanceInfo, amb: Ambulance):
ai.Allocated $\wedge$ ai.AllocationDest = if.Location
$\wedge$ ai.AmbID = amb.AmbID
$\wedge$ ● amb.Available $\wedge$ ● $\neg$ ai.Allocated
$\wedge$ ●TimeDist(amb.Location, if.Location) $\leq$ 11’

Goal Achieve[AllocatedAmbulanceMobilized]
InformalDef When an ambulance is allocated to an incident location, it should eventually be mobilized to that location. This should take no more than “mob_communication_delay” time units.
FormalDef $\forall$ ai: AmbulanceInfo, amb: Ambulance, loc: Location
@ ai.Allocated $\wedge$ ai.AllocationDest = loc
$\wedge$ ai.AmbID = amb.AmbID $\wedge$ ● amb.Available
$\Rightarrow \Diamond_{\leq mob\_communication\_delay}$
amb.Mobilized $\wedge$ amb.Destination = loc

The real-time delays in the above definitions must be defined such that:

allocation_delay + mob_communication_delay $\leq$ 3’

The validity of the goal refinement is also dependent on the satisfaction of the following goals that were formally identified when refining the goal:

Goal Avoid[LocationChangeOnIncidentForm]
Definition The location on an incident form should not change.
FormalDef $\forall$ if: IncidentForm, loc: Location
if.Encoded $\wedge$ if.Location = loc $\Rightarrow$ ❑ if.Location = loc

Goal Maintain[AllocatedAmbulanceInServiceUntilMobilized]
Definition An allocated ambulance should remain in service until it is effectively mobilized. (This goal could be violated if an allocated ambulance stops its shift before it receives the mobilization order.)
FormalDef $\forall$ ai: AmbulanceInfo, amb: Ambulance, loc: Location
ai.Allocated $\wedge$ amb.InService $\wedge$ Tracking(ai, amb)
$\Rightarrow$ amb.InService *W* amb.Mobilized

Goal Maintain[AllocatedAmbulanceNearToIncident]
Definition An allocated ambulance should remain close to the incident until it is effectively mobilized. (This goal could be violated if an allocated ambulance gets away from the incident location before it receives the mobilization order.)
FormalDef $\forall$ ai: AmbulanceInfo, amb: Ambulance
ai.Allocated $\wedge$ Tracking(ai, amb) $\wedge$ TimeDist(amb.Location, loc) $\leq$ 11’
$\Rightarrow$ TimeDist(amb.Location, loc) $\leq$ 11’ *W* amb.Mobilized

These last two goals are quite idealized. The model will therefore need to be transformed so as to weaken these goals. In order to keep our description of the elaboration process within bounds, we will not describe such transformations.

3. Resolve unsatisfiability

The goal Achieve[AmbulanceAllocationBasedOnIncidentForm] still suffers from the unsatisfiability problem inherited from his parent goals. Since the resolution of incident is safety-critical, the tactic prevent unsatisfiability is used to resolve that unsatisfiability problem, thereby generating the two subgoals:

Maintain[AmbulanceAvailability]

Achieve[AmbulanceAllocationBasedOnIncidentForm WhenNearAmbAvailable]

These goals are defined as follows.

Goal Maintain[AmbulanceAvailability]
InformalDef For every location, there should always be an available ambulance able to arrive at that location within 11 minutes.
FormalDef $(\forall$ loc: Location): $\square$ $(\exists$ amb: Ambulance, ai: AmbulanceInfo):
amb.Available $\wedge$ $\neg$ ai.Allocated $\wedge$ Tracking(ai, amb) $\wedge$ TimeDist(amb.Location, loc) $\leq$ 11'

Goal Achieve[AmbulanceAllocationBasedOnIncidentForm WhenNearAmbAvailable]
InformalDef For every incident form, an available ambulance able to arrive at the incident scene within 11 minutes should be allocated to the corresponding location *except if there is no such ambulance available*. The ambulance allocation time should take no more than "allocation_delay" time units.
FormalDef $\forall$ c: UrgentCall, if: IncidentForm
@ if.Encoded
$\Rightarrow \Diamond_{\leq allocation\_delay}$ $(\exists$ ai: AmbulanceInfo, amb: Ambulance):
(ai.Allocated $\wedge$ ai.AllocationDest = if.Location $\wedge$ ai.AmbID = amb.AmbID
$\wedge$ $\bullet$ amb.Available $\wedge$ $\bullet$ $\neg$ ai.Allocated
$\wedge$ $\bullet$TimeDist(amb.Location, if.Location) $\leq$ 11')
$\vee$ $\neg$ $(\exists$ amb: Ambulance, ai: AmbulanceInfo):
(amb.Available $\wedge$ $\bullet$ $\neg$ ai.Allocated $\wedge$ Tracking(ai, amb)
$\wedge$ TimeDist(amb.Location, if.Location) $\leq$ 11')

4. resolve lack of monitorability for ambulance availability and location

The goal

Achieve[AmbulanceAllocationBasedOnIncidentForm WhenNearAmbAvailable]

is still unsatisfiable by CAC agents, because they lack of monitorability for the actual locations and availability of ambulances. The tactic introduce tracking object is then used to generate the subgoals:

Maintain[AmbulanceTracked]

Maintain[AccurateAmbulanceAvailabilityandLocationInfo],

Achieve[AmbulanceAllocatationBasedOnIncidentFormandAmbulanceInfo
WhenNearAmbAvailable].

These goals are defined as follows:

Goal Maintain[AmbulanceTracked]
InformalDef Every ambulance is tracked by exactly one AmbulanceInfo object.
FormalDef ($\forall$ amb: Ambulance, $\exists!$ ai: AmbulanceInfo): Tracking(ai, amb)
$\wedge$ $\forall$ amb: Ambulance, $\forall!$ ai: AmbulanceInfo
Tracking(ai, amb) $\Rightarrow$ $\Box$ Tracking(ai, amb)

Goal Maintain[AccurateAmbulanceAvailabilityandLocationInfo]
InformalDef Informations about ambulances' availability and location should be accurate
FormalDef $\forall$ amb: Ambulance, ai: AmbulanceInfo
Tracking(ai, amb)
$\Rightarrow$
ai.Available $\leftrightarrow$ amb.Available
$\wedge$ ai.Location = amb.Location

Goal Achieve[AmbulanceAllocatationBasedOnIncidentFormandAmbulanceInfo WhenNearAmbAvailable]
InformalDef For every incident forms, and based on ambulance information (status and location), an available ambulance able to arrive at the incident scene within 11 minutes should be allocated to the corresponding location except if there is no such ambulance available. The ambulance allocation time should take no more than "allocation_delay" time units.
FormalDef $\forall$ c: UrgentCall, if: IncidentForm
@ if.Encoded
$\Rightarrow \Diamond_{\leq allocation\_delay}$ ($\exists$ ai: AmbulanceInfo):
(ai.Allocated $\wedge$ ai.AllocationDest = if.Location
$\wedge$ $\bullet$ ai.Available $\wedge$ $\bullet$ $\neg$ ai.Allocated
$\wedge$ $\bullet$TimeDist(ai.Location, if.Location) $\leq$ 11')
$\vee$ $\neg$ ($\exists$ ai: Ambulance): (ai.Available $\wedge$ $\bullet$ $\neg$ ai.Allocated
$\wedge$ TimeDist(ai.Location, if.Location) $\leq$ 11')

5. Handling ambulance mobilization when no near ambulances are available

The above refinement graph relies on the goal Maintain[AmbulanceAvailability] that was introduces to resolve the unsatisfiability problem. This goal can be further refined and operationalized by moving ambulances appropriately so as to ensure the availability of near ambulances for every location. It may for instances involve the positioning of a sufficient number of ambulances nearby locations where incidents are likely to happen. We will not be concerned here with the refinement of this goal.

Nevertheless, this goal could still be violated from time to time. The handling of such violations is part of the obstacle analysis loop. We feel however it is the right time to describe how such violation is tolerated. The goal Maintain[AmbulanceAvailability] is obstructed by the obstacle NoNearAmbulanceAvailble. This obstacle is resolved by applying the tactic mitigate obstacle, thereby generating the new goal

Achieve[AmbulanceMobilizationBasedOnIncidentForm WhenNoNearAmbAvailable].

The precise definition of this goal has to be elicited from domain experts. This goal can the be further refined by applying the same tactics as those used to refine the goal

Achieve[AmbulanceMobilizationBasedOnIncidentForm WhenNearAmbAvailable]

in Figure 9.27, yielding a similar goal refinement graph.

6. Alternative Responsibility Assignments for ambulance allocation

At this stage, we identify alternative responsibility assignment for the goal

Achieve[AmbulanceAllocatationBasedOnIncidentFormandAmbulanceInfo WhenNearAmbAvailable].

concerned with the allocation of ambulances based on incident forms and ambulance informations.

In the manual system, responsibility for this goal is split among several ResourceAllocator agent based on the division of London (North East, North West or South) in which the incident is reported to have occurred. The goal is therefore refined into the subgoals:

Achieve[IncidentFormAssignedToRA]

Achieve[AllocationForAssignedIncidentFormBasedOnAmbulanceInfo]

The first goal requires the incident form to be handed to the resource allocator of the division in which the incident occurred. This goal is assigned as the responsibility of a human agent collecting incident forms from ControlAssistant's at a central collection point. The second goal is assigned as the responsibility of the ResourceAllocator to which the incident form is assigned. Note that there is an obvious reason for dividing responsibility for incident forms based on the division of london in which it occurred; a human ResourceAllocator agent would not be capable of managing alone all the incidents and ambulances of London.

Such restriction does not apply to a CAD software agent. In a fully automated system, the leaf goal concerned with the allocation of ambulances would be directly assigned as the responsibility of the CAD agent. This seems to be more or less the design chosen for the 1992 automated system. (The Inquiry Report mentions that only in the most complex cases would a human allocator need to identify and allocate the best resource.)

Of course, many other alternatives, involving the cooperation of human agents and the CAD software can be envisaged. For instances, the CAD agent could be responsible for proposing the more appropriate ambulance to the Resource Allocator (or to the Control Assistant) who would then accept the proposition or select another ambulance. As another alternative, the CAD software could be responsible for displaying incidents and ambulances on a map on the screen of the Resource Allocator, who would use that information to allocate the most appropriate ambulance.

We are here crossing the boundary between requirements engineering and human-computer interaction (HCI). Although in principle the concepts and techniques of goal-oriented requirements engineering could be used to generate alternative designs at this level of details, it is certainly not the most appropriate method to do so. The result of our goal-oriented requirement elaboration process could be used as the starting point for HCI design. The links between goal-oriented RE and HCI would be worth studying further.

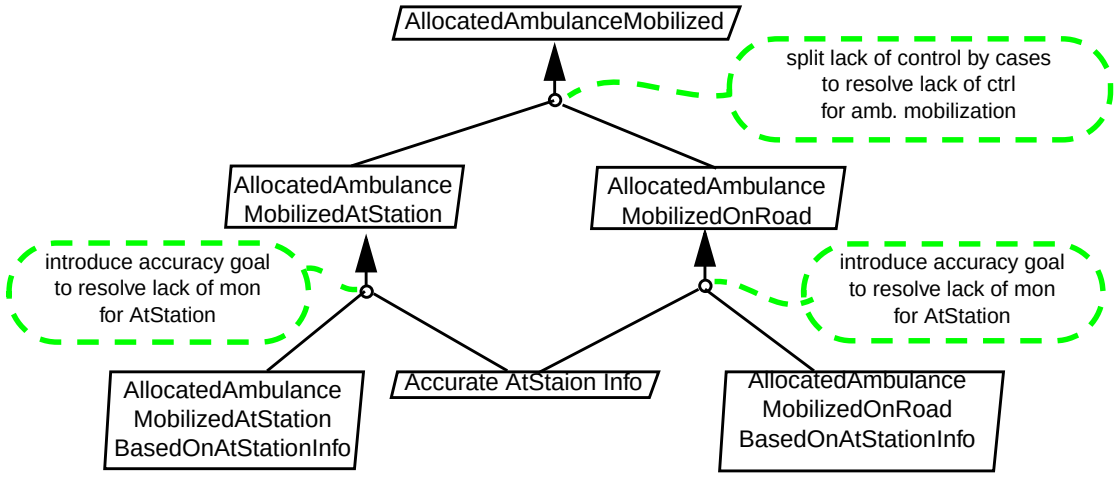


FIGURE 9.29. Refining the goal Achieve[AllocatedAmbulanceMobilized]

9. 1. 2. 3. Refining the goal Achieve[AllocatedAmbulanceMobilized]

We now come back in Figure 9.27 to the previously identified goal

Achieve[AllocatedAmbulanceMobilized]

In the definition of this goal, the attributes Ambulance.Mobilized and Ambulance.Destination are controlled by AmbulanceStaff agents; and the attributes AmulanceInfo.Allocated and AmulanceInfo.All_Dest controlled by CAC agents are not directly monitored by AmbulanceStaff agents.

The goal is not realizable by CAC agent because they lack of control for the actual mobilization of ambulances. Figure 9.29 shows a portion of the refinement graph for this goal that is obtained by recursively applying agent-driven tactics.

1. Split lack of control by cases

Allocated ambulances have to be mobilized differently according to whether the ambulance is waiting at a station or is somewhere on the roads. The tactic split lack of control by cases is therefore used to generate the two goals:

Achieve[AllocatedAmbulanceMobilizedAtStation]

Achieve[AllocatedAmmbulanceMobilizedOnRoad]

These goals are formally defined as follows:

Goal Achieve[AllocatedAmbulanceMobilizedAtStation]
InformalDef When an ambulance waiting at an ambulance station is allocated to an incident location, it should eventually be mobilized to that location. This should take no more than “mob_communication_delay” time units
FormalDef $\forall$ ai: AmbulanceInfo, amb: Ambulance, loc: Location
ai.Allocated $\wedge$ ai.AllocationDest = loc
$\wedge$ ai.AmbID = amb.AmbID $\wedge$ ● amb.Available
$\wedge$ ($\exists$ st: AmbulanceStation): AtStation(amb, st)
$\Rightarrow \Diamond_{\leq mob\_communication\_delay}$
amb.Mobilized $\wedge$ amb.Destination = loc

Goal Achieve[AllocatedAmmbulanceMobilizedOnRoad]
InformalDef When an ambulance already in the road is allocated to an incident location, it should eventually be mobilized to that location. This should take no more than "mob_communication_delay" time units
FormalDef $\forall$ ai: AmbulanceInfo, amb: Ambulance, loc: Location
ai.Allocated $\wedge$ ai.AllocationDest = loc
$\wedge$ ai.AmbID = amb.AmbID $\wedge$ $\bullet$ amb.Available
$\wedge \neg$ ($\exists$ st: AmbulanceStation): AtStation(amb, st)
$\Rightarrow \Diamond_{\leq mob\_communication\_delay}$
amb.Mobilized $\wedge$ amb.Destination = loc

2. Resolve lack of monitorability for the relationship AtStation

The two goals above are unrealizable by CAC agents because they lack of monitorability for the relationship AtStation. In both cases, the tactic introduce accuracy goal is used to resolve this lack of monitorability.

The goal Achieve[AllocatedAmbulanceMobilizedAtStation] is therefore refined into:

Maintain[AccurateAtStationInfo]

Achieve[AllocatedAmbulanceMobilizedAtStationBasedOnAtStationInfo]

Similarly, the goal Achieve[AllocatedAmmbulanceMobilizedOnRoad] is refined into:

Maintain[AccurateAtStationInfo]

Achieve[AllocatedAmbulanceMobilizedOnRaodBasedOnAtStationInfo]

3. Resolve lack of control for ambulance mobilization at station

Different alternatives can be generated to resolve lack of control for ambulance mobilization when the ambulance is at a station. Theses alternatives are generated by applying the tactic split lack of control with milestone with alternative milestones.

A first alternative consists in transmitting mobilization orders by phone to the adequate station. The tactic split lack of control with milestone is therefore applied with the following milestone:

M: ($\exists$ mc: MobilizationCall): ReceivedAtStation(mc, st)
$\wedge$ mc.AmbMob = ai.AmbId $\wedge$ mc.MobDest = ai.AllDest

thereby generating the two goals

Achieve[MobilizationCallIssuedAtStation]

Achieve[AmbulanceAtStationMobilizedFromMobilizationCall]

The second subgoal is assigned as the responsibility of the AmbulanceStaff agent The first subgoal is further refined into:

Achieve[MobilizationCallRequestedToDespatcher]

Achieve[RequestedMobilizationCallIssuedAtStation]

that are respectively assigned to the ResourceAllocator agent and the a Despatcher agent.

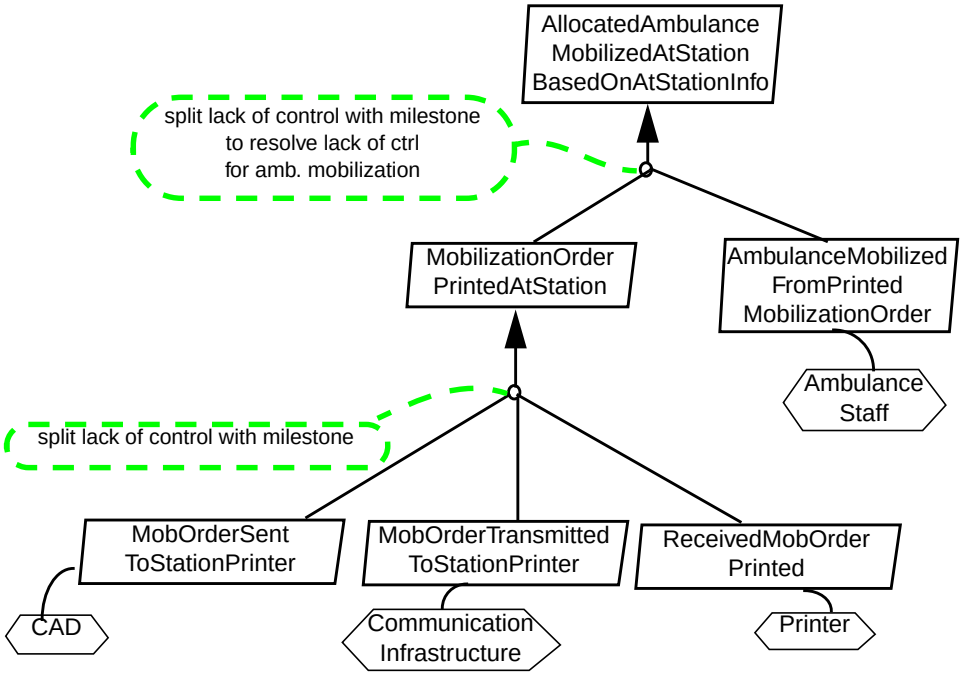


FIGURE 9.30. Refinement and responsibility assignments for the goal Achieve[AllocatedAmbulanceMobilizedAtStationBasedOnAtStationInfo]

Another alternative, the one chosen for the automated system of 1992, is shown in Figure 9.30. It consists in issuing mobilization order to a printer at the adequate station. The tactic split lack of control with milestone is therefore applied with the milestone:

M: (∃ mob_order: PrintedMobilizationOrder): PrintedAtStation(mob_order, st)
∧ mob_order.AmbMob = ai.AmbId
∧ mob_order.MobDest = ai.AllDest,

thereby generating the two goals

Achieve[MobilizationOrderPrintedAtStation]

Achieve[AmbulanceMobilizedFromPrintedMobilizationOrder]

The second goal is assigned as the responsibility of AmbulanceStaff agents.

For the first subgoal, since LAS agents cannot directly control the printers at station, two further application of the tactic split lack of control with milestone generate the subgoals:

Achieve[MobilizationOrderSentToStationPrinter]

Achieve[SentMobilizationOrderTransmittedToStation]

Achieve[TransmittedMobilizationOrderPrintedAtStation]

In the 1992 automated system, these goals are respectively assigned to the CAD software agent, to the Communication Infrastructure agent, and to the Printer agent.

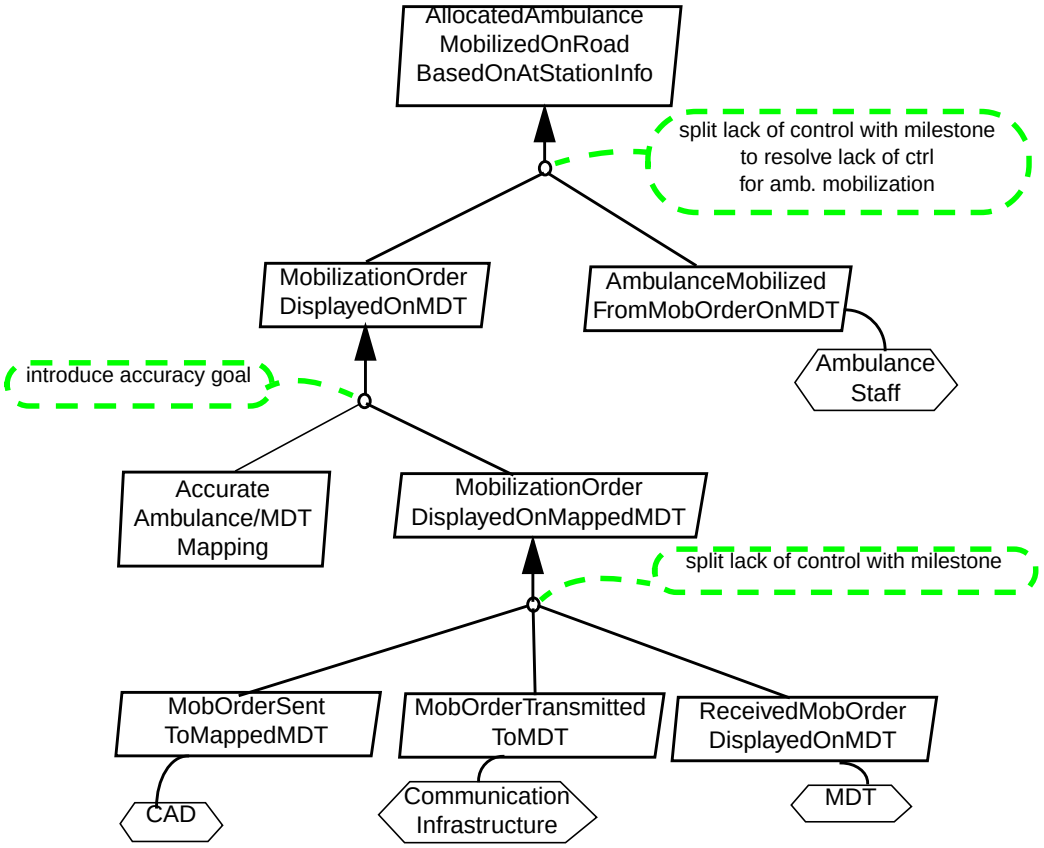


FIGURE 9.31. Goal Refinement and responsibility assignments for the goal Achieve[AllocatedAmbulanceMobilizedAtStationBasedOnAtStationInfo]

4. Resolve lack of control for ambulance mobilization on roads

The refinement of the goal

AchieveAchieve[AllocatedAmbulanceMobilizedOnRaodBasedOnAtStationInfo]

is similar. Lack of control for ambulance mobilization is resolved by applying the tactic split lack of control with milestone. Alternative milestone are used to generate alternative designs. A first alternative consists in transmitting mobilization orders to ambulances on the road through radio communications. Another alternative, the one implemented in the 1992 automated system, consists in mobilizing ambulances on the roads through messages displayed on their Mobile Data Terminals (MDT's). The goal refinement graph and responsibility assignments for the 1992 automated system is shown in Figure 9.31.

9. 1. 2. 4. Refining the goal Maintain[AccurateAmbulanceAvailabilityandLocation-Info]

We now come back in Figure 9.27 to the accuracy goal

Maintain[AccurateAmbulanceAvailabilityAndLocationInfo].

This goal is not realizable by CAC agents because they lack of monitorability for the actual availability and locations of ambulances.

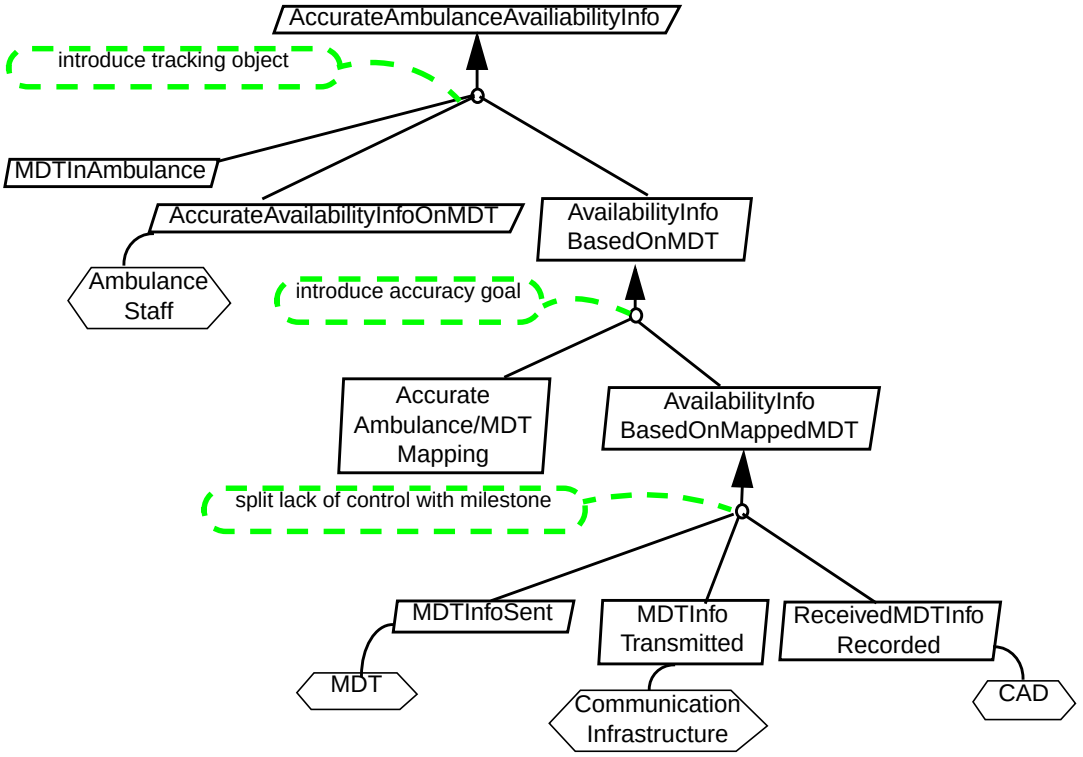


FIGURE 9.32. Goal Refinement and responsibility assignments for the goal Maintain[AccurateAmbulanceAvailabilityInfo]

We begin to split this goal by cases into the following subgoals:

Maintain[AccurateAmbulanceAvailabilityInfo]

Maintain[AccurateAmbulanceLocationInfo].

Each goal is then treated in turn.

1. Maintaining Accurate Ambulance Availability Information

Consider the above goal Maintain[AccurateAmbulanceAvailabilityInfo]. Figure 9.32 shows the goal refinement graph and responsibility assignments corresponding to 1992 the automated system. This refinement graph was again produced through the systematic application of agent-driven tactics. This system relies on the use of Mobile Data Terminals that act as intermediate agents between Ambulance Staff agents and the CAD software.

Further work is required to model formally the delays between the actual availability of ambulances and information about this availability.

As an alternative to this fully automated system, information about ambulances availability could be maintained through radio communications as it was done it the manual system.

2. Responsibility assignments for the goal Maintain[AccurateAmbulanceLocation-Info]

In the fully automated system of 1992, the accuracy

Maintain[AccurateAmbulanceLocationInfo]

is assigned as the responsibility of an Automatic Vehicle Location System (AVLS) agent. Actually, the AVLS is not required to maintain such idealized relationship between the actual locations of ambulances and the information about theses locations. The idealized definition of the accuracy goal is therefore weakened into:

Tracking(ai, amb) $\Rightarrow$ a.Location $\in$ ai.Location

In this definition, the attribute AmbulanceInfo.Location is now declared to be a set of locations that must include the actual location of the ambulance. This deidealization is then propagated in Figure 9.27 to the goal

Achieve[AmbulanceAllocatationBasedOnIncidentFormandAmbulanceInfo WhenNearAm-bAvailable]

that is now defined as follows:

@ if.Encoded
$\Rightarrow \Diamond_{\leq 3'}$ ($\exists$ ai: AmbulanceInfo):
(ai.Allocated $\wedge$ ai.AllocationDest = if.Location
$\wedge$ $\bullet$ ai.Available $\wedge$ $\bullet$ $\neg$ ai.Allocated
$\wedge$ $\bullet$ (ai.Location $\subseteq$ {l: Location | TimeDist(l, if.Location) $\leq$ 11' })
$\vee$ $\neg$ ($\exists$ ai: Ambulance): (ai.Available $\wedge$ $\bullet$ $\neg$ ai.Allocated
$\wedge$ ai.Location $\subseteq$ {l: Location | TimeDist(l, if.Location) $\leq$ 11' })

This deidealized definition requires that every possible locations (i.e. the locations in the set ai.Location) of the allocated ambulance is at less than 11 minutes from the incident location. The parent goal of these two goals remains unchanged. (Note that the accuracy goal could be further deidealized because one only needs location information about available ambulances.)

As an alternative to the AVLS system, information about ambulances' locations could be maintained through radio communications as it was done in the manual system. Without going into the details, the refinement graph for this goal together with alternative responsibility assignments are shown in Figure 9.33. In this alternative, ambulances staffs communicate their new location when they stop at a new location; and communicate their depart and destination locations when they leave a location. The requirement AmbulanceOnNormalRoute assigned to ambulance staff is used to infer a set of possible location for the ambulance based on the communicated depart and destination locations of the ambulance.

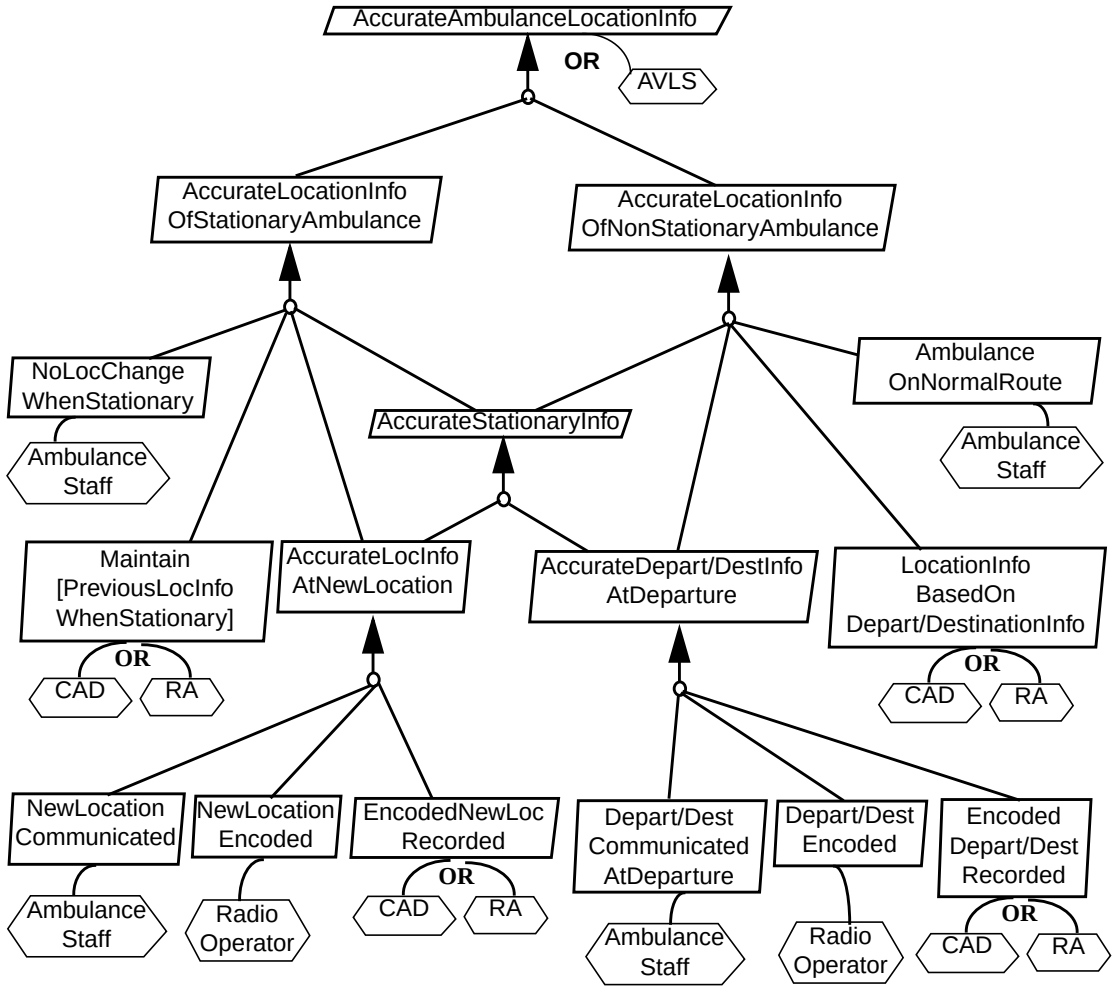


FIGURE 9.33. Goal Refinement and alternative responsibility assignments for the goal Maintain[AccurateAmbulanceLocationInfo]

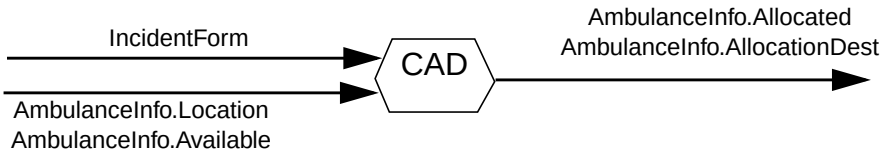
9. 1. 3. Goal Operationalization

Agents interfaces and operational requirements are now derived from the terminal goals generated during the goal refinement process. Consider for instance the following terminal goal appearing in Figure 9.27:

Goal Achieve[Amb.AllocatationBasedOnIncidentFormandAmbulanceInfo WhenNearAmbAvailable]

FormalDef $\forall$ c: UrgentCall, if: IncidentForm
@ if.Encoded
$\Rightarrow \Diamond_{\leq allocation\_delay}$ ($\exists$ ai: AmbulanceInfo):
(ai.Allocated $\wedge$ ai.AllocationDest = if.Location
$\wedge$ ● ai.Available $\wedge$ ● $\neg$ ai.Allocated
$\wedge$ ●TimeDist(ai.Location, if.Location) $\leq$ 11')
$\vee \neg$ ($\exists$ ai: Ambulance): (ai.Available $\wedge$ ● $\neg$ ai.Allocated
$\wedge$ TimeDist(ai.Location, if.Location) $\leq$ 11')

In the 1992 automated system, this goal is assigned as the responsibility of the CAD software agent. The portion of the agent interface model derived from that responsibility assignment is given by:



From the formal definition of this goal, we also derive the following operation to be performed by the CAD agent, together with required pre- and trigger conditions that guarantees the satisfaction of the goal:

Operation AllocateAmbulance
PerfBy CAD
Input IncidentForm {**arg** if}
AmbulanceInfo/Location, Available
Output AmbulanceInfo {**res** ai}/Allocated, AllocationDest
Dompre $\neg$ (ai: AmbulanceInfo): ai.Allocated $\wedge$ ai.Destination = if.Location
DomPost ai.Allocated $\wedge$ ai.AllocationDest = if.Location
ReqTrigFor AllocatationBasedOnIncidentFormandAmbulanceInfoWhenNearAmbAvailable
$\blacksquare_{\leq allocation\_delay}$ if.Encoded
ReqPreFor AllocatationBasedOnIncidentFormandAmbulanceInfo WhenNearAmbAvailable
ai.Available $\wedge \neg$ ai.Allocated $\wedge$ TimeDist(ai.Location, if.Location) $\leq$ 11')
$\vee \neg$ ($\exists$ ai: Ambulance): (ai.Available $\wedge \neg$ ai.Allocated
$\wedge$ TimeDist(ai.Location, if.Location) $\leq$ 11')

Further agent interface, operations, and requirements on operations are similarly derived from other goals assigned as the responsibility of single agents. Figure 9.34 shows the agent interface model derived from the responsibility assignments corresponding to the fully automated system of 1992. Alternative responsibility assignments lead of course to alternative agent interface. These alternative models correspond to systems in which more or less functions are automated. Note that switching from one model to another by selecting alternative goal refinements, responsibility assignments and agent interfaces corresponds to a staged introduction of automated capabilities.

9. 1. 4. Obstacle Analysis

We now derive obstacles for each terminal goals and assumptions generated during the goal refinement process. Many of them are formalized; a mix of regression, obstruction patterns and informal heuristics from Chapter 8 is used. We then compare the list of potential obstacles thereby obtained with the scenarios that actually occurred during the two system failures in October-November 1992. While our obstacles cover the various problems that occurred during those failures (notably, Inaccuracy problems), they also cover many other problems that could (but did not) occur --see the comparison tables below. Handling those obstacles during goal-oriented requirements elaboration would have forced requirements engineers to raise issues whose resolution hopefully would have resulted in making such scenarios (*and others*) infeasible. Finally we explore the space of possible resolutions by application of the tactics discussed in Chapter 8.

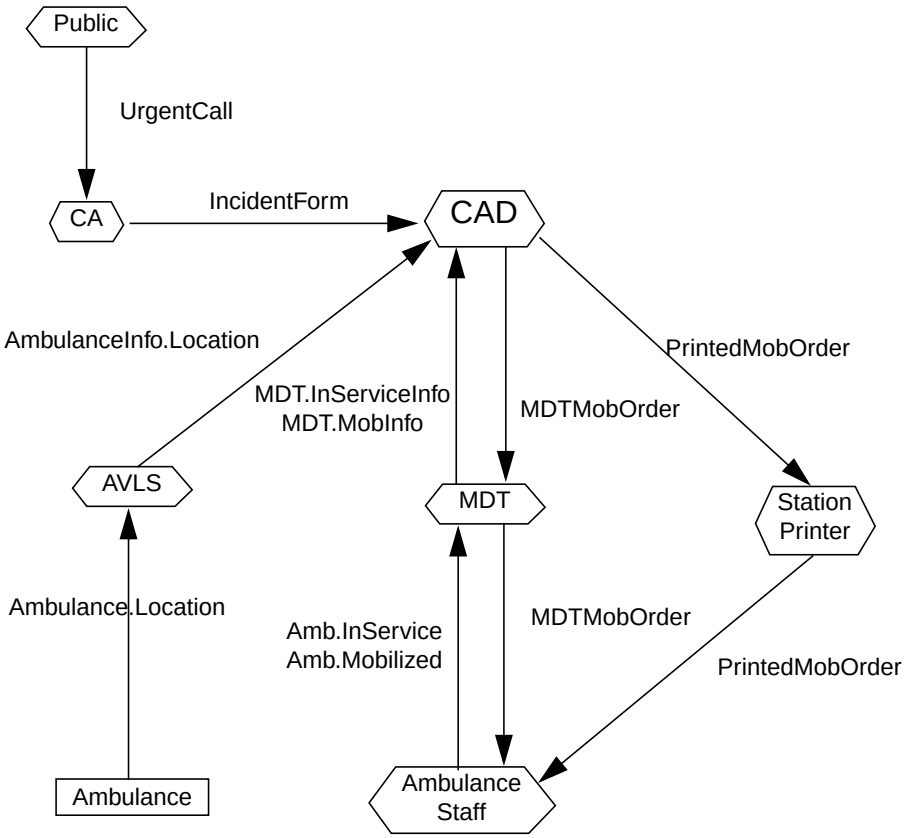


FIGURE 9.34. Partial Agent Interface Model Derived from Responsibility assingments in Figures 9.24 to 9.33 corresponding to the 1992 fully automated system
(the communication infrasturcture agent is omitted from the diagram)

9. 1. 4. 1. Obstacles generation

1. Generating obstacles to the assumption Achieve[IncidentResolvedByIntervention]

Let us illustrate some of the formal derivations first. Consider the following assumption appearing in Figure 9.24:

Goal *Achieve*[IncidentResolvedByIntervention]
FormalDef ∀ a: Ambulance, inc: Incident
Intervention (a, inc) ⇒ ◊ inc.Resolved

Applying the regression procedure, we negate this goal to produce the high-level obstacle

Obstacle IncidentNotResolvedByIntervention:
FormalDef ◊ ∃ a: Ambulance, inc: Incident
Intervention (a, inc) ∧ ❑ ¬ Resolved (inc)

In order to identify further subobstacles, we look for domain properties that provides necessary conditions for incident resolution. Suppose that the following properties are identified:

For an incident to be considered resolved, the following conditions must be satisfied
(i) every patient injured in the incident receives the necessary urgent care at the incident scene;
(ii) every patient injured in the incident is admitted at an hospital

These two properties may be formalized as follows:

Resolved (inc) ⇒
(∀ p: Patient) Injured (p, inc) → TreatedAtLocation(p,inc)

Resolved (inc) ⇒
(∀ p: Patient) Injured (p, inc) →
(∃ h: Hospital) AdmittedAt (p, h)

Regressing the high-level obstacle above through these two domain properties yields the following two subobstacles:

Obstacle PatientNotTreatedAtLocation
InformalDef an ambulance makes an intervention at an incident, and some patient does not receives the necessary urgent care at the incident scene.
FormalDef ◊ ∃ a: Ambulance, inc: Incident
Intervention (a, inc)
∧ ❑ (∃ p: Patient, r: Resource)
Injured (p, inc) ∧ ¬ TreatedAtLocation(p,inc)

Obstacle PatientNotAdmittedToHospital
InformaDef A patient injured in the incident is not admitted at an hospital
FormalDef ◊ ∃ a: Ambulance, inc: Incident
Intervention (a, inc)
∧ ❑ ∃ p: Patient
Injured (p, inc) ∧ (¬ ∃ h: Hospital) AdmittedAt (p, h)

Considering the first subobstacle, we identify the domain property that a patient receives the necessary urgent care at the location scene if the medical resources critically needed by patients injured in the incident are effectively used on the patients:

TreatedAtLocation(p,inc) ⇔
(∀ r: Resource) CriticallyNeeds (p, r) →
(∃ ru: ResourceUnit) Unit (ru, r) ∧ UsedOn (ru, p)

Regressing the obstacle PatientNotTreatedAtLocation through the domain property yields the new subobstacle:

Obstacle CriticalCareNotGivenToPatient
InformalDef an ambulance makes an intervention at an incident, and medical resources are not used on a patient that critically needs it.
FormalDef ◊ ∃ a: Ambulance, inc: Incident
Intervention (a, inc)
∧ ❑ (∃ p: Patient, r: Resource)
Injured (p, inc) ∧ CriticallyNeeds (p, r)
∧ ¬ (∃ ru: ResourceUnit) Unit (ru, r) ∧ UsedOn (ru, p)

In order to regress this obstacle further, we identify the domain property that in order to use a medical resource on a patient, there must be a resource unit in the ambulance that is not already used for another patient:

Intervention (a, inc)
∧ Injured (p, inc) ∧ UsedOn (ru, p)
⇒ InAmbulance (ru, a)
∧ ¬ (∃ p': Patient) p' p∧ UsedOn (ru, p')

Regressing the obstacle CriticalCareNotGivenToPatient through the domain property yields the new subobstacle:

Obstacle InsufficientResourceInAmbulance
FormalDef ◊ ∃ a: Ambulance, inc: Incident
Intervention (a, inc)
∧ ❑ ∃ p: Patient, r: Resource
Injured (p, inc) ∧ CriticallyNeeds (p, r)
∧ Intervention (a, inc)
∧ (∀ ru: ResourceUnit) Unit (ru, r) →
InAmbulance (ru, a) →
(∃ p': Patient) p' p∧ UsedOn (ru, p')

By completing this refinement we obtain a new subobstacle to produce a domain-complete set of subobstacles to CriticalCareNotGivenToPatient:

Obstacle AvailableResourceNotUsedOnPatient
FormalDef ◊ ∃ a: Ambulance, inc: Incident
Intervention (a, inc)
∧ ❑ ∃ p: Patient, r: Resource
Injured (p, inc) ∧ CriticallyNeeds (p, r)
∧ Intervention (a, inc)
∧ (∃ ru: ResourceUnit) Unit (ru, r) ∧ InAmbulance (ru, a)
∧ ¬ (∃ p': Patient) p' p∧ UsedOn (ru, p')

Further refinement of the latter subobstacle by, e.g., use of the heuristics in Chapter 8, yields new subobstacles such as WrongInfoAboutPatient and ResourceOutOfOrder. In the former case, one might find out that the incident form produced by the CAD has inaccurate or missing information.

The complete obstacle refinement tree derived is as follows:

IncidentNotResolvedByIntervention
- ← CriticalCareNotGivenToPatient
  - ← InsufficientResourceInAmbulance
    - ← WrongInfoAboutIncident
    - ← ResourceUnavailable
    - ← ResourceConfusion
  - ← AvailableResourceNotUsedOnPatient
    - ← WrongInfoAboutPatient
    - ← ResourceOutOfOrder
- ← PatientNotAdmittedToHospital
  - ← PatientNotTransportedToHospital
    - ← PatientNotPutInAmbulance
      - ← InsufficientAmbulanceCapacity
      - ← PatientNotInAvailableAmbulance
        - ← ...
    - ← PatientInAmbulanceNotPortedToHospital
  - ← PatientAtHospitalNotAdmitted
    - ← NoBedAvailableAtHospital
    - ← AvailableBedNotAssigned

This tree amounts to a *goal-based* fault tree.

2. Generating obstacles to the assumption Achieve[MobilizedAmbulanceIntervention]

Consider now the following other assumption appearing in the goal graph in Figure 9.25:

Goal *Achieve*[MobilizedAmbulanceIntervention]
Responsibility AmbulanceStaff
FormalDef $\forall$ a: Ambulance, loc: Location
a.Mobilized $\wedge$ a.Destination = loc
$\wedge$ ● a.Available $\wedge$ TimeDist (a.Location, loc) $\leq$ 11
$\Rightarrow \Diamond_{\leq 11'}$ a.Intervention $\wedge$ a.Location = loc

Negating the goal yields a high-level obstacle:

Obstacle MobilizedAmbulanceNotInTimeAtDestination
FormalDef $\Diamond \exists$ a: Ambulance, inc: Incident
a.Mobilized $\wedge$ a.Destination = loc
$\wedge$ ● a.Available $\wedge$ TimeDist (a.Location, loc) $\leq$ 11
$\wedge \square_{\leq 11m}$ ¬ a.Intervention $\wedge$ a.Location = loc

The non-persistence obstruction patterns of Table 8.3 in Chapter 8 suggest looking for domain properties involving persistent conditions ***P*** that must continuously hold, from the time of allocation to the time of intervention:

a.Mobilized $\wedge$ a.Destination = loc
$\wedge$ ● a.Available $\wedge$ TimeDist (a.Location, loc) $\leq$ 11
$\wedge \Diamond$ a.Intervention $\wedge$ a.Location = loc
$\Rightarrow$ ***P*** *W* (***P*** $\wedge$ a.Intervention $\wedge$ a.Location = loc)

Three candidates ***P*** are suggested from the antecedent of the goal:

P1: a.InService
P2: a.Mobilized $\wedge$ a.Destination = loc
P3: TimeDist (a.Loc, inc.Loc) < TimeDist ($\bullet$ a.Loc, inc.Loc)

These candidates produce three persistence conditions that are domain properties indeed: the first property says that if a sufficiently close ambulance is mobilized and intervenes at the location within 11 minutes, then it remains in service unless it intervenes at that location; the second says that the ambulance remains mobilized for that location unless it intervenes at the location; the latter says that the time distance between the mobilized ambulance and the destination keeps decreasing unless the ambulance intervenes at the location. We may therefore apply the second non-persistence pattern in Table 8.3 to generate the three following obstacles (one for each persistent condition):

Obstacle MobilizedAmbulanceStopsServiceBeforeIntervention
FormalDef $\Diamond \exists$ a: Ambulance, loc: Location
a.Mobilized $\wedge$ a.Destination = loc
$\wedge \bullet$ a.Available $\wedge$ TimeDist (a.Location, loc) $\leq$ 11
$\wedge$ ($\neg$ (a.Intervention $\wedge$ a.Location = loc) $U_{\leq 11}$, $\neg$ a.InService)

Obstacle AmbulanceMobilizationRetracted
FormalDef $\Diamond \exists$ a: Ambulance, loc: Location
a.Mobilized $\wedge$ a.Destination = loc
$\wedge \bullet$ a.Available $\wedge$ TimeDist (a.Location, loc) $\leq$ 11
$\wedge$ ($\neg$ (a.Intervention $\wedge$ a.Location = loc) $U_{\leq 11}$, $\neg$ (a.Mobilized $\wedge$ a.Destination = loc))

Obstacle MobilizedAmbulanceStoppedOrInWrongDirection
FormalDef $\Diamond \exists$ a: Ambulance, loc: Location
a.Mobilized $\wedge$ a.Destination = loc
$\wedge \bullet$ a.Available $\wedge$ TimeDist (a.Location, loc) $\leq$ 11
$\wedge$ ($\neg$ (a.Intervention $\wedge$ a.Location = loc) $U_{\leq 11}$, TimeDist (a.Loc, loc) $\geq$ TimeDist ($\bullet$ a.Loc, loc))

(In the above assertions, $P U_{\leq d} Q$ stands for $P U Q \wedge \Diamond_{\leq d} P$.)

Further refinement of these formal obstacles based on regression, patterns, and heuristics from Chapter 8 yield the following obstacle OR-refinement tree:

MobilizedAmbulanceNotInTimeAtDestination
- ← Mobilized*AmbulanceStopsServiceBeforeIntervention*
- ← *AmbulanceMobilizationRetracted*
  - ← MobilizedAmbulanceDestinationChanged
    - ← LocationConfusedByCrew
  - ← MobilizedAmbulanceDestinationForgotten
  - ← AmbulanceMobilizationCancelled
- ← *MobilizedAmbulanceStoppedOrInWrongDirection*
  - ← AmbulanceStopped
    - ← AmbulanceBreakdownOrAccident
    - ← AmbulanceStoppedInTraffic
  - ← AmbulanceInWrongDirection
    - ← AmbulanceLost
      - ← CrewInUnfamiliarTerritorry
    - ← TrafficDeviation

3. Generating Obstacles to the subgoals of Achieve[AllocatedAmbulanceMobilized]

We now consider obstacles to the requirements and assumptions generated by the refinements of the goal Achieve[AllocatedAmbulanceMobilized] in Figures 9.29 to 9.31.

For the terminal goal AmbulanceMobilizedFromPrintedMobilizationOrder appearing in Figure 9.30, the obstacle OR-refinement tree generated using our techniques is:

MobOrderNotTakenByAmbulance
- ← MobOrderIntendedForUnavailableAmbulance
- ← MobOrderIgnored
- ← MobOrderTakenByOtherAmbulance

Many reported failures were in fact caused by inappropriate resolution of the latter subobstacle [LAS93].

Obstacles should be identified not only from assumptions about human agents in the environments, but also for requirements assigned to software agents. Consider for instance the following goal appearing in Figure 9.31 and assigned to the CAD software agent:

Goal Achieve[MobilizationOrderSentToMappedMDT]
FormalDef ∀ ai: AmbulanceInfo, loc: Location, mdt_id: MDT_ID
@ ai.Allocated ∧ ai.AllocationDest = loc ∧ ai.MDTID = mdt_id
⇒ ❍(∃ mob_order: MDT_MobilizationOrder):
mob_order.Sent ∧ mob_order.DestMDT = mdt_id ∧ mob_order.AllocationDest = loc

Negating this goal yield the following obstacle:

Obstacle MobOrderNotSentToMappedMDT
FormalDef ◊ ∃ ai: AmbulanceInfo, loc: Location, mdt_id: MDT_ID
@ ai.Allocated ∧ ai.AllocationDest = loc ∧ ai.MDTID = mdt_id
∧ ❍ ¬ (∃ mob_order: MDT_MobilizationOrder):
mob_order.Sent ∧ mob_order.DestMDT = mdt_id ∧ mob_order.AllocationDest = loc

That obstacle is then split into cases to generate the following complete set of subobstacles:

Obstacle MobOrderNotSent
FormalDef ◊ ∃ ai: AmbulanceInfo, loc: Location, mdt_id: MDT_ID
@ ai.Allocated
∧ ❍ ¬ (∃ mob_order: MDT_MobilizationOrder):
mob_order.Sent

Obstacle MobOrderSentWithWrongMDTID
FormalDef ◊ ∃ ai: AmbulanceInfo, loc: Location, mdt_id: MDT_ID
@ ai.Allocated ∧ ai.MDTID = mdt_id
∧ ❍ (∃ mob_order: MDT_MobilizationOrder):
mob_order.Sent ∧ mob_order.DestMDT ≠ mdt_id

Obstacle MobOrderNotSentWithWrongAllocationDest
FormalDef ◊ ∃ ai: AmbulanceInfo, loc: Location, mdt_id: MDT_ID
@ ai.Allocated ∧ ai.AllocationDest = loc ∧ ai.MDTID = mdt_id
∧ ❍ (∃ mob_order: MDT_MobilizationOrder):
mob_order.Sent ∧ mob_order.AllocationDest ≠ loc

(A domain property saying that at most one MDT message can be sent simultaneously is used to refine the parent obstacle.)

Further refinement of the last two obstacles yield the following obstacle refinement tree:

MobOrderNotSentToMappedMDT
- ← *MobOrderNotSent*
- ← *MobOrderSentWithWrongMDTID*
  - ← MobOrderSentToOtherMDT
  - ← MobOrderWithInvalidMDTID
- ← *MobOrderNotSentWithWrongAllocationDest*
  - ← MobOrderNotSentWithOtherValidAllocationDest
  - ← MobOrderNotSentWithInvalidAllocationDest

Tables 9.1 summarizes the obstacles generated for the various terminal goals in Figure 9.30 and 9.31. The table compares the set of obstacles generated systematically using our techniques with the scenarios that actually occurred during the two major system failures in October-November 1992 as reported in [Las93]. While our obstacles cover the various problems that occurred during those failures, they also cover many other potential problems that could (but did not) occur. The table provides, for each requirement/assumption, the responsible agent assigned to it, the (sub)obstacles derived, and features of the scenarios -covered by the obstacle - that occurred during the reported system failures.

Table 9.1: Obstacles to subgoals of the goal *Achieve[AllocatedAmbulanceMobilized]* (Fig. 9.30 - 9.31)

agent	goal	obstacle	Oct/Nov'92 scenario
CAD	MobOrderSent ToStationPrinter	MobOrderNotSent	*no PSTN line free*
		MobOrderSentToWrongStation	
		MobOrderSentToWrongAmbulance	
		MobOrderSentWith WrongDestination	
		InvalidMobOrderSent	
	MobOrderSent To MappedMDT	MobOrderNotSent	
		MobOrderSentToOtherMDT	
		MobOrderNotSent WithOtherValidAllocationDest	
		InvalidMobOrderSentToMDT	
Communic. Infrastructure	MobOrderTransmitted ToStationPrinter	MobOrderNotTransmitted	*radio congestion*, *radio blackspot*
		MobOrderDeliveredAtWrongStation	
		MobOrderCorruptedDuring Transmission ← WrongDestination ← WrongAmbulance ← InvalidMobOrder	
	MobOrderTransmitted ToMDT	MobOrderNotTransitted	
		MobOrderTransmittedAt WrongMDT	
		MobOrderCorruptedDuring Transmission ← WrongDestination ← OtherValidMsgDelivered ← InvalidMsgDelivered	

agent	goal	obstacle	Oct/Nov'92 scenario
Station Printer	ReceivedMobOrder Printed	ReceivedMobOrderNotPrinted ← Paper Jammed ←Out of Paper ←Data Lost ←...	
		PrintedMobOrderUnreadable	
MDT	ReceivedMobOrder DisplayedOnMDT	ReceivedMobOrderNotDisplayed OnMDT	
		IncorrectDestinationDisplayed	
Ambulance Staff	AmbulaceMobilized FromPrinted MobilizationOrder	AmbNotMobilized FromPrintedMobOrder ← MobOrderIgnored ← AmbNotAt Station ← AmbNotAvailable	
		MobOrderTakenByOtherAmbulance ← MobOrderConfuision ← AllocatedAmbNotAvailable ← AllocatedAmbNotAtStation ← established work practice	*crews take different vehicle from those allocated by CAD*
		LocationConfusedByCrew	
	AmbulanceMobilized FromMobOrderOnMDT	AmbulanceNotMobilizedFrom MobOrderonMDT ← MDTMobOrderIgnored ← CrewNotInAmbulance ← AmbulanceNotAvailable	
		AmbulanceMobilizedWithDifferent DestinationThanMDTDestination ← LocationConfusedByCrew ← OtherMobilizationDestination Pending	

4. Generating obstacles to the accuracy goals

The accuracy goals play a critical role in the LAS system. In order to allocate appropriate ambulances, the CAD software needs accurate informations about the locations and availability of ambulances. Many reported failures were caused by inappropriate resolutions of obstacles to these accuracy goals [Las93].

In the fully automated system, the goal Maintain[AccurateAmbulanceLocationInfo] was assigned as the responsibility of an Automated Vehicle Location System (AVLS) agent. From this goal, we generate the obstacle InaccurateAmbulanceLocationInfo. Further refinement of that obstacle would require further knowledge about properties of the AVLS which were unavailable to us. Further refinement of that obstacle is also not necessary for obstacle resolution since it is not necessary to know why the AVLS might fail to locate ambulances accurately. Table 9.2 shows the obstacle to this goal, and actual failures scenarios covered by the obstacle.

Table 9.2: Obstacles to the goal *Maintaim[AccurateAmbulanceLocationInfo]*

agent	goal	obstacle	Oct/Nov'92 scenario
AVLS	AccurateAmbulance LocationInfo	InAccurateAmbulance LocationInfo	*AVLS equipment not working* *Interference from vehicle equipment* *Swapped callsigns* *No AVLS exception reporting*

Consider now the goal Maintain[AccurateAmbulanceAvailabilityInfo] whose refinement is shown in Figure 9.32. Table 9.3 summarizes the obstacles generated from each terminal goals/assumption together with failures scenarios reported in [LAS93] satisfying those obstacles. Again, our obstacles cover the various problem that occurred during the 1992 failures, and identify various other potential problems that could (but did not) occur.

Table 9.3: Obstacles to subgoals of the goal *Achieve [AmbulanceMobilizationKnown]*

agent	goal	obstacle	Oct/Nov'92 scenario
Ambulance Staff	AccurateAvailibilityInfo OnMDT	AmbulanceStaffForgetTo EncodeAvailibilityOnMDT	*crews don't press status buttons*
		AmbStaffPushWrongButton ToEncodeAvailability	*crews press buttons in wrong order*
MDT	MDTInfoSent	MDTInfoNotSent	
		MDTSendsOtherMsg	
		InvalidMDTInfoSent	
Communic. Infrastructure	MDTInfoTransmitted	MDTInfoNotDelivered	*radio channel congestion (particulalry bad at crew log on/off), radio blackspot*
		MDTInfoCorrupted	
CAD	ReceivedMDTInfoRecorded	ReceivedMDTInfoIgnored	*failure of system to catch all data*
		ReceivedMDTInfoConfusedWith OtherMsg	
		ReceivedMDTInfoRecordedFor WrongAmbulance	

9. 1. 4. 2. Obstacles resolution

We now discuss the application of various obstacle resolution tactics from Chapter 8 for some of the obstacles generated.

1. Alternative resolutions for the obstacle MobOrderTakenByOtherAmbulance

Let us first consider the obstacle MobOrderTakenByOtherAmbulance seen in the previous section to obstruct the goal AmbulanceMobilizedFromPrintedMobilizationOrder.

The tactic mitigate obstacle would result in letting the system know that the mobilization order has been taken by the other ambulance. A mitigation goal is thus introduced to resolve this obstacle, say,

MobilizationByOtherAmbulanceKnown.

This new goal may be refined into two subgoals, namely,

MobilizationByOtherAmbulanceSignalledToRadioOperator,

assigned to AmbulanceStaff, and

MobilizationStatusUpdated,

assigned to RadioOperator. (An alternative refinement/assignment would consist in letting the change be signalled to the MDT instead).

The tactic prevent obstacle would result here in the introduction of the new goal

Avoid [AmbulanceMobilizedWithoutOrder].

A benefit of applying this strategy here is that the latter subgoal would also contribute to the other goal

Avoid [DuplicateAmbulanceMobilization]

The new prevention goal might be under responsibility of a human agent at the station or might be operationalized through an automatic system preventing ambulance departure from station if the MDT is not mobilized. (Such resolution would however be quite risky if MDT's or ambulances are likely to break down.)

As suggested in Section 8. 5. 1 of Chapter 8, the tactic make obstacle infeasible in the domain can be used resolve the same obstacle by changing domain properties of the application domain. In this case, it would consists in transforming the PrintedMobilizationOrder object so that it does not mention the incident location any more; the latter information would only be given by the MDT inside the ambulance. (Such resolution also seems quite risky if MDT's or ambulances are likely to break down.)

The tactic choose alternative goal would result in an alternative operationalization in which mobilization orders sent to stations do not prescribe which particular ambulance to mobilize but instead leave that decision to ambulance crews. In this case, this goes together with an agent substitution and a domain transformation (as PrintedMobilizationOrder objects no longer have an attribute indicating the target ambulance).

Finally, an application of the tactic reduce obstacle might consist here in trying to change ambulance crew practice by a reward/dissuasion system.

2. Examples of goal/agent substitutions

The more efficient way to resolve an obstacle is to eliminate the problem completely by choosing alternative goal refinement and responsibility assignments that do not involve the obstructed goal any more.

A first example of goal substitution was given above to resolve the obstacle MobOrderTakenByOtherAmbulance.

As another example of goal substitution, the obstacle InaccurateAmbulanceLocationInfo obstructing the gaol AccurateAmbulanceLocationInfo assigned to the AVLS system to maintain accurate information about ambulances location can be resolved by selecting another goal refinement and responsibility assignments in which informations about ambulances locations are maintained through radio communications with ambulance staffs (Figure 9.31). This is an example of both agent substitution and goal substitution.

Similarly, the obstacles AmbStaffPressWrongButton or AmbStaffPressButtonInWrongOrder obstructing the goal AccurateAvailibilityInfoOnMDT can be resolved by choosing alternative goals and responsibility assignments in which ambulances availability is maintained through radio communcations.

3. Examples of Obstacles Tolerance

We now illustrate the tactic restore goal. Consider the obstacle

MDTMobOrderIgnored

that appears at the bottom of Table 9.1. A low-level restoration goal would be to generate an audible signal to make crews aware of the mobilization order.

An alternative, complementary, higher-level resolution would consist in introducing a higher-level restoration goal

FailedMobilizationRecovered

to resolve the higher-level obstacle

AllocatedAmbulanceNotMobilized

This goal would restore the higher-level goal AllocatedAmbulanceMobilized through the following goal refinement tree:

FailedMobilizationRecovered
- ← AmbulanceMobilizationKnown
  - ← ...
- ← UnrespondedAllocationRestored
  - ← UnrespondedAllocationSignalled
  - ← SignalledUnrespondedAllocReallocated

This is the tactic that seems to have been followed for the 1992 system. A problem here is that the operationalization of the restoration goal contributes to the obstacle in Table 9.1 MobOrderNotTransmitted (because of radio congestion). Since that obstacle obstruct a subgoal of Achieve[AllocatedAmbulanceMobilized], it may create a snowball effect. This is exactly what happened in November 1992.

4. Deidealizing Goals/Assumptions Definitions

Finally, we illustrate the *goal deidealization* strategy on the overideal goal

$\forall$ a: Ambulance, inc: Incident
Mobilized (a, inc) $\Rightarrow \Diamond$ Intervention (a, inc)

The following obstacle was generated by a non-persistence pattern from Table 1:

$\Diamond \exists$ a: Ambulance, inc: Incident
Mobilized (a, inc)
$\wedge$ ($\neg$ Intervention (a, inc) *U* Breakdown (a))

Using the third deidealization pattern of Table 7 in Section 8.5.1, we obtain the weakened version for that goal:

$\forall$ a: Ambulance, inc: Incident
Mobilized (a, inc) $\Rightarrow \Diamond$ Intervention (a, inc) $\vee$ Breakdown(a)

The propagation will result in strengthened companion goals like

$\forall$ inc: Incident, p: Person
Reported (inc, p) $\Rightarrow$
$\Diamond \exists$ a: Ambulance, inc: Incident
Mobilized (a, inc)
$\wedge$ ($\neg$ Breakdown(a) *W* Intervention (a, inc))

to be refined and deidealized in turn.

As another example, consider the idealized assumption IncidentResolvedByIntervention assuming that an incident is resolved by the intervention of a single ambulance. We identified that this assumption is obstructed by the obstacles PatientNotTreatedAtLocation and PatientNotAdmittedAtHospital. In this case, the idealized assumption is removed and replaced by the goal:

Achieve[EveryPatientTreatedAtLocationAndAdmittedToHospital].

The goal refinement tree for the high-level goal IncidentResolved is now given by:

IncidentResolved
- ← IncidentReported
- ← FastAmbulanceIntervention
- ← EveryPatientTreatedAtLocationAndAdmittedToHospital

In this goal refinement, the goal FastAmbulanceIntervention, a simple renaming of our initial goal AmbulanceIntervention, is concerned with the rapid intervention of a first ambulance at the incident scene as required by the Government standard. The new subgoal address separately the problem of mobilizing sufficient ambulances carrying the appropriate medical resources. Note that the goal FastAmbulanceIntervention is concerned with the intervention of an ambulance *for an incident*, whereas the goal EveryPatientTreatedAtLocationAndAdmittedToHospital is concerned with the intervention of an ambulance *for a patient* (this refinement can be seen as decomposition by views [Jac96]: the incident view and the patient view). This new goal will then be refined in a way that is much similar to the refinement of the goal FastAmbulanceIntervention.

9. 2. The BART Train Control Case Study

9. 2. 1. Introduction

The second case study is concerned with the development of a portion of the new Advanced Automatic Train Control system being developed for the San Francisco Bay Area Rapid Transit (BART). The purpose of the new system is to serve more passengers by running trains more closely spaced.

The case study description [Win99] focuses on those aspects of BART that are necessary to control the speed and acceleration for the trains in the system. The problem is to develop the speed/acceleration control system whose responsibility is to get trains from one point to another as fast and smoothly as possible, subject to the following safety constraints:

- A train should not enter a closed gate. (In the context of the BART system, a gate is not a physical gate, but a signal, received by the speed/acceleration control system, that establish when a train has the right to enter a track segment.)
- A train should never get so close to a train in front so that if the train in front stopped suddenly (e.g., derailed) the following train would hit it.
- A train should stay below the maximum speed that track segment can handle.

9. 2. 2. Identifying and Formalizing Preliminary Goals

1. Identifying goals from the initial document

The first step of the elaboration method consists in identifying high-level goals from the initial problem statement. Figure 9.35 shows a portion of the goal graph identified after a first reading of the initial document. The goals were obtained by searching for keywords such as “purpose”, “objective”, “intent”, “in order to”, and so forth. In this graphical specification, clouds denote softgoals and optimization goals (used in general to select among alternatives - see Section 3.2.4.6), parallelograms denote Achieve/Maintain/Avoid goals that constrain the behaviours of the system.

The objective of the new BART system is to serve more passengers. Figure 9.35 shows this high-level goal (ServeMorePassenger) together with alternative subgoals that contribute to its satisfaction. Two alternative subgoals contributing to the goal ServeMore-Passenger are to run trains more closely spaced (TrainsMoreCloselySpaced) and to add more tracks (NewTracksAdded).

Another goal of the system is to minimize costs, which is refined into the two subgoals Minimize[DvlptCosts] and Minimize[OperationalCosts]. The figure also records a conflict between the goal NewTracksAdded and the goal Minimize[DvlptCosts].

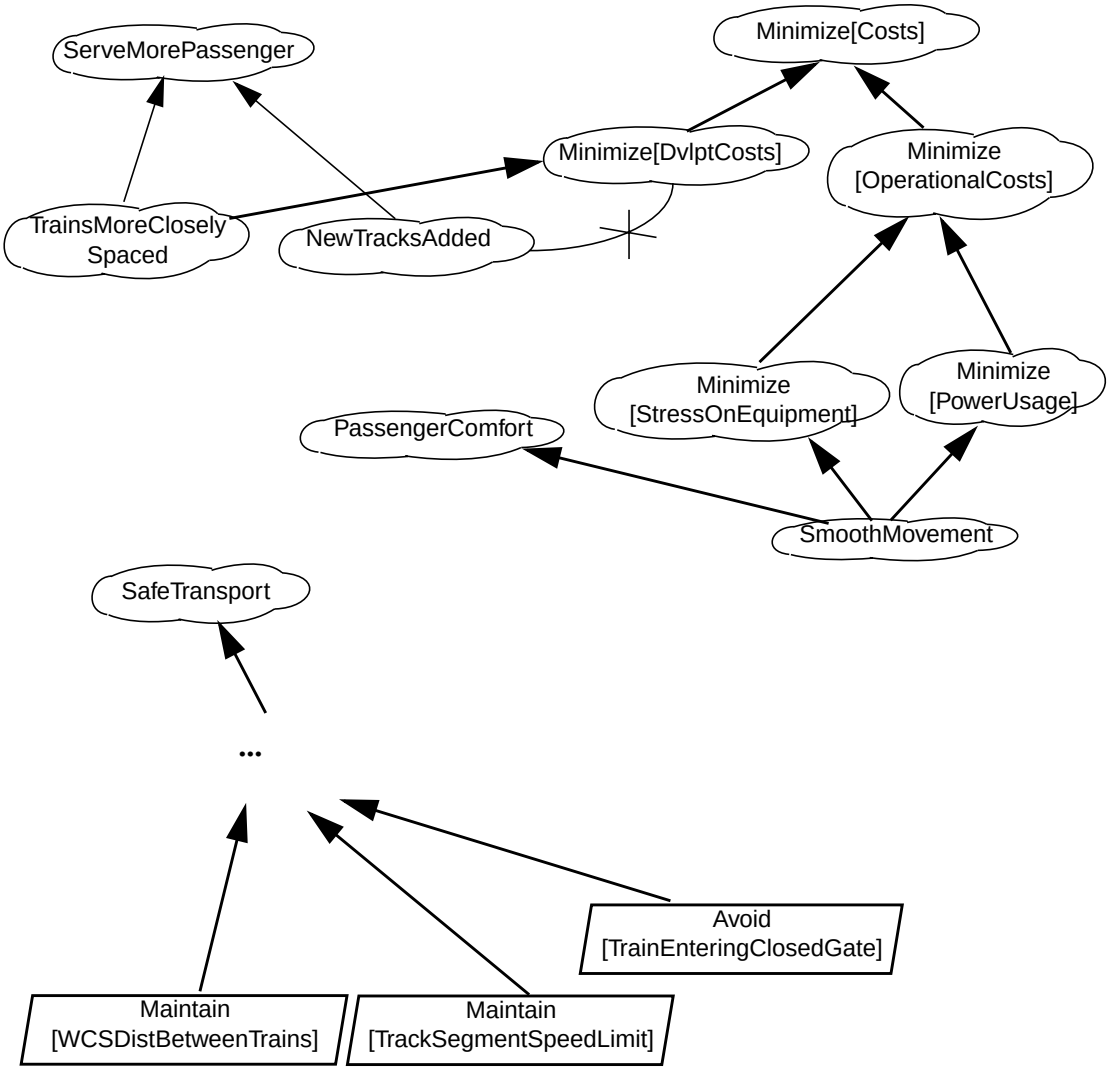


FIGURE 9.35. Preliminary goal graph for the BART system

Another import aspects of the BART system concern the safety of transport. The initial problem statement defines three safety goals that the system must satisfy:

Goal Maintain[WCSDistBetweenTrains]
Definition A train should never get so close to a train in front so that if the train in front stopped suddenly (e.g., derailed) the (following) train would hit it.

Goal Avoid[TrainEnteringClosedGate]
Definition A train should not enter a closed gate.

Goal Maintain[TrackSegmentSpeedLimit]
Definition A train should stay below the maximum speed the track segment can handle.

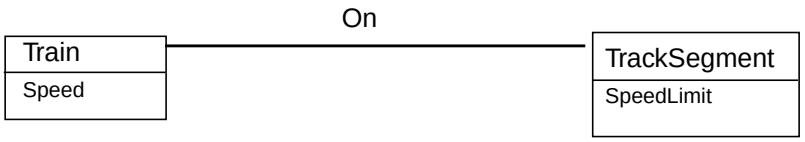
Finally, not only should trains run fast and safely, they should also run smoothly. Figure 9.35 shows the goal SmoothMovement and its goal dependencies mentioned in the case study description: the smooth movement of trains contributes to the passengers' comfort, but also minimize wear stress on equipment and power usage. These last two goals contributes to the satisfaction of the goal Minimize[OperationalCosts].

2. Formalizing Goals and Identifying Objects

The three safety goal identified in the previous section may be defined more precisely. The goal Maintain[TrackSegmentSpeedLimit] is formally defined as follows:

Goal Maintain[TrackSegmentSpeedLimit]
Definition A train should stay below the maximum speed the track segment can handle.
FormalDef $\forall$ tr: Train, s: TrackSegment
On(tr, s) $\Rightarrow$ tr.Speed $\leq$ s.SpeedLimit

The predicates, objects, and attributes appearing in this goal formalization give rise to the following portion of the object model:



The goal Maintain[WCSDistBetweenTrains] is also defined formally:

Goal Maintain[WCSDistBetweenTrains]
Definition A train should never get so close to a train in front so that if the train in front stopped suddenly (e.g., derailed) the following train would hit it.
FormalDef $\forall$ tr1, tr2: Train
Following(tr1, tr2) $\Rightarrow$ tr2.Loc - tr1.Loc $\geq$ tr1.WCSDist

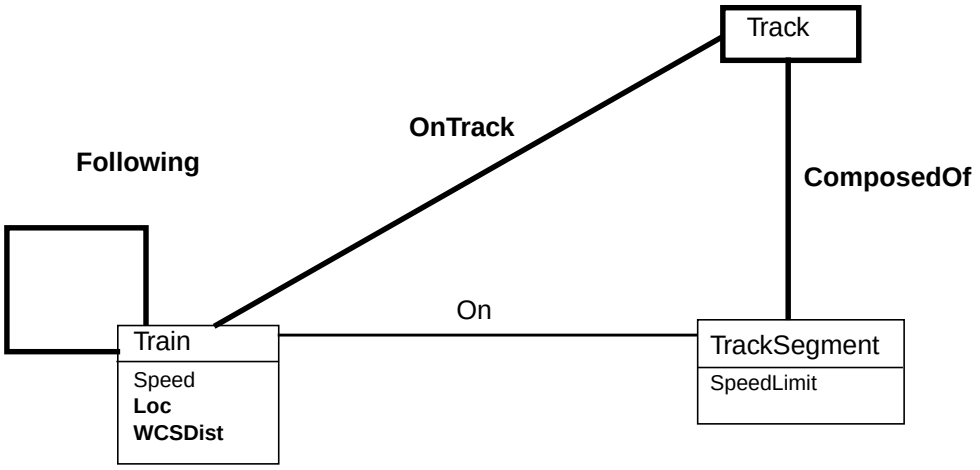
In this definition, the attribute WCSDist denotes the actual worst case stopping distance of a train based on the *physical speed* of the train. The definition of the worst case stopping distance of a train is given as part of the domain knowledge. (Note that the initial document defines a worst case stopping distance based on the *commanded speed* of the train rather then on the physical speed.)

We assume that the location of a train is given by its position on a given track. (For simplicity, we also assume that trains have no length.) The predicate Following(tr1, tr2) appearing in this definition is then formally defined by:

Following(tr1, tr2) $\Leftrightarrow$
($\exists$ track: Track): OnTrack(tr1, track) $\wedge$ OnTrack(tr2, track) $\wedge$ tr1.Loc $\leq$ tr2.Loc
$\wedge \neg$ ($\exists$ tr3: Train): OnTrack(tr3, track) $\wedge$ tr1.Loc $\leq$ tr3.Loc $\wedge$ tr3.Loc $\leq$ tr2.Loc

Note that according to this definition, two trains linked by the Following relationship are necessarily on the same track.

The initial portion of the object model is now enriched from these definitions:

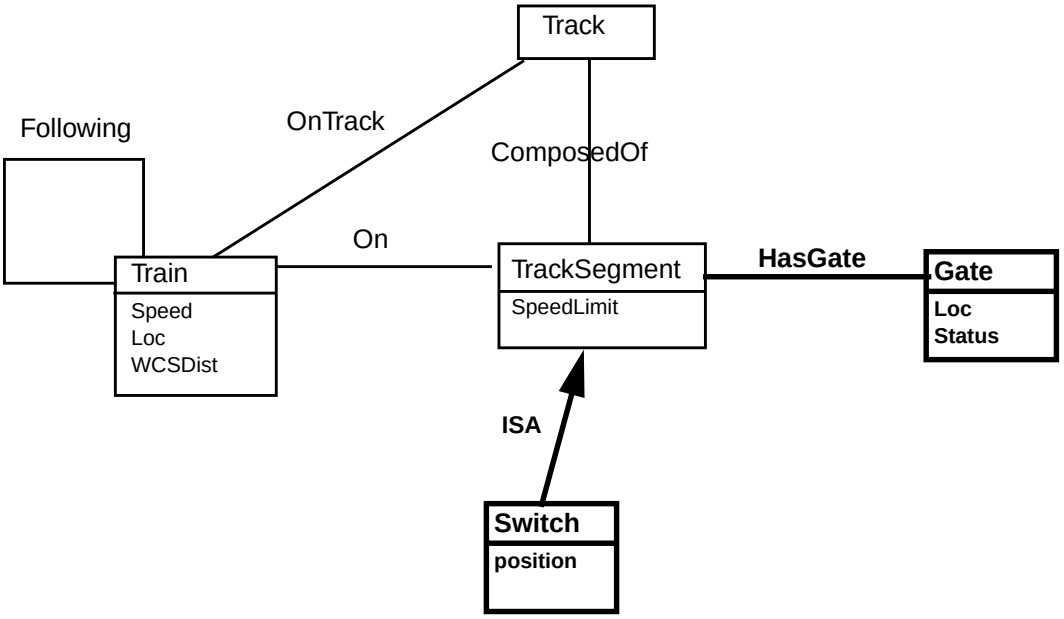


Finally, the goal Avoid[TrainEnteringClosedGate] can be given the following first-sketch definition:

Goal Avoid[TrainEnteringClosedGate]
Definition A train should not enter a closed gate.
FormalDef ∀ g: Gate, s: TrackSegment, tr: Train
g.Status = 'Closed' ∧ HasGate(s, g) ⇒ ¬ @ ¬ On(tr, s)

Note that gates are placed at the end of some track segments. The formal definition says that a train should not leave a track segment if the gate ending that segment is closed. (Remember that gates are not physical gates.)

The portion of the object model derived from the formalization of the goals is given by:



The above definition of the goal Avoid[TrainEnteringClosedGate] is too strong to be satisfiable in the domain. A train cannot stop instantaneously, therefore if a gate becomes closed when the train is too close to the gate, it will be impossible for the train to stop in time. In the actual BART system, the definition of that goal is weakened so that a train is allowed to enter a closed gate if the gate has become closed when the distance between

the train and the gate was too short for the train to stop in time. (Note that this correspond to an application of the tactic weaken goal with unsatisfiability condition). The actual definition of the goal Avoid[TrainEnteringClosedGate] is therefore given by:

Goal Avoid[TrainEnteringClosedGate]
Definition A train should not enter a closed gate provided that the gate has been closed when the distance between the train and the gate was more than the worst case stopping distance of the train.
If the gate is open when the distance between the train and the gate is less than the worst case stopping distance of the train, the train may ignore the gate, even if it becomes closed later.
FormalDef $\forall$ g: Gate, s: TrackSegment, tr: Train
(g.Status = 'Closed' B (g.Loc - tr.Loc) $\geq$ tr.WCSDist)
$\wedge$ HasGate(s, g)
$\Rightarrow \neg @ \neg$ On(tr, s)

Allowing trains to enter a closed gate if the gate becomes closed when it is impossible for the train to stop in time is not necessarily unsafe. The rationale for such weakening of the goal can only be understood by identifying the higher level goal that this goal refines. This is done in the next section.

3. Identifying new goals trough WHY questions

Asking WHY question about the goal Avoid[TrainEnteringClosedGate] yields a new portion of the goal graph, shown in Figure 9.36.

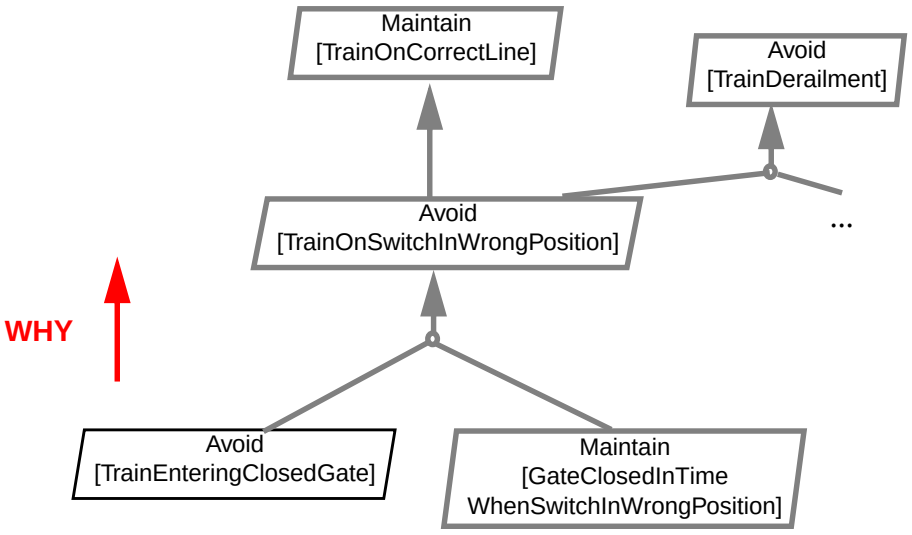


FIGURE 9.36. Asking WHY questions for the goal Avoid[TrainEnteringClosedGate]

The graph shows that the purpose of not entering closed gates is to keep trains from passing through switches that are not appropriately positioned:

Goal Avoid[TrainOnSwitchInWrongPosition]
InformalDef A train should not enter a switch if it is not appropriately positioned.
FormalDef $\forall$ tr: Train, sw: Switch, track: Track
ApproachingSwitchOnTrack(tr, sw, track) $\wedge$ sw.Position $\neq$ tr.Direction $\Rightarrow \neg @$ On(tr, sw)

In this goal graph, the definition of the goal

Maintain[GateClosedInTimeWhenSwitchInWrongPosition]

is elicited formally by matching a chain-driven refinement pattern to the formalization of the parent goal Avoid[TrainOnSwitchInWrongPosition] and of the initial goal Avoid[TrainEnteringClosed Gate]. After further simplification of the generated formal assertion, the goal is defined as follows:

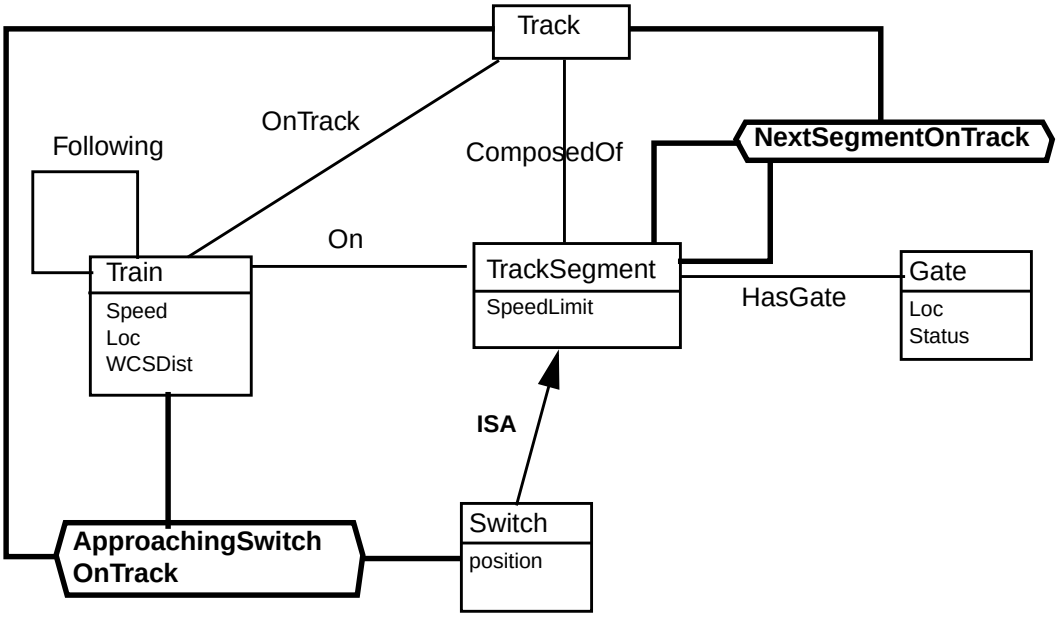
Goal Maintain[GateClosedInTimeWhenSwitchInWrongPosition]
InformalDef When a switch is not appropriately positioned for a train approaching the switch on a given track, the gate guarding the switch on that track must be closed in time so that it is still possible for the train to stop before the switch.
FormalDef ∀ tr: Train, sw: Switch, s: TrackSegment, g: Gate, track: Track
ApproachingSwitchOnTrack(tr, sw, track) ∧ sw.Position ≠ tr.Direction
∧ NextSegmentOnTrack(track, s, sw)
∧ HasGate(s, g)
⇒
g.Status = 'Closed' *B* (g.Loc - tr.Loc) ≥ tr.WCSDist

The following domain properties were also identified and used to produce this goal definition:

every track segment leading to a switch is ended with a gate
NextSegmentOnTrack(track, s, sw) ⇒ (∃ g: Gate): HasGate(s, g)

a train enters a switch iff it leaves a track segment preceding the switch
@ On(tr, sw)
⇔ (∃ track: Track, s: TrackSegment): NextSegmentOnTrack(track, s, sw) ∧ @ ¬ On(tr, s)

The object model derived from these definitions is given by:



Similarly, asking WHY questions about the goals Maintain[TrackSegmentSpeedLimit] and Maintain[WCSDistBetweenTrains] yield respectively the goals Avoid[TrainDerailment] and Avoid[TrainsCollisions] (Figure 9.37).

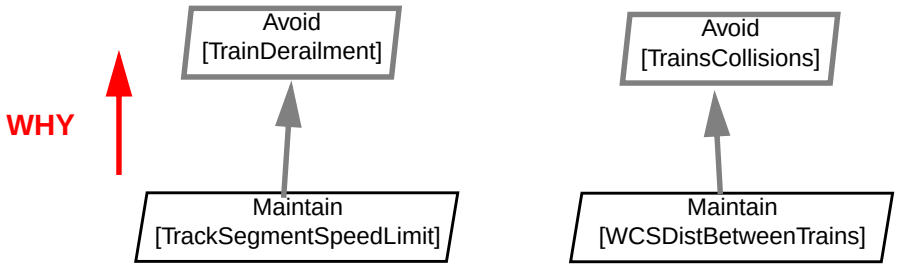


FIGURE 9.37. Asking WHY questions for the goals Maintain[TrackSegmentSpeedLimit] and Maintain[WCSDistBetweenTrains]

9. 2. 3. Refining goals and identifying alternative responsibility assignments

We now illustrate the use of agent-driven tactics for refining the goal Maintain[WCSDistBetweenTrains] until all leaf goals are realizable by single agents. Figure 9.38 gives an overview of the goal refinement graph and responsibility assignments that will be generated.

The refinement graphs for the other safety goals, Avoid[TrainEnteringClosedGate] and Maintain[TrackSegmentSpeedLimit], can be generated in a similar way.

1. Split lack of control by cases

The goal Maintain[WCSDistBetweenTrains] constrains the speed and location of the following train based on the location of the preceding train. It is therefore not realizable by the TrainControlSystem agent, because it lacks of monitorability and control for the actual speed and location of trains.

In order to resolve such realizability problem, the goal is first split by cases according to whether in the previous state the two trains are already on the same track or not. The tactic split lack of control by cases is therefore used to generate the subgoals:

Maintain[WCSDistBetweenTrainsOnSameTrack]

Avoid[ViolationOfWCSDistWhenTrainEnteringTrack]

The second subgoal is further split by cases according to whether it is the preceding train or the following train that enters the common track, yielding the two goals:

Avoid[TrainEnteringTrackInFrontOfCloseTrain]

Avoid[TrainEnteringTrackBehingCloseTrain]

The figures below show two states of the system from which each of these two goals could be violated. In state *A*, the train tr2 is about the enter the track in front of tr1, thereby violating the goal Avoid[TrainEnteringTrackInFrontOfCloseTrain]; in state *B*, the train tr1 is about the enter the track behind tr2, thereby violating the goal Avoid[TrainEnteringTrackBehingCloseTrain]. Although the initial document is silent about this, the satisfaction of the first subgoal is probably taken care of by the interlocking system that manages track switches and associated signals. For the second subgoal, it is less clear whether the train control system may rely on the interlocking system to satisfy that goal.

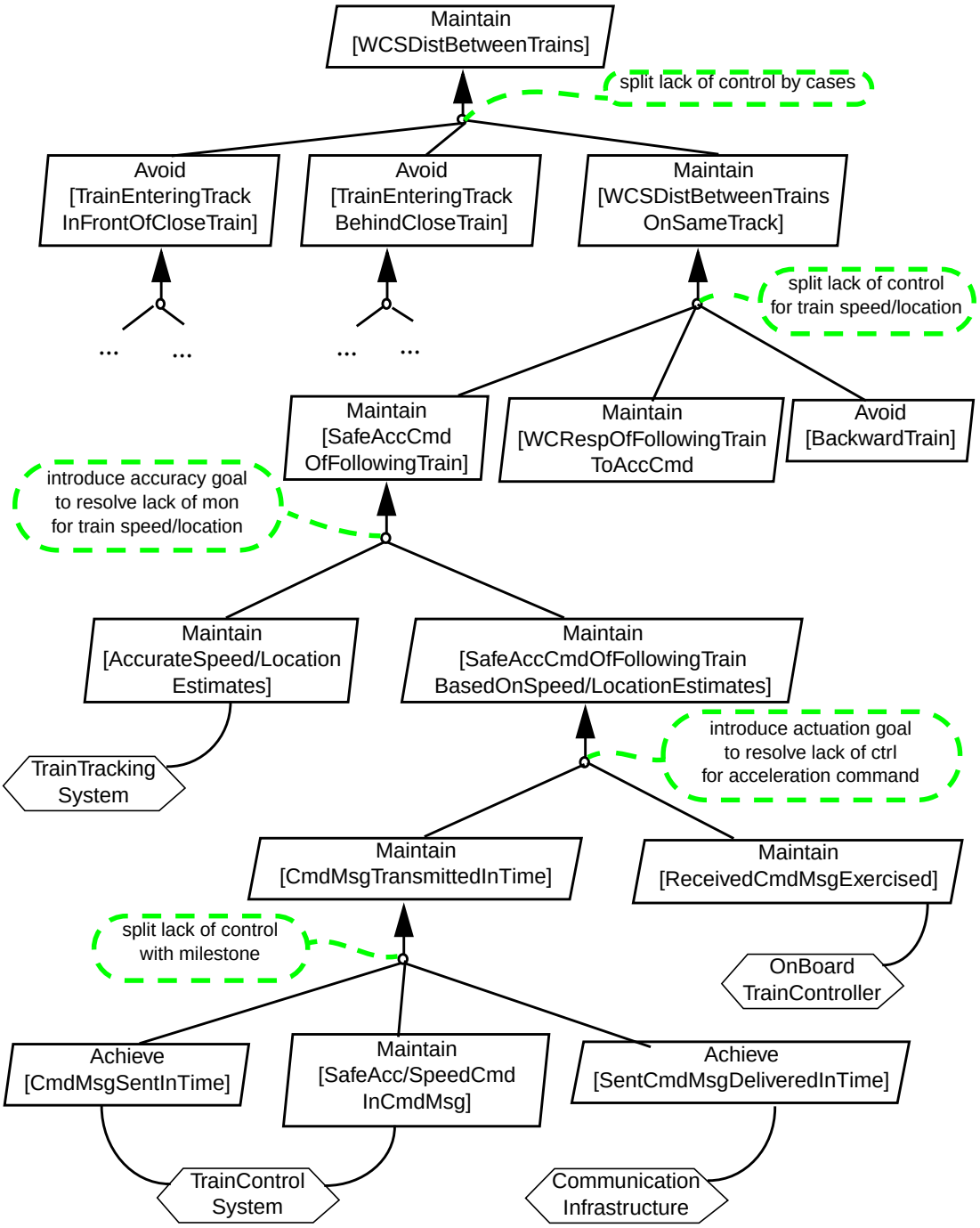
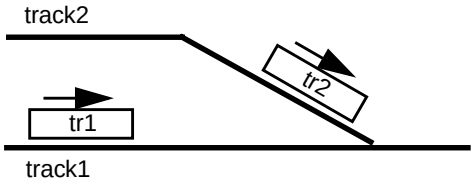
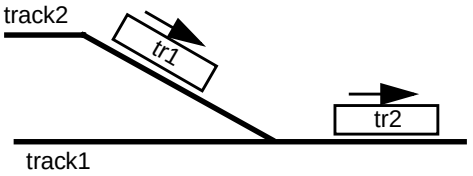


FIGURE 9.38. Goal refinement graph and responsibility assignments for the goal Maintain[WCSDistBetweenTrains]



A. train entering track in front of close train



B. train entering track behind close train

These two goals may be formally defined as follows:

Goal Avoid[TrainEnteringTrackInFrontOfCloseTrain]
Definition A train should not enter a track in front of another train if it violates the worst case stopping distance between the two trains.
FormalDef $\forall$ tr2: Train, track: Track
@ OnTrack(tr2, track)
$\Rightarrow \neg$ ($\exists$ tr1: Train):
tr1$\neq$tr2 $\wedge$ OnTrack(tr1, track) $\wedge$ tr2.Loc $\geq$ tr1.Loc $\wedge$ tr2.Loc - tr1.Loc $\geq$ tr1.WCSDist

Goal Avoid[TrainEnteringTrackBehingCloseTrain]
Definition A train should not enter a track behind another train if it violates the worst case stopping distance between the two trains.
FormalDef $\forall$ tr1: Train, track: Track
@ OnTrack(tr1, track)
$\Rightarrow \neg$ ($\exists$ tr2: Train):
tr1$\neq$tr2 $\wedge$ OnTrack(tr2, track) $\wedge$ tr2.Loc $\geq$ tr1.Loc $\wedge$ tr2.Loc - tr1.Loc $\geq$ tr1.WCSDist

The goal Maintain[WCSDistBetweenTrainsOnSameTrack] is defined formally as follows:

Goal Maintain[WCSDistBetweenTrainsOnSameTrack]
Definition If a train is following another so that the distance between the two trains is safe, then the distance between the two trains must remain safe in the next state.
FormalDef $\forall$ tr1, tr2: Train, track: Track
$\bullet$ (Following(tr1,tr2) $\wedge$ tr2.Loc - tr1.Loc $\geq$ tr1.WCSDist)
$\wedge$ Following(tr1, tr2)
$\Rightarrow$
tr2.Loc - tr1.Loc $\geq$ tr1.WCSDist

This goal is further refined in the following sections.

2. split lack of control for train location

The above goal

Maintain[WCSDistBetweenTrainsOnSameTrack]

constrain the speed and location of the following train based on the location of the preceding train. It is unrealizable by the centralized TrainControlSystem agent because it lacks of control for the speed and location of the following train, and lack of monitorability for the location of both trains.

The tactic split lack of control is used to resolve lack of control for the acceleration of the following train, by identifying the new intermediate attribute Train.AccCmd that denotes the acceleration command controlled by the OnBoardTrainController[1], and generating the following subgoals:

Maintain[SafeAccCmdOfFollowingTrain]

Maintain[WCRespOfFollowingTrainToAccCmd]

Avoid[BackwardTrain]

The first subgoal defines an upper bound on the acceleration command of the following train. In order to define such upper bound, it is necessary to make some assumptions about the behaviours of the preceding and following trains. The second subgoal is a worst case assumption that relates the physical acceleration of the following train to its acceleration command. The third subgoal is a worst case assumption on the behaviour of the preceding train, i.e. it may not go backward (but could suddenly stop).

A definition of the worst case assumption on the following train will look something like this:

$$\begin{aligned} & tr.AccCmd \geq 0 \Rightarrow tr.Acc \leq tr.AccCmd \\ & \wedge \blacksquare_{\leq MCDelay}\ tr.AccCmd < 0 \Rightarrow tr.Acc < 0 \\ & \wedge\ tr.AccCmd < 0 \wedge \neg \blacksquare_{\leq MCDelay}\ tr.AccCmd < 0 \Rightarrow tr.Acc \leq \bullet\ tr.Acc \end{aligned}$$

This definition says that: (i) if the acceleration command of the train is positive, the actual acceleration of the train is less than the commanded acceleration, (ii) if the acceleration command of the train has been negative during the last MCDelay time units, where MCDelay is the delay needed to go into braking mode, the train is actually decelerating, and (iii) if the acceleration command is negative but has not been negative for the less MCDelay time units, we assume that the acceleration of the train is not increasing. This definition is given here as an example. Further elicitation from and validation by domain experts are required to define that assumption more accurately. It is important to note that the definition of that assumption is not intended to capture the exact relation between the actual acceleration of a train and the acceleration command of the train, but only some worst case assumption on those quantities.

The third goal defines a worst case assumption on the preceding train. The initial document says that collisions between trains has to be avoided even if the preceding train stopped suddenly (e.g. because of derailment). Therefore, the worst case assumption for the preceding train is that it does not go backward:

$$\square\ (tr2.Loc \geq \bullet\ tr2.Loc)$$

1. Note that the acceleration command denotes here a quantity controlled by the OnBoardTrainController. It correspond to the state of the acceleration and braking commands of the train. It should not be confused with the acceleration command in a command message sent by the TrainControlSystem to the OnBoardTrainController.

Finally, the formal definition of the goal Maintain[SafeAccCmdOfFollowingTrain] looks like this:

Goal Maintain[SafeAccCmdOfFollowingTrain]
Definition The acceleration command of a train tr1 should be less than
F(tr1.Loc, tr2.Loc, tr1.Speed),
where tr2 is the train preceding tr1. The value of that function is calculated so that it ensures a worst case distance is maintained between the two trains.
FormalDef $\forall$ tr1, tr2: Train
● Following(tr1,tr2)
$\Rightarrow$
tr.AccCmd $\leq$ F(tr1.Loc, tr2.Loc, tr1.Speed)

In this goal, the function F(tr1.Loc, tr2.Loc, tr1.Speed) defines an upper bound for the acceleration command of the train. It must be calculated so that the parent goal, Maintain[WCSDistBetweenTrainsOnSameTrack], is satisfied provided that the two other assumptions in the goal refinement are satisfied. This function is the solution of a differential equation. We assume that such solution can be given by domain experts.

This goal is further refined in the next section.

3. Resolve lack of monitorability for trains' speed and location

The goal

Maintain[SafeAccCmdOfFollowingTrain]

constrains the commanded acceleration of a train based on the speed and position of that train and on the position of the preceding train. This goal is unrealizable by the TrainControlSystem, because it lacks monitorability for the trains' speed and locations.

The tactic introduce tracking object is applied to resolve such lack of monitorability by generating the subgoals:

Maintain[AccurateSpeed/LocationEstimates]

Maintain[SafeAccCmdBasedOnSpeed/LocationEstimates]

The second subgoal is an accuracy goal that relates the actual speed and locations of trains to the estimated speed and locations of trains. That goal is assigned as the responsibility of a TrainTrackingSystem agent mentioned in the initial document. The second goal constrain the value of the acceleration command of the following train based on the position estimates of the following and preceding train, and on the speed estimate of the following train.

These goals refer to the new intermediate object TrainInfo that denotes informations about trains speed and location. At this stage, such object is defined as follows:

Entity TrainInfo
Has
TrainID: TrainID_Domain
TrackId: TrackID_Domain
the track on which the position of the train is given
Loc: LocationUnit
the mean location estimate of the train
LocDev: LocationUnit
the standard deviation from the mean location
Speed: SpeedUnit
the mean speed estimate of the train
SpeedDev: SpeedUnit
the standard deviation from the mean speed

The formal definition of the subgoals generated by the tactic are given by:

Goal Maintain[AccurateSpeed/LocationEstimates]
Definition The actual speed and location of a train should be within the mean and standard deviation of the speed and position estimates of that train.
FormalDef $\forall$ tr: Train, ti TrainInfo
Tracking(ti, tr)
$\Rightarrow$
($\forall$ track: Track, track_id: TrackID_Domain):
(ti.TrackId = track_id $\leftrightarrow$ OnTrack(tr, track) $\wedge$ track.TrackID = track_id)
$\wedge$ ti.Loc - ti.LocDev $\leq$ tr.Loc $\leq$ ti.Loc + ti.LocDev
$\wedge$ ti.Speed - ti.SpeedDev $\leq$ tr.Speed $\leq$ ti.Speed + ti.SpeedDev

Goal Maintain[SafeAccCmdBasedOnTrainSpeed/PositionEstimates]
Definition The acceleration command of a train tr1 should be less than
F(ti1.Loc + ti1.LocDev, ti2.Loc - ti2.LocDev, ti1.Speed + ti2.SpeedDev)
where ti1 denotes the object tracking tr1 and ti2 denotes the object tracking the preceding train.
FormalDef $\forall$ tr1, tr2: Train, ti1, ti2: TrainInfo
Tracking(ti1, tr1) $\wedge$ Tracking(ti2, tr2)
$\wedge$ $\bullet$ FollowingInfo(ti1,ti2)
$\Rightarrow$
tr1.AccCmd $\leq$ F(ti1.Loc + ti1.LocDev, ti2.Loc - ti2.LocDev, ti1.Speed + ti2.SpeedDev)

Note that the definition of the last subgoal assumes the worst-case for the speed/position estimates. For the following train, the worst-case speed and position are respectively ti1.Speed + ti1.SpeedDev and ti1.Loc + ti1.LocDev. For the preceding train, the worst-case position is ti2.Loc - ti2.LocDev.

These goal definitions are first-sketch approximations of the actual goals of the BART system. In the actual system, the TrainTrackingSystem issues speed and location estimates every 1/2 seconds. Further work is therefore required to define those goals more precisely.

The above goal refinement is also based on the following goal requiring that every train is tracked by exactly one TrainInfo instance, and that the Tracking relationship is static:

$\forall$ tr: Train $\square$ $\exists!$ ti: Tracking(ti, tr)
($\forall$ tr: Train, ti: TrainInfo): Tracking(ti, tr) $\Rightarrow$ $\square$ Tracking(ti, tr)

4. Resolve lack of control for the acceleration command of trains

The goal

Maintain[SafeAccCmdBasedOnSpeed/LocationEstimates]

is still unrealizable by the TrainControlSystem because it lacks of control for the acceleration command of the train. (the acceleration command is a quantity controlled by the OnBoardTrainController, and that is used by that agent to command the actual acceleration of the train.) This goal is also unrealizable by the OnBoardTrainController because it lacks of monitorability for the TrainInfo entities.

At this point, alternative system design are generated through alternative applications of agent-driven tactics.

A First Centralized Design

In a first alternative, the tactic introduce actuation goal is used to resolve lack of control of the TrainControlSystem agent for the acceleration command of the train, by generating the subgoals:

Maintain[SafeCmdMsgReceivedInTime]

Maintain[ReceivedCmdMsgExercised]

The first goal requires that every 1/2 second, the OnBoardTrainController receives a command message from the TrainControlSystem. (The 1/2 second delay requirement is taken from the initial case study description. It corresponds to the frequency at which speed/location estimates of trains are received by the TrainControlSystem.) The command message has an acceleration command attribute that defines the acceleration command to be applied by the OnBoardTrainController. The second subgoal is an actuation goal assigned to the OnBoardTrainController. It requires the acceleration command of the train to be equal to the value of the acceleration command contained in the last received command message.

Note that in this alternative, the acceleration command of the train is fully calculated by the centralized TrainControlSystem. This alternative is not exactly the one chosen for the BART system. The actual design for the BART system will be discussed later.

The first subgoal is given the following first-sketch definition:

Goal Maintain[SafeCmdMsgReceivedInTime]
Definition The OnBoardTrainController should receive a command message from the TrainControlSystem every 1/2 second. The value of the acceleration command in the command message must be safe, i.e. it should be less than
G(ti1.Loc + ti1.LocDev, ti2.Loc - ti2.LocDev, ti1.Speed + ti2.SpeedDev)
where attributes of ti1 denote speed/position estimates about the following train and attributes about ti2 denote speed/position estimates about the preceding train.
FormalDef $\forall$ tr1, tr2: Train, ti1:TrainInfo
Tracking(ti1, tr1)
● FollowingInfo(ti1,ti2)
$\Rightarrow \Diamond_{\leq 1/2sec}$ ($\exists$ cmd_msg: CommandMessage):
Received(cmd_msg, tr)
$\wedge$ cmd_msg.AccCmd $\leq$ G(ti1.Loc + ti1.LocDev, ti2.Loc - ti2.LocDev, ti1.Speed + ti2.SpeedDev)

Note that the upper bound on the acceleration command has been strenghtened to G(ti1, ti2) (where G(ti1, ti2) ≤ F(ti1, ti2)) to take into account the 1/2 second delay in the reception of the command message. The function G has to be calculated by making some worst case assumption on the behaviour of the following train during this 1/2 second delay.

The second subgoal is defined as follows:

Goal Maintain[ReceivedCmdMsgExercised]
Definition The acceleration command of the train should be equal to the acceleration command of the last received command message.
FormalDef ∀ tr: Train, cmd_msg: CommandMessage
Received(cmd_msg, tr)
∧ ¬ (∃ cmd_msg': CommandMessage): cmd_msg' ≠ cmd_msg
∧(Received(cmd_msg', tr) S Received(cmd_msg, tr))
⇒
tr.AccCmd = cmd_msg.AccCmd

In order to reach subgoals assignable to single agents, the first goal in this alternative is further refined as follows:

Maintain[SafeCmdMsgReceivedInTime]

← Achieve[CmdMsgSentInTime]

← Maintain[SafeAccCmdInCmdMsg]

← Maintain[SentCmdMsgDeliveredInTime]

The first and second subgoals are assigned as the responsibility of the TrainControlSystem agent; the third is assigned as the responsibility of a CommuncationInfrastructure agent.

A Fully Distributed Design

An alternative design can be generated by applying the tactic introduce accuracy goal to resolve lack of monitorability of the OnBoardTrainController for the TrainInfo objects. An application of this tactic refines the goal Maintain[SafeAccCmdBasedOnSpeed/LocationEstimates] into the subgoals:

Maintain[TrainInfoMsgDeliveredInTime]

Maintain[SafeAccCmdBasedOnTrainInfoMsg]

The first subgoal requires that the OnBoardTrainController of a given train receives periodically information messages from the TrainControlSystem (or directly from the TrainTrackingSystem agent). These information messages contain informations about the position of the train and the position of its preceding train. (Information about the speed of the train is not necessary as it can be obtained by the OnBoardTrainController.) The second subgoal is assigned as the responsibility of the OnBoardTrainController. It constrain the value of the acceleration command of the train based on the informations contained in the last received train information message.

This alternative correspond to a fully distributed system in which the acceleration command of trains are calculated by the OnBoardTrainController's rather than by the centralized TrainControlSystem.

The BART Design

The design for the BART system described in the initial document is based on yet another alternative. That design shows an interesting split of responsibilities between the centralized TrainControlSystem and the OnBoardTrainController: the acceleration command of a train is partly calculated by the centralized TrainControlSystem and partly by the OnBoardTrainController's.

In this design, the command messages sent to OnBoardTrainController's contains both a commanded acceleration and a commanded speed. The commanded speed is used by the OnBoardTrainController to ensure a smooth acceleration of the train toward that speed, and to minimize mode changes when the train seeks to maintain that speed[1]. A motivation for this design is therefore that it contributes to the goal SmoothMovement in Figure 9.35.

The commanded speed is also used in the BART system as a way to strengthen the safety of the system. The commanded speed is used as an upper bound of the actual speed when calculating an acceleration command that ensure a worst case stopping distance between the trains. In order for this to be safe, the commanded speed should always be bigger than the actual speed of the train. (When deceleration is commanded, the commanded speed should still be bigger than the actual speed, and the commanded speed is not used by the OnBoardTrainController.)

The goal

Maintain[SafeAccCmdBasedOnSpeed/LocationEstimates]

is therefore refined as follows:

← Maintain[SafeCmdMsgReceivedInTime]
  ← Achieve[CmdMsgSentInTime]
  ← Maintain[SafeAcc/SpeedCmdInCmdMsg]
  ← Maintain[SentCmdMsgDeliveredInTime]
← Maintain[ReceivedCmdMsgExercised]

In this alternative, the goal Maintain[ReceivedCmdMsgExercised] defines the behaviour of the OnBoardTrainController in response to the speed and acceleration commands in the received command messages; the general form for the goal Maintain[SafeAcc/SpeedCmdInCmdMsg] will look something like this:

Goal Maintain[SafeAcc/SpeedCmdInCmdMsg]
FormalDef $\forall$ cmd_msg: CommandMessage, ti1, ti2: TrainInfo
cmd_msg,.Sent $\wedge$ cmd_msg.TrainId = ti1.TrainId
$\wedge$ ● FollowingInfo(ti1, ti2)
$\Rightarrow$
cmd_msg.AccCmd $\leq$ F(ti1, ti2)
$\wedge$ cmd_msg.SpeedCmd $\leq$ G(ti1, ti2)
$\wedge$ cmd_msg.SpeedCmd $>$ ti1.Speed + ti1.SpeedDev

1. When given a positive acceleration command, the OnBoardTrainController will accelerate the train at the given rate until the actual speed is within 7 mph of the commanded speed. When the actual speed is within 7 mph of the commanded speed, the OnBoardTrainController will limit its acceleration to smoothly reach a speed at 2 mph below the commanded speed. Once a speed at 2 mph below the commanded speed is reached, the OnBoardTrainController will try to maintain that speed, although there may be some small fluctuations.

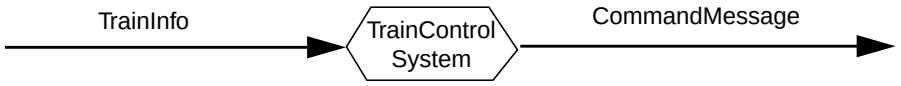
In our model, a single instance of the TrainControlSystem agent is responsible for issuing safe command messages for all trains. In the actual BART design, the handling of trains is split among multiple instances of the TrainControlSystem agent. (This design can be generated by splitting the goal Maintain[SafeCmdMsgReceivedInTime] into cases). Decisions about the number of instances of TrainControlSystem agent, the number of trains to be handled by each instance, and the required delays for sending command messages are fundamental design decisions that have critical impacts on the performance, safety and costs of the system. These decisions are at the boundary between requirements engineering and architectural designs. Techniques such as those of the Architecture TradeOffs Analysis Method [Kaz99] are relevant to make such decisions and record their rationale.

9. 2. 4. Goal Operationalization

Agents interfaces and operational requirements are now derived from the terminal goals generated during the goal refinement process. Consider for instance the above terminal goal

Maintain[SafeAcc/SpeedCmdInCmdMsg]

assigned as the responsibility of the TrainControlSystem agent. The portion of the agent interface model derived from that responsibility assignment is given by:



From the formal definition of that goal, we also derive the following operation to be performed by the TrainControlSystem agent, together with the required post conditions that guarantees the satisfaction of the goal:

Operation SendCommandMessage
PerfBy TrainControlSystem
Input TrainInfo {**arg** ti1, ti2}
Output CommandMessage {**res** cmd_message}
Dompre ¬ cmd_msg.Sent
DomPost cmd_msg.Sent ∧ cmd_msg.TrainId = ti1.TrainId
ReqPostFor Maintain[SafeAcc/SpeedCmdInCmdMsg]
● FollowingInfo(ti1, ti2)
→
cmd_msg.AccCmd ≤ F(ti1, ti2)
∧ cmd_msg.SpeedCmd ≤ G(ti1, ti2)
∧ cmd_msg.SpeedCmd > ti1.Speed + ti1.SpeedDev

From the formal definition of the goal Achieve[CmdMsgSentInTime], we derive a further required trigger condition on that operation:

ReqTrigFor Achieve[CmdMsgSentInTime]
$\blacksquare_{\leq 1/2\ sec}$ ¬ (∃ cm2: CommandMessage): cm2.Sent ∧ cm2.TrainID = ti1.TrainId

Further operations and agent interfaces are derived from other responsibility assignments. Figure 9.39 shows the agent interface model derived from the responsibility assignments in Figure 9.38.

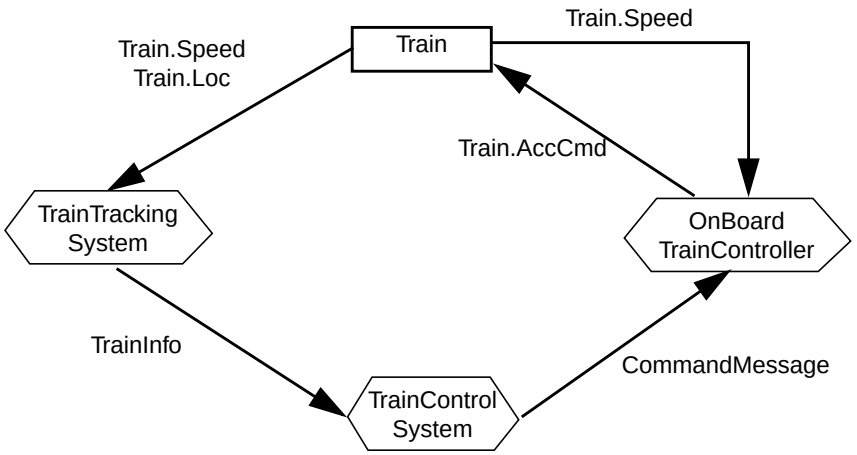


FIGURE 9.39. Portion of the agent interface model derived from the responsibility assignments in Figure 9.38

9. 2. 5. Obstacle Analysis

We now apply obstacle identification techniques to generate obstacles for the terminal goals and assumptions generated during the goal refinement process. Several of the generated obstacles correspond to problems that are anticipated in the initial case study description; other obstacles correspond to problems that are not raised in this document.

Next, we apply obstacle resolution tactics to generate alternative resolutions for the generated obstacles. Several of the new requirements generated by these tactics correspond to features of the BART system mentioned in the initial document. We also generate alternative requirements corresponding to alternative ways to resolve obstacles anticipated in the initial document; and further new requirements to tolerate obstacles that are not mentioned in this document.

9. 2. 5. 1. Generating Obstacles

Obstacles were identified by applying a mix of formal techniques and heuristics from Chapter 8. For example, consider the assumption Maintain[WCRespOfFollowingTrainToAccCmd] appearing in Figure 9.38:

Assumption Maintain[WCRespOfFollowingTrainToAccCmd]
FormalDef $\forall$ tr: Train
$tr.AccCmd \geq 0 \Rightarrow tr.Acc \leq tr.AccCmd$
$\wedge \blacksquare_{\leq MCDelay}\ tr.AccCmd < 0 \Rightarrow tr.Acc < 0$
$\wedge\ tr.AccCmd < 0 \wedge \neg \blacksquare_{\leq MCDelay}\ tr.AccCmd < 0 \Rightarrow tr.Acc \leq \bullet\ tr.Acc$

The negation of this assumption yield the obstacle BadRespOfTrainToAccCmd, which is then OR- refined into the subobstacles:

Obstacle ExcessiveAccelerationInRespToAccCmd
FormalDef $\Diamond \exists$ tr: Train
(tr.AccCmd $\wedge$ tr.Acc > tr.AccCmd)
$\vee$
tr.AccCmd < 0 $\wedge \neg \blacksquare_{\leq MCDelay}$ tr.AccCmd < 0 $\wedge$ tr.Acc > $\bullet$ tr.Acc

Obstacle BadBraking
FormalDef $\Diamond \exists$ tr: Train
$\blacksquare_{\leq MCDelay}$ tr.AccCmd < 0 $\wedge$ tr.Acc $\geq$ 0

A further subobstacle of the obstacle ExcessiveAccelerationInRespToAccCmd is mentioned in the initial case study description: since trains accelerometers cannot sense acceleration due to gravity, the actual acceleration of a train on a downgrade will be bigger than commanded. Therefore the condition AcceleratingTrainOnDownGrade is a subobstacle of ExcessiveAccelerationInRespToAccCmd. Another subobstacle of ExcessiveAccelerationInRespToAccCmd is that the acceleration command of the train is malfunctioning. The following obstacle refinement is thereby obtained:

ExcessiveAccelerationInRespToAccCmd
  ← AcceleratingTrainOnDownGrade
  ← AccelerationCommanddMalfunction

From the formal definition of the obstacle BadBraking, we also generate the following obstacle refinement graph:

← BadBraking
  ← TrainDoesNotReconfigureForBraking
  ← ExcessiveModeChangeDelay
  ← NoBrakingInBrakingMode

Other obstacles generated from the terminal goals and assumptions of the goal refinement graphs in Figure 9.38 are summarized in Table 9.4.

Table 9.4: Obstacles to subgoals of the goal *Maintain [WCSDistBetweenTrainsOnSameTrack]*

agent	goal/assumption	obstacles
--	Avoid [BackwardTrain]	BackwardTrain
--	WCRespOfFollowingTrain ToAccCmd	BadRespOfTrainToAccCmd ← ExcessiveAccelerationInRespToAccCmd ← AcceleratingTrainOnDownGrade ← AccelerationCommanddMalfunction ← BadBraking ← TrainDoesNotReconfigureForBraking ← ExcessiveModeChangeDelay ← NoBrakingInBrakingMode
TrainTracking System	AccurateSpeed/Location Estimates	InaccurateSpeed/LocationEstimates ← NoSpeed/PositionEstimates ← OutOfDateSpeed/LocationEstimates ← InaccurateRecentSpeed/LocationEstimates ← ImpossibleChangeInSpeed/LocationEstimates ← PlausibleInaccurateSpeed/PositionEstimates
OnBoard TrainController	ReceivedCmdMsgExer-cised	BadRespToCmdMsg ← CmdMsgIgnored ← ExcessiveAccelerationCommanded ByOnBoradtrainController

agent	goal/assumption	obstacles
TrainControl System	CmdMsgSentInTime	CmdMsgNotSentInTime ← CmdMsgNotSent ← CmdMsgSentLate ← CmdMsgSentToWrongTrain
	SafeAcc/SpeedCmd InCmdMsg	UnsafeCmdMsg ← UnsafeAccInCmdMsg ← CmdSpeedInCmdMsgExceedsSpeedEstimate
Communic. Infrastructure	SentCmdMsg DeliveredInTime	SentCmdMsgNotDeliveredInTime ← SentCmdMsgNotDelivered ← SentCmdMsgDeliveredLate ← SentCmdMsgDeliveredToWrongTrain ← DeliveredCmdMsgCorrupted ← DeliveredCmdMsgWithInvalidSpeed/AccCmd ← DeliveredCmdMsgWithCorruptedValidSpeed/ AccCmd

9. 2. 5. 2. Resolving Obstacles

For each generated obstacles, one should identify alternative obstacle resolutions proposals for preventing, reducing or tolerating the obstacle. In the sequel, we illustrate various obstacle resolution tactics applied to the BART system. These tactics generate further goals and requirements, some of which correspond to fault tolerance requirements described in the initial document. Other goals generated by our tactics correspond to alternative resolutions for the obstacles that are anticipated in the initial document, or to resolutions of obstacles not covered in that document.

1. Goal Substitution

As an example of goal substitution, consider the obstacle CmdMsgSentLate that obstructs the goal CmdMsgSentInTime under the responsibility of the TrainControlSystem agent. This obstacle occurs when the centralized system is not able to send command messages in time for every trains.

One possible resolution for this obstacle is generated by applying the tactic choose alternative goal. It consists in eliminating the problem thoroughly by choosing the alternative fully distributed design in which command acceleration are computed by the OnBoardTrainController's instead of the centralized TrainControlSystem.

2. Obstacle Prevention

As an example of obstacle prevention, consider the obstacle UnsafeCmdMsg obstructing the goal Maintain[SafeAcc/SpeedCmdInCmdMsg] assigned as the responsibility of the TrainControlSystem. An application of the tactic prevent obstacle generate the new goal Avoid[UnsafeCmdMsgSent]. In the actual BART system, this new goal is assigned as the responsibility of a Vital Station Computer agent. This new goal introduces a form of redundancy: the (non-vital) TrainControlSystem agent is responsible for issuing safe command messages in time, the Vital Station Computer agent is responsible for blocking command messages issued by the TrainControlSystem if these messages contain unsafe speed/acceleration commands. Note that the TrainControlSystem is to be operated on hardware that is reliable enough to meet performance related goals, but not safety-related goals; whereas the Vital Station Computer agent is slower but reliable enough to meet safety-related goals.

As another example of obstacle prevention, consider the obstacle ImpossibleChangeInSpeed/LocationEstimates that obstructs the goal Maintain[AccurateSpeed/PositionEstimates]. That obstacle covers situations in which changes in speed position estimates correspond to movement that are known to be physically impossible (for instance, a train is on a given track at one instant, and on a different unrelated track at the next instant.) Such obstacle can be resolved by applying the tactic prevent obstacle which generate the new goal

Avoid[ImpossibleChangeInSpeed/LocationEstimates]

which could be assigned as the responsibility of the TrainTrackingSystem. This new goal corresponds to a standard technique in safety critical system for verifying that validity of the inputs of the system with respect to possible values in the environment. Alternatively, the same obstacle could be resolved by applying the tactic mitigate obstacle and generating the new goal

Avoid[CmdMsgBasedOnImpossibleChangeInSpeed/LocationEstimates]

which could be assigned as the responsibility of the TrainControlSystem agent. In this alternative, the TrainControlSystem is responsible for not issuing a command message if the train speed position estimates are impossible in the domain.

3. Goal Deidealization

As an example of deidealization, consider the obstacle AcceleratingTrainOnDownGrade obstructing the assumption Maintain[WCRespOfFollowingTrainToAccCmd]. In this case, the obstacle may be resolved by deidealizing the assumption so that it takes into account the acceleration due to the grade of the track. Such deidealization will then be propagated to the other goals in Figure 9.38. It will ultimately result in a modification of the goal Maintain[SafeAcc/SpeedCmdInCmdMsg]], in which the acceleration in command messages will have to take into account the grade of the track segment of the following train.

4. Obstacle Mitigation

As an example of obstacle mitigation, consider the obstacles SentCmdMsgNotDelivered and SentCmdMsgDelivedLate that obstructs the goal Maintain[SentCmdMsgDeliveredInTime]. The tactic mitigate obstacle can here be used to generate the new goal

Avoid[TrainsCollisionsWhenCmdMsgNotTransmittedInTime].

This new goal ensures the higher level safety goal Avoid[TrainsCollisions]. This new goal is also used to mitigate the other obstacle CmdMsgNotSent and CmdMsgSentLate that obstructs the goal Achieve[CmdMsgSentInTime] assigned to the TrainControlSystem agent.

The newly generated goal can then be refined into subgoals that are realizable by single agents:

Avoid[TrainsCollisionsWhenCmdMsgNotTransmittedInTime]
← FullBrakingWhenNoCmdMsgReceivedDuring2Seconds
← FullBrakingTrainStoppedWithinWCSDist

The second assertion in this refinement is a domain property characterizing the worst-case stopping distance of a train. To satisfy that domain property, the worst case stopping distance is calculated by taking into account the distance that the train will travel during the 2 second delay before applying the emergency brake.

Note that this delay of 2 seconds is an important design decision that is not documented in the initial case study description. Choosing a shorter delay (e.g. 1 sec) would allow for a shorter distance between trains, but may result in more frequent emergency braking when command messages are not transmitted in time. The probability that a train will not receive a command message in time should therefore be used to choose the most appropriate trade-off for this delay.

The other obstacle OutOfDateSpeed/LocationEstimates, obstructing the goal AccurateSpeed/LocationEstimates assigned to the TrainTrackingSystem agent, is resolved similarly by applying the tactic mitigate obstacle to generate the new goal

Avoid[TrainsCollisionsWhenOutOfDateSpeed/LocationEstimates].

This goal is refined in turns:

Avoid[TrainsCollisionsWhenOutOfDateSpeed/LocationEstimates].
← FullBrakingWhenOutOfDateTrainInfo
← FullBrakingWhenMOTTinCmdMsgExpired
← AccurateMOTTinCmdMsg
← FullBrakingTrainStoppedWithinWCSDist

This strategy allows one to derive new requirements and the Message Origination Time Tag (MOTT) attribute mentioned in the initial case study description and attached to TrainInfo and CommandMessage entities.

9. 3. Discussion

Our experience in using the KAOS goal-oriented requirement elaboration method and the techniques presented in the thesis for the LAS, the BART and other systems revealed a number of issues that are worth pointing out. Future work required by some these issues is discussed in the concluding chapter.

Eliciting and formalizing high-level goals

- The goal-oriented requirements elaboration method relies on the *identification of high-level goals*. For the LAS and the BART case studies, such goals were easily identified as they were explicitly defined in the initial documents.

- The method also relies on the *early formalization of goals and assumptions* in the environment. Such formalization allows for the systematic derivation of objects from goal definitions, the verification of the correctness of goal refinements, the detection of conflicts between goals and the generation of obstacles from goals and assumptions. Such early formalization of high-level goals and assumptions is desirable and feasible. High-level goals and assumptions about the environment are actually easier to define (both formally and informally) than lower-level goals and requirements expressed at the interface of the software. The reason for this is obvious; higher-level goals are more abstract properties that are not concerned with the details of the

requirements at the interface of the software. In the LAS case study, for example, the definition of a high-level goal such as Achieve[AmbulanceIntervetion] is simple and given in the initial problem statement. The definition of requirements assigned to the CAD software agent involves many more details, and would be quite hard to specify formally without constructive guidance.

- The availability of obstacle analysis techniques was felt to be essential to the practicality of the goal-oriented requirements elaboration method. It allows one to *start from idealized definitions of goals and assumptions*, and to consider more realistic definitions later in the requirements elaboration process. Without obstacle analysis, writing realistic goal definitions seems very hard, if at all possible. Considering idealized goal definitions first is also essential as it allows one to avoid premature, implicit, and possibly not optimal compromises about the required behaviour of the system (see Section 3.3.3.2 of Chapter 3).

Agent-Driven Tactics: Benefits and Issues

- The systematic identification and resolution of violations of the realizability meta-constraint was felt to provide useful, practical guidance for elaborating goal refinements graphs and responsibility assignments. Although it should be further enriched, the library of agent-driven tactics of Chapter 6 already provides rich and detailed support for refining goals and identifying agents. We also observed that by applying these tactics systematically, we have been able to build fairly large goal refinement graphs significantly faster than before.

- Labelling the goal refinements with the tactics applied to produce them makes the goal graph *easier to understand*; each refinement step is motivated by the resolution of a realizability problem, and the tactic applied to produce this goal refinement describes how the problem is solved. Knowing the definition of a high-level goal, someone familiar with the library of agent-driven tactics can easily infer the definition of all its subgoal from the tactics used to produce them.

- Agent-driven tactics are no “silver bullets”. Much creative thinking and domain knowledge is required to generate alternative goal refinements and responsibility assignments from high-level goals. The tactics provide a way to organize such creative thinking; they do not generate automatically all possible alternative designs by themselves.

- *For a goal raising several realizability problems, it is not a priori clear in what order each of these problems should be solved*. During the actual elaboration of the requirements for the LAS and the BART system, we frequently switched the order in which agent-driven tactics were applied. This did not result in different requirements at the end the requirements elaboration process, but had an impact on the definition of intermediate goals and on the presentation of the goal graph. The order of application of agent-driven tactics that was finally chosen was mostly driven by the objective of making the goal graph and the formal definitions of goals as easy to understand as possible. The effort of producing a well-structured goal graph could be significantly reduced by a tool supporting such transformations.

- The *number of alternatives* that can be generated during the goal refinement and responsibility assignment process may become quite large. Techniques and tools to manage large numbers of alternatives are definitely required.

- Frequently, the *strict application of formal goal refinement patterns produces first-sketch formal definitions* that need to be adapted to fit the details of the particular application domain. This may limit the amount of support for the automated application of tactics.

- *Accuracy goals* were seen to play an important role in the two significant case studies we have worked on. The role of accuracy goals is often neglected in formal specifications. Further work is required to identify specific tactics for refining and operationalizing accuracy goals. The formal specification of and reasoning about accuracy goals involving *tolerances and delays* is also a difficult issue that requires further work. The work reported in [Smi2K] is an interesting first step in that direction.

- The *boundary between the activities of elaborating requirements and designing a software architecture* is known to be blurred. This was clearly seen during the elaboration of the requirements for the BART train control system; we generated several alternative responsibility assignments and agent interfaces that ranged from a fully distributed system in which the trains' accelerations are computed by a centralized train controller software agent, to a fully distributed system in which the trains' accelearations are computed by on-board train controllers. We need to investigate further how the techniques developed for requirements engineering and for software architectures can be integrated. In particular we should investigate how goal-oriented techniques can be used to systematically derive a software architecture from a set of functional and non functional requirements.

- The *boundary between requirements engineering and human-computer interaction* was also seen to be an issue. When elaborating requirements for the London Ambulance Service system, we identified alternative responsibility assignments for a goal such as

  Achieve[AmbulanceAllocationBasedOnIncidentFormAndAmbulanceInfo].

  The later is concerned with the allocation of ambulances based on incident and ambulance informations. These alternative responsibility assignments corresponded to (i) the previous paper-based system in which decisions about ambulance allocation were made by human resource allocators, (ii) a fully automated decision procedure for ambulance allocation, or (iii) to many various forms of cooperation between the automated system and the human resource allocator. The detailed specification of and reasoning about such alternatives was felt to fall outside the scope of what could be practically handled by the KAOS method; it was left to more specialized techniques of Human Computer Interaction design. Techniques are needed to integrate or bridge the gap between the results of a goal-oriented requirements elaboration process and the techniques developed in the area of Human Computer Interaction.

The real elaboration process and implications for tool support

- The elaboration process for the LAS and BART case studies described in this chapter is idealized. The actual elaboration process went as follows.

  - We started by quickly elaborating preliminary, incomplete and possibly inconsistent models through the rough application of the agent-driven tactics. During this phase, many details were left unspecified; preliminary formal definitions were written (they helped in deriving the object and guided the refinement of goals), but no attempt was made to have fully accurate definitions (sometimes, only propositional logic was used).
  - In a second phase, much effort was spent on filling in the details and polishing the models. This included finding better names for goals, objects and attributes; revising our preliminary definitions (formal and informal); restructuring the goal graph (for instance by applying agent-driven tactics in a different order as discussed above); identifying and correcting incomplete goal refinements; etc. During this phase, the manual propagation of changes throughout the model was a particularly tedious task.

The way we actually build requirements models has implications on the kind of automated support a requirements elaboration tool should offer.

- The usability of the editing facilities of the tool is a critical requirement. The tool should allow users to quickly elaborate requirements models that may be left incomplete and inconsistent; it should allow users to freely and easily switch between different activities of the method.
- The tool should help users in detecting and keeping track of various incompleteness and inconsistencies of the models. Such incompleteness and inconsistencies range from simple syntactical problems (such as inconsistent use of goal names or object names; vocabulary used in the definition of goals but not declared in the object model; etc.) to formal semantic problems (such as incomplete goal refinements, divergences between goals, etc.).
- The tool should guide the user in resolving the various incompleteness and inconsistency problems. For instance, it could help in propagating changes of goals and object names throughout the models; in applying formal goal refinement patterns to complete incomplete goal refinements [Dar95, Dar96]; in applying agent-driven tactics to generate goal refinements, etc.

In brief, we favor a tool based on the “inconsistency implies action” paradigm of the viewpoint-oriented software development framework [Fin94, Hun98]. A preliminary prototype tool, called GRAIL/KAOS, has been developed at CEDITI [Dar98]. This tool was not used for the case studies described in this chapter. Using such a tool may have helped in editing the requirements models and keeping track of various simple syntactical inconsistencies, such as inconsistent use of goal names, object names, etc.

The possibilities of automating the techniques described in the thesis are further discussed in the concluding chapter.

Obstacles analysis

- For a number of goals, obstacle identification only involved a small number of regression steps --sometimes it did not go further than just negating the goal. For example, the obstacle to the goal AccurateAmbulanceLocationInfo under responsibility of the AVLS agent was obtained just by negation; regressing this negation further would have required detailed knowledge about properties of this agent which were unavailable to us. In this case, further regression was anyway not necessary for obstacle resolution since it is not necessary to know why the AVLS might fail to locate ambulances accurately.

- Finer agent granularity requires goals to be refined further and thus allows more detailed obstacles to be derived. There is a trade-off here between the level of abstraction of the specification and the level of detail of obstacle analysis; the finer-grained the agents are, the more RE work is required, but the more detailed obstacle analysis will be.

- Deciding when to stop obstacle refinement is not always easy. The refinement process may be stopped when an adequate resolution can be selected among those generated; the risk and impact of the obstacle should become acceptable with respect to the cost for resolving it. More knowledge about the causes of the obstacle, that is, its subobstacles, may result in the generation of better resolutions.

- Domain-complete OR-refinement of obstacles as discussed in this chapter allows one to stop looking for alternative obstacles.

- It is often the case that a new goal is introduced to resolve *several* obstacles simultaneously; the new goal actually resolves an obstacle to some higher-level goal which might be obstructed by the many obstacles to its subgoals. For example, the new goal Avoid[InaccurateAmbAvailabilityInfo] may resolve both obstacles InaccurateAmbAvailabilityOnMDT and EncodedMDTAvailabilityNotTransmitted.
  This suggests an heuristics for resolution selection: favor resolution *R1* over *R2* if at similar cost *R1* resolves more obstacles than *R2*.

- It is often the case that an obstacle is resolved by the introduction of *several* new goals --e.g., a combination of reduction, mitigation, and restoration goals.

- Identifying all the goals obstructed by the same obstacle is necessary for assessing the impact of this obstacle and thereby for deciding on an appropriate resolution. To support this, a cause-effect graph could be built from the goal refinement graph, the obstacle refinement graph, and the obstruction relation.

- A specific combination of multiple obstacles may sometimes increase their individual effects. This was clearly the case during the two LAS failures. In such cases one should clearly favor resolutions that address such combinations.

- Identifying the implications of an obstacle resolution is a serious issue. A new goal introduced for resolution may resolve critical obstacle combinations; but it may also interfere with other goals in the goal graph. A new cycle of conflict analysis [Myl92, Lam98b] may therefore be required.

Chapter 10
Related Work

10.1. Agent Responsibility, Monitoring and Control

As mentioned in Chapter 2, our work has been significantly influence by the paradigm of Composite System Design described in [Fea87]. A composite system is a system of multiple interacting agents that collaborate to achieve some global behavior. The paper describes a requirement elaboration method in which requirements on the behaviour of individual agents are gradually derived from constraints on the behaviours of the global system. The method is based on a simple formal framework for modeling agents, agent interface and agent responsibilities. The formal model of agents described in Chapter 4 is similar to the framework of [Fea87]. The presentations of the two models are slightly different; [Fea87] describes the semantics of agent behaviours in terms of a generate-and-prune paradigm, while our model is described in terms of a more traditional notion of transition systems. Besides this difference of presentation, the important concept of agent responsibility for goals and its relationship to the operational model the agents are taken from [Fea87]. We have then added to the model of [Fea87] a precise definition of the realizability meta-constraint relating agent responsibility for goals to agent interfaces.

Formal techniques supporting the Composite System Design approach have been proposed in [Fic92]. These techniques consist in (i) identifying inconsistencies between operational specifications of individual agents and declarative specifications of global goals; and (ii) resolving the identified inconsistencies by transformation of the operational and declarative specifications. That paper describes a small core of specification elaboration operators capable of generating a whole range of composite system designs. These operators were discovered empirically by studying various existing composite systems and trying to rationally rederive their features. In contrast to that work, the method described in the thesis consists in identifying and resolving violations of the realizability consistency rule between goals and agent interfaces. This yields three advantages over the more operational approach of [Fic92]: (i) the elaboration of the goal model can be performed without dealing with the intricacies of operational models; (ii) the agent-driven tactics for resolving realizability problems are structured according to the kind of realizability problems to be solved, providing better *retrievability* of tactics; (iii) the complete taxonomy of realizability problems provides a map for the systematic identification of specification elaboration tactics, providing better *coverage* of the space of tactics than with an empirical identification of tactics only.

The work of Zave and Jackson on the foundation of RE [Jack95, Zav97] is also closely related to ours. In particular, their work discusses the importance of identifying which actions are controlled by the environment, which actions are controlled by the machine, and which actions of the environment are shared with the machine. This is clearly similar to our model of agent monitorability and control, with the difference that our model is based on states rather than actions and that we consider multi-agent systems instead of only two agents: the machine and the environment. [Zav97] also describes three proper-

ties that must be satisfied for a requirement to be implementable (realizable): (i) the requirement must not be stated in terms of actions that are not shared with the machine, (ii) the requirement must not constrain an action that is controlled by the environment, and (iii) the requirements must not be stated in terms of the future. The first condition can be checked directly from the syntax of the requirement. The second condition is defined in the context of Buchi automata. The third condition lacks a precise definition. However, these conditions are given as primitives, without an underlying definition of what is meant for a requirement to be "implementable". As a result, it is not clear whether the three stated properties are complete. (Can there be other reasons for which a requirement cannot be realized by the machine alone?) The work reported in Chapters 4 and 5 has the same objective as the work of Zave and Jackson. We aim at defining conditions that have to be met by a goal for the goal to be assignable to a single agent. For this purpose, we have first proposed a formal definition of realizability that was lacking in [Zav97]. Next, we have defined formal conditions that a goal must satisfy in order to be realizable, and we have shown that these conditions are complete. Our conditions for unrealizability are similar to the one presented in [Zav97]. For our first theorem of realizability (see Section 4.2.2), conditions (i) and (ii) are semantic conditions that are similar to conditions (i) and (ii) of [Zav97]. Note that reference to the future, which is presented as a separate condition in [Zav97], is in our view a special case of a constraint on the environment. For our second theorem (see Section 5.2), note that unsatisfiability is not identified in [Zav97]. Actually, unsatisfiability could also be seen a special case of constrain on the environment. Our conditions of unrealizability are therefore more specialized than those of [Zav97]. This is a key advantage, as it allows one to define more specialized requirements elaboration techniques for each category of unrealizable goal. Other important points are that all our conditions are defined formally, and are proven to be complete. A requirements elaboration process that consists in using domain properties to gradually refine goals by resolving lack of monitorability, lack of control, and reference to future is also outlined in [Jac95, Zav97]. A fundamental difference with our work is that this process assumes that agent interfaces are given at the beginning of the requirements elaboration process, whereas the method described in the thesis is concerned with the exploration of alternative agent responsibility and interfaces. Furthermore, the agent-driven tactics described in Chapter 6 provide much richer elaboration tactics than the high-level principles outlined in [Jac95, Zav97]. Our tactics are also described in a more formal framework.

The Four-Variable Model underlying the SCR method also emphasizes that the requirements must be described as a relation between monitored and controlled quantities [Par95]. However, the description of this relation is not sufficiently precise: it does not explicitly require that the controlled variables do not depend on the *future* values of monitored quantities. The more detailed semantic domain for the SCR notation presented in [Heit96] solves this problem. A similar semantic domain has also been defined for the RSML notation [Heim96]. Unlike our goal-oriented method, all these models are only concerned with operational specification at the interface of the automated system. They are not concerned with the specification of goals that describe properties on quantities that are not at this interface, the refinement of such goals into realizable requirements, and the derivation of alternative agent responsibilities and interfaces.

Some of the tactics described in Chapter 6 drives the identification of sensor and actuation devices. The resulting requirements model has a structure that corresponds to the standard structure of control system, such as in the Four-Variable Model [Par95] or in the requirements state machine model for process-control system [Jaf91].

Related ground work has also been carried out in the context of concurrent programs. The concept of realizability defined in [Aba89] is similar to the one presented in this dissertation. A temporal specification is said to be realizable if there exists an operational specification that satisfies the temporal specification. However, the concept of operational specification used in [Aba89] differs from ours in several aspects. First, it is based on an interleaving semantics with fairness conditions, whereas our semantics is based on true concurrency. More importantly, [Aba89] does not consider agent monitorability and control. This line of work is concerned with the specification and derivation of concurrent programs rather than with requirements engineering. Agent monitorability and control are important concepts of RE that are missing from that framework.

In the KAOS framework, the use of formal goal refinement patterns and tactics has first been proposed in [Dar95, Dar96]. A library of refinement patterns has been built by extracting patterns from a wide variety of case studies, and by deriving further patterns top-down from basic patterns according to weakening/strengthening relationships between patterns. Goal Refinement tactics are also proposed for selecting appropriate patterns based on semantic criteria. The tactics identified are classified into milestone-driven tactics, case-driven tactics, and agent-driven tactics. The agent-driven tactics proposed there are only defined in a sketchy way and are not related to violations of the realizability meta-constraint.

The library of agent-driven tactics proposed in the thesis builds on that work by providing further guidance for selecting appropriate patterns. Specification elaboration tactics are considered for each kind of realizability problem. Appropriate refinement patterns are then retrieved by browsing the library of agent-driven tactics based on the kind of realizability problem to be solved.

Our library of agent-driven tactics also provides a systematic way to enrich the library of goal refinement patterns. For each agent-driven tactic, we considered whether appropriate goal refinement patterns already existed in the library. New goal refinement patterns have been identified when no appropriate patterns could be found. For instance, the tactic introduce_accuracy_goal (Section 6.5.1) led to the identification of new refinement patterns based on the substitutivity property of temporal logic. Other formal refinements patterns identified are the patterns associated with the tactics resolve reference to strict future, and resolve synchronization problems (Section 6.8); resolve unsatisfiable goal (Section 6.7); and replace unmonitorable/uncontrollable states by events (Sections 6.5.5 and 6.6.5).

10.2. Exception Handling and Fault-Tolerance

In order to get high-quality software, it is of utmost importance to reason about exceptions and faults during software development. There has been a lot of software engineering research to address this for the later stages of architectural design or implementation.

Rigorous definitions of various concepts underlying exception handling can be found in [Cri95, Gar99] --such as specification, program correctness, exception, robustness, failure, error, fault, fault tolerance, and redundancy. Exception handling for modular programs structured as hierarchies of data abstractions is also discussed in [Cri95], including the issues of exception detection and propagation, consistent state recovery, and masking. A failure is defined as a deviation between the actual behavior of the system and that required by its specification [And81, Gar99]. An error is a part of the system

state which leads to failure. The cause of an error is a fault. The objective of fault-tolerance is to avoid system failures, even in the presence of faults [Jal94], or to precisely define the acceptable level of system behavior degradation when faults occur, if the former objective is not realizable [Cri91].

The notion of ideal fault-tolerant component provides a basis for structuring software systems [And81, Ran95]. A system is viewed as a set of interacting components that receive requests for services and produce responses. An idealized fault-tolerant component should in general provide both normal and exceptional responses. Three classes of exceptional situations are identified: interface exception, local exception and failure exception. Different parts of the system are responsible for handling each class of exception.

The concepts involved in fault tolerance are put on more formal grounds in [Aro93, Gar99]. What is meant for a program to tolerate a certain class of fault is formally defined in [Aro93]. This paper also illustrates how fault-tolerant programs can be systematically verified and designed. A compositional method for designing programs that tolerate multiple fault classes is described in [Aro98]. The method is based on the principle of adding detector and corrector components to intolerant programs in a stepwise and non-interfering manner. Various forms of fault-tolerance are discussed in [Gar99]; they are based on whether a program still satisfies its safety properties, liveness properties, or both. Detection and correction are also discussed there as the two main phases in achieving fault-tolerance.

In the database area, [Bor85] describes language mechanisms for handling violations of assumptions in a database. Using such mechanisms, programs can be designed to detect and handle exceptional facts, or the database can adjust its constraints to tolerate the violation.

All the work reviewed above addresses the later phases of architectural design or programming. At those stages, the boundary between the software and its environment has been decided and cannot be reconsidered; the requirements specifications are postulated realistic, correct and complete --which is rarely the case in practice. Empirical studies have suggested that the problem should be tackled much earlier in the software lifecycle [Lut93]. Our work follows that recommendation by addressing the problem of handling abnormal behaviors at requirements engineering time. Reasoning at this stage, in a goal-oriented way, provides much more freedom on adequate ways of handling abnormal behaviors --like, e.g., producing more realistic and more complete requirements, and/or considering alternative requirements or alternative agent assignments that achieve the same goals but result in different system proposals.

There are however clear analogies between exception handling at program level and obstacle analysis at requirements level. The objective of fault-tolerance is to satisfy the program specification despite the presence of faults whereas the objective of obstacle analysis is to satisfy goals despite agent failures. Some of the obstacle resolution strategies are conceptually close to fault-tolerant techniques lifted and adapted to the earlier phase of requirements engineering. The obstacle prevention strategy introduces a form of redundancy where a new goal is introduced to prevent an obstacle from occurring. The obstacle anticipation substrategy is reminiscent of the fault detection and resolution phases for fault-tolerance. (Note, however, that one should not confuse obstacle identification, which is performed at specification time and takes an “external” view on the system, with obstacle detection which is performed at run-time by agents “inside” the

system [Fea98].) The goal restoration and obstacle mitigation strategies also introduce new redundant goals to ensure higher-level goals in spite of the occurrence of obstacles. On the other hand, there are important obstacle resolution strategies, such as goal substitution and agent substitution, that are specific to requirements engineering because of the freedom still left.

In their work, de Lemos et al have also recognized the need for moving towards the requirement analysis phase many of the concerns that may arise during later phases of software development --particularly, the possibility of system faults and human errors [Lem95, And95]. They propose an approach based on an incremental and iterative analysis of requirements for safety-critical systems in the context of system faults and human errors. Their scheme is similar to ours in that it consists of incrementally and iteratively identifying the defects of a requirement specification being elaborated; they use the identified defects to guide the modification of the specification. However, no systematic techniques are provided there for generating the possible faults from the elaborated requirement specification, and for transforming the requirement specification so as to resolve the identified faults. Another difference is that their scheme is based on the progressive decomposition of system entities while we favor goal refinement. (See also [Ber98] for a comparison of this work with ours.)

Many specification languages provide constructs for specifying software functionalities separately for normal and abnormal cases, and then in combination. The Z logical schema combination constructs are typical examples of this [Pot91].

We have tried to convince the reader about the importance of exception handling at the requirements engineering level and, more specifically, at the goal level. Although there are no other formal techniques at the goal level that we are aware of, there has been a lot of work addressing the later stages of RE where a detailed operational model of the software is already available (typically under the form of state machine specifications).

For example, the completeness techniques in [Heim96, Heit96] are aimed at checking whether the set of conditions guarding transitions in a state machine cover all possible cases.

Model checking techniques generate counter examples showing that a temporal logic specification is violated by a finite state machine specification [Hol97, McM93]. In the same vein, planning techniques can be used to exhibit scenarios showing the inconsistency between an abstract property and an operational model [And89, Fic92, Hal97]. One might expect such techniques to be able to generate the scenarios satisfying our obstacles as traces that refute a goal assertion conjoined with the domain theory. However, we currently envision two problems in applying these techniques directly for our purpose. On one hand, we want to conduct the analysis at the goal level for reasons explained throughout the thesis; model checking requires the availability of an operational description of the target system, or of relational specifications [Jac96] that do not fit our higher-level formulation of goals in terms of temporal patterns of behaviour. On the other hand, for the purpose of resolution we need to obtain a formal specification of the obstacle rather than an instance-level scenario satisfying it. A derivation calculus on more abstract specifications seems therefore more appropriate, even though instance scenarios generated by a tool like Nitpick [Jac96] could provide concrete insights for identifying obstacles to relational specifications.

Another important stream of work at the operational specification level concerns the generation of fault trees from a detailed operational model of the system. The technique in

[Lev87] generates fault trees from a Petri-net model. This technique has been adapted to generate fault trees from a state machine model expressed in RSML [Rat96, Mod97]. Several other techniques have also been proposed to generate other standard hazard analysis models from RSML specifications [Rat96, Mod98]. Those techniques can however be applied only once a complete operational specification of the system has been obtained. Furthermore, a very detailed operational specification of the environment of the system would be needed to identify faults caused in the environment (e.g., a detailed model of the behavior of human operators). In contrast, our techniques are intended to be used earlier in the requirements engineering process when a complete specification of the system is not yet available and alternative system boundaries are still being explored. It allows obstacles to be generated from partial declarative specifications that may be gradually elicited during the obstacle identification process. (Note that the generation of fault trees from a state machine model is similar to a recursive application of our 1-state-back obstacle refinement pattern.) Furthermore, goals provide a precise entry point for starting hazard analysis.

The heuristics proposed in this paper for identifying obstacles are somewhat related in spirit to safety requirements checklists [Lev95], in that they embed experience about known forms of obstruction. General criteria corresponding to such checklists have been identified in [Jaf91]. These criteria cover exceptional circumstances such as unexpected inputs, computer error, environmental disturbances, etc. Good RE practices also consider checklists that cover unexpected inputs, operator errors, and other faults or exceptional circumstances [Som97]. Our heuristics are in fact closer to HAZOP-like guidewords that can be used to elicit hazards [Lev95]; such guidewords are made more specific here thanks to our requirements meta-model and specific goal classifications. More formal HAZOP-based techniques have been proposed for forward propagation of perturbations from input variables to output variables in operational specifications [Ree97].

Our work builds on Potts' paper which was the first to introduce the notion of obstacle as a dual notion to goals [Pot95]. Obstacles are identified there by exploration of scenarios of interaction between software and human agents. This exploration is informal and based on heuristics (some of these have been transposed to this thesis, see Section 8.4.4). Obstacle resolution is not studied there.

[Sut98] also builds on Potts' work by proposing additional heuristics for identifying possible exceptions and errors in such interaction scenarios --e.g., scenarios in which events happens in the wrong order, or in which incorrect information is transmitted. Influencing factors such as agent motivation and workload are also used to help anticipate when exceptions may occur and assign probabilities to abnormal events. Generic requirements are attached to exceptions to suggest possible ways of dealing with the problem encountered. The heuristics proposed in [Sut98] are close in spirit to ours; their generic exception handling requirements share the same general objective as our obstacle resolution strategies. Their work is largely informal and centred around the concept of scenario. This provides little systematic guidance compared with the range of obstacle generation/resolution techniques that can be precisely defined through rigorous reasoning on declarative specifications of goals.

Chapter 11
Conclusion

Reasoning about alternatives is at the heart of the software development process. At the implementation stage, the choice of algorithms and data structures may have a critical impact on the performance of the software. During the architectural design stage, the choice of one architecture among several alternatives has a critical impact on competing quality attributes (such as performance, availability, maintainability, security, etc.).

At those stages, the boundary between the automated system has been decided and cannot be reconsidered.

At the requirements engineering stage, alternative decisions consist in:

- alternative refinements of goals into subgoals;
- alternative responsibility assignments of goals to agents, leading to alternative agent interfaces;
- alternative resolutions of conflicts between goals; and
- alternative resolutions of obstacles to the satisfaction of goals.

Choosing among those alternatives generates alternative systems proposals that may be quite different. Choices may have a critical impact on the performance, cost and risks associated with the system. This step, however, is taken for granted by most specification techniques. As a result, alternative, perhaps superior, decisions are not systematically explored; and the rationale for such decisions is not made explicit for easier evolution.

The thesis has described systematic support for *generating* alternative system proposals at the RE level. More specifically, it describes systematic techniques supporting

(i) the generation of alternative goal refinements, responsibility assignments and agent interfaces;

(ii) the derivation of operational requirements from goal formulations;

(iii) the identification of obstacles to the satisfaction of goals and the generation of alternative obstacle resolutions through the deidealization of initial goals and assumptions and through the generation of new goals to prevent, reduce or tolerate the identified obstacles.

11.1. Contributions

The work reported in the thesis is based on an existing goal-oriented requirements elaboration method, called KAOS. The thesis enriches the KAOS framework through three sets of techniques:

1. The first set of techniques provides systematic guidance to constructively explore alternative agent responsibilities and interfaces from high-level goals.

These techniques are grounded on a formal model of agent responsibility, monitorability and controllability. In particular, our model defines a *realizability* meta-constraint between an agent responsibility for a goal and its interfaces. The meta-constraint formally captures what is meant for a goal to be assignable as the responsibility of a single agent.

The realizability meta-constraint has been seen to play a central role in the goal-driven requirements elaboration process. Violations of that meta-constraint drives the identification of new agents and the refinement of goals into subgoals until the latter are realizable by single agents.

A *taxonomy of realizability problems* was defined to guide the identification and classification of realizability problems. The realizability problems were thereby seen to be related to lack of monitorability, lack of control, goal unsatisfiability, references to the future, and unbounded achievement goals. This taxonomy is based on the formal definition of realizability, and was proved to be complete.

A library of *agent-driven tactics* was then proposed to provide systematic guidance for identifying agents and recursively refining goals into subgoals. Applications of these tactics are prompted by the need to resolve realizability problems. The application of alternative agent-driven tactics allows one to explore alternative goal refinements, alternative responsibility assignments of goals to agents, and alternative agent interfaces.

2. When a goal is realizable by a single agent, *formal operationalization patterns* can be used to derive, from the formal definition of the goal, the operations that are relevant to the goal, and the requirements on those operations that ensure that the goal is satisfied.

3. A third set of techniques was described to identify obstacles to the satisfaction of idealized goals and assumptions, and to generate alternative obstacle resolutions.

   We have defined a library of *obstacle identification patterns and heuristics* for identifying obstacles from goals; and a library of *obstacle resolution tactics* that transform the goal model by deidealizing goals and assumptions or by generating new goals so as to avoid, reduce or tolerate the identified obstacles.

We have also contributed to a fourth set of techniques, not presented in the thesis, to identify conflicts between goals, and to generate alternative conflict resolutions [Lam98b].

The techniques were applied to two real case studies: the LAS ambulance despatching system, and the BART automated train control system.

11.2. Limitations and Future Directions

The KAOS framework and the techniques described in the thesis require extensions in several directions.

11. 2. 1. Evaluating and Selecting Alternative Designs

The techniques presented in the thesis can be used to *generate* a set of alternative decisions. For goal refinement, we have defined different elaboration tactics for introducing alternative agents and for generating alternative goal refinements. The selection of which tactics to apply may depend on numerous criteria such as performance, cost, safety, security, and so on. For obstacle analysis, we have defined different elaboration tactics for avoiding, reducing or tolerating obstacles. In this case also, the selection of which tactics to apply will depend on the kind of obstacles, on the severity of its consequences, on its probability of occurrence, and on the cost of its resolution.

Alternative responsibility assignments and alternative obstacle resolutions result in alternative system designs. The evaluation of the generated alternatives and the selection of an "optimal" design is a critical step for which much further work is required. Issues that need to be addressed include the following:

- *What are the criteria to be used for evaluating and selecting among alternative design decisions?*

  As mentioned before, the evaluation and selection of alternative design decisions is based on multiple competing 'non-functional' requirements such as performance, cost, risk, usability, maintainability, etc. The NFR framework provides *qualitative* reasoning schemes for evaluating alternative system designs against non-functional requirements described at a very high-level of abstraction [Myl92, Chu2K]. This framework could be extended with *quantitative* techniques that would evaluate alternative design decisions with respect to optimization goals defined in terms of *measurable* properties of the system. For instance, in the LAS case study, alternative design decisions would be evaluated against the expected percentage of incidents for which an ambulance will arrive at the incident scene within 14 minutes.

- *When and how should such evaluation of alternatives be carried out?*

  The activities of generating alternative design decisions and deciding which alternatives will actually be selected should be clearly distinguished and separated to prevent premature evaluation from stifling generation of new suggestions [Eas94]. On the other hand, it is necessary to discard some alternatives early in order to keep the generation of alternative designs within reasonable bounds. It is also likely -and even desirable- that non-functional requirements used to evaluate alternatives lead to the generation of further alternatives. The generation and evaluation phases are therefore clearly intertwined, and proceed iteratively.

- *How detailed and formal does the specification of generated alternatives need to be in order to evaluate them accurately?*

  Specifying every possible alternative design in detail is clearly not justified at this early stage of the development process. One needs to make trade-offs between the effort spent in generating and evaluating alternative designs, on the one hand, and the confidence in the design decisions being made, on the other. Again, an iterative model seems to be the most sensible approach. Some high-level design decisions can be made early based on high-level non-functional requirements, before specifying the alternatives in full detail; other design decisions may require more detailed knowledge about the domain and the proposed system.

Some issues are also specific to the selection of obstacle resolution tactics; e.g. how to determine the severity of the consequences of an obstacle, and how to determine the consequence of the *combination* of obstacles? How to determine the probabilities of occurrences of obstacles?

11. 2. 2. Specialized Elaboration Tactics based on Goal Categories

The various techniques described in the thesis provide *general* strategies for generating alternative goal refinements and alternative responsibility assignments, and for generating and resolving obstacles to goals and assumptions. As a result, their application still requires much work from the user, whereas more specialized tactics would provide more specific guidance.

The adaptation and extension of these techniques to specific goal categories, such as satisfaction, information, accuracy, safety, or security would provide such a specialized guidance in elaborating requirements models. In particular, the specialization of obstacle analysis techniques to security goals would allow for the systematic identification of potential threats to the system. The basic idea for such specialized tactics based on goal categories would be similar to the idea of using problem frames [Jac95b, Jac2K].

11. 2. 3. A Rich Taxonomy of Formal Patterns for Requirements Elaboration

The techniques presented in the thesis are based on formal patterns for goal refinement, goal operationalization, and goal obstruction. *The coverage of these patterns is relative to the coverage of a taxonomy of goal definition patterns*. The efficiency of our techniques in practice rely on the assumption that most properties occurring in practice can be specified using a small set of goal patterns. This assumption is partly supported by an empirical study reported in [Dwy99].

Further work is required to validate that assumption, to identify an appropriate taxonomy of relevant formal goal patterns, and to define a language that would allow one to use such patterns without writing formal assertions in temporal logic. The long-term objective is to be able to completely hide (or disguise) the formal assertion layer from users of the method while keeping its mathematical rigour.

11. 2. 4. Tool Support

The construction of adequate tool support for the goal-oriented requirements process is an ongoing research project. A preliminary prototype tool called KAOS/GRAIL has been developed at CEDITI [Dar98].

Further work is required to integrate the various techniques presented in the thesis in such a tool. We discuss the possibilities of automated support for the various techniques presented in the thesis.

1. Identifying realizability problems

Identifying lack of monitorability and lack of control is straightforward as these problems are purely syntactical. References to the future and unsatisfiability are defined as semantic conditions and are more difficult to identify automatically. For references to future, we have identified a set of recurrent patterns of goal definitions with such a problem. These patterns can be used to identify references to the future from the syntactical definition of the goal; however, this set of patterns is probably not complete. Further work could investigate the definition of general techniques for computing references to future and for determining unsatisfiability.

Theorem 1 could also provide the basis for automatically checking whether a goal is realizable by an agent on some restricted subset of the language. For instance, one could try to define syntactical conditions on Buchi automata (similar to those presented in [Zav9]) that are equivalent to the conditions of Theorem 1 in Chapter 4. This would allow one to automatically check the realizability of a goal defined over a limited, finite state space.

2. Selecting applicable agent-driven elaboration tactics

Once realizability problems are identified, a tool could automatically identify the tactics whose preconditions holds, and allow users to browse the library of applicable tactics.

3. Applying agent-driven elaboration tactics

Tool support could be investigated to help users in applying a selected tactic. The following problems will limit the amount of automated support that can be provided to apply the tactics:

- the strict application of tactics produces first-sketch goal definitions that need to be adapted manually;
- the number of variants of formal goal refinement patterns associated to a tactic may make the automatic application of tactics impractical.

One approach toward reducing these problems would be to restrict the language in which goals are defined to a small set of goal patterns. This require the preliminary definition of an adequate taxonomy of goal patterns as discussed above.

4. Applying operationalization patterns

Applying operationalization patterns is not different from applying goal refinement patterns; applicable patterns can be retrieved automatically by matching goal formulations to patterns in the library [Dar96].

5. Generating obstacles from goals

Relevant obstruction patterns can also be automatically retrieved by matching goal formulations to patterns in the library. This will prompt users to identify appropriate domain properties needed to derive the obstacles.

6. Selecting and applying obstacle resolution tactics

Tool support for the selection and application of obstacle resolution tactics would be based on the same principles as those used to select and apply agent-driven tactics. The same limitation apply; the strict application of tactics produces first-sketch goal definitions that need to be adapted manually.

7. Using the semantics proposal as a basis for animation and dedicated checks

The formal model of agents and the formal semantics of operations provides the basis for the construction of further tool support such as an animation tool and a tool to check the completeness and consistency of operational models of agents in the spirit of SCR [Heit96] and RSML [Heim96].

11. 2. 5. Goal-Oriented Elaboration of Software Architecture

The systematic derivation of a software architecture from a set of functional and non-functional requirements is an important research issue that has received fairly little attention so far. Recently, goal-based and agent-based approaches have been proposed for the constructive elaboration of software architectures [Lam2Kc, Myl2K].

By viewing components of a software architecture as agents of finer granularity, the techniques described in the thesis could be specialized and adapted to the generation of alternative software architectures from goals assigned as responsibilities of software components. The evaluation of alternative software architectures would also be based on non-functional requirements identified during the requirements elaboration process.

11. 2. 6. Agent Refinement

The KAOS method and the techniques described in the thesis assume agents of fixed granularity. Such agents correspond to concrete agents (such as software, people, and hardware devices) that can be found in the application domain. Sometimes one may need to reason about agents at different levels of abstraction. For example, in the BART system, one might view the system formed of the train tracking system, the train controllers at the stations, and the on-board train controllers as a single (composite) agent. This would allow responsibility assignments to be made earlier on more abstract goals. One could then refine such goals assigned to abstract agents into subgoals assigned to agents of finer granularity. As another example, the derivation of a software architecture from requirements will also require to represent and reason about agents at different levels of granularity; a software agent would be refined into several agents corresponding to software components of finer granularity. This sort of agent refinement process would be similar in spirit to the refinement of Abstract Machines in the B method [Abr96]. Future work could extend the KAOS language and method to support a requirements elaboration process in which goals and agents are refined in parallel.

References

[Aba89] Abadi M, Lamport L, Wolper P, “Realizable and Unrealizable Specifications of Reactive Systems”, *Proc 16 th ICALP*, 1989, LNCS 372, pp 1-17

[Abr96] J.-R. Abrial. The B-Book: Assigning Programs to Meanings. Cambridge University Press, 1996

[Alp87] B. Alpern and F.B. Schneider. “Recognizing safety and liveness”, *Distributed Computing*, 2:117--126, 1987.

[Amo94] E. G. Amoroso. *Fundamentals of Computer Security Technology.* Prentice-Hall PTR, Upper Saddle River, NJ, 1994.

[And81] T. Anderson and P.A. Lee, *Fault Tolerance: Principles and Practice. Prentice Hall*, 1981.

[Ant94] Anton, A. I., McCracken, W. M., Potts, C., “Goal Decomposition and Scenario analysis in Business Process Engineering”, CAiSE’94, LNCS 811, Springer-Verlag, pp. 94-104.

[And89] J.S. Anderson and S. Fickas, “A Proposed Perspective Shift: Viewing Specification Design as a Planning Problem”, *Proc. IWSSD-5 - 5th Intl. Workshop on Software Specification and Design*, IEEE, 1989, 177-184.

[And95] T. Anderson, R. de Lemos, and A. Saeed, “Analysis of Safety Requirements for Process Control Systems”, in *Predictably Dependable Computing Systems*, B. Randell, J.C. Laprie, B. Littlewood and H. Kopetz (ds.), Springer-Verlag, 1995.

[Aro93] A. Arora and M.G. Gouda, “Closure and Convergence: A Foundation of Fault-Tolerant Computing”, *IEEE Trans. Software Eng.*, vol. 19, no. 11, pp. 1015- 1027, 1993.

[Aro98] A. Arora and S. Kulkarni, “Component Based Design of Multitolerant Systems”, *IEEE Trans. Software Eng.*, vol. 24, no. 1, pp. 63 - 78, Jan 1998.

[Bel76] Bell, T. E., and Thayer, T. A., "Software Requirements: are they really a problem," Proc. 2nd Int. Conf. on Software Engineering, 1976, pp. 61-68.

[Ber98] D.M. Berry, “The Safety Requirements Engineering Dilemma”, *Proc. IWSSD’98 - 9th International Workshop on Software Specification and Design*, Isobe, IEEE CS Press, April 1998.

[Bor85] A. Borgida, “Language Features for Flexible Handling of Exceptions in Information Systems”, ACM Transactions on Database Systems Vol. 10 No. 4, Dec. 1985, 565-603.

[Bra85] R.J. Brachman and H.J. Levesque (eds.), *Readings in Knowledge Representation*, Morgan Kaufmann, 1985.

[But96] Ricky W. Butler. *An Introduction to Requirements Capture Using PVS: Specification of a Simple Autopilot*. NASA Technical Memorandum 110255. NASA Langley Research Center, May 1996.

[Chu93] Lawrence Chung. *Representing and using Non-functional Requirements: a Process-Oriented Approach*. PhD thesis, Computer Science Department, University of Toronto, Toronto (Canada), June 1993.

[Chu2K] L. Chung, B. A. Nixon, E. Yu and J. Mylopoulos, Non-Functional Requirements in Software Engineering, Kluwer Academic Publishers, Boston, 2000.

[Cri91] F. Cristian, "Understanding Fault-Tolerant Distributed Systems", *Comm. of the ACM*, February 1991.

[Cri95] F. Cristian, "Exception Handling", in: *Software Fault Tolerance*, M.R. Lyu (Ed.), Wiley, 1995.

[Dar91] Dardenne, A., Fickas, S., van Lamsweerde, A., "Goal-Directed Concept Acquisition in Requirements Elicitation", Proc. IWSSD-6 - 6th Intl. Workshop on Software Specification and Design, Como, 1991, 14-21.

[Dar93] Dardenne, A., van Lamsweerde, A., Fickas, S., "Goal-Directed Requirements Acquisition", Science of Computer Programming, Vol. 20, 1993, 3-50.

[Dar95] R. Darimont, *"Process Support for Requirements Elaboration"*, PhD Thesis, Université catholique de Louvain, Dépt. Ingénierie Informatique, Louvain-la-Neuve, Belgium, 1995.

[Dar96] R. Darimont and A. van Lamsweerde, "Formal Refinement Patterns for Goal-Driven Requirements Elaboration", *Proceedings 4th ACM Symposium on the foundation of Software Engineering (FSE 4)*, San Fransisco, Oct. 1996, pp. 179-190.

[Dar98] Darimont, R, Delor, E., Massonet, P., van Lamsweerde, A.,GRAIL/KAOS: An Environment for Goal-Driven Requirements Engineering, IEEE, Proceedings of the 20th Interactional Conference on Software Engineering, Kyoto, April 1998, Vol 2, pp. 58-62.

[Dem78] T. DeMarco, *Structured Analysis and System Specification*, Yourdon Press, New York, 1978.

[Dij71] E.W. Dijkstra, "Hierarchical Ordering of Sequential Processes," *Acta Informatica* 1, 1971, pp. 115-138.

[Dij76] E.W. Dijkstra, *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, N.J. (1976).

[Dwy99] M. Dwyer, G. Avrunin, and J. Corbett. "Patterns in property specifications for finite-state verification", In *Proceedings of the 21st International Conference on Software Engineering*, May 1999.

[Fea87] M. Feather, "Language Support for the Specification and Development of Composite Systems", *ACM Trans. on Programming Languages and Systems* 9(2), Apr. 87, 198-234.

[Fea91] M. Feather, S. Fickas, and R. Helm, "Composite System Design: the Good News and the Bad News", in *Proceedings of the 6th Annual Knowledge-Based Software Engineering (KBSE) Conference*, Syracuse, NY, September 1991, pp. 16-25, IEEE Computer Society Press.

[Fea94] M. Feather, "Towards a Derivational Style of Distributed System Design", *Automated Software Engineering* 1(1), 31-60.

[Fea98] M. Feather, S. Fickas, A. van Lamsweerde, and C. Ponsard, “Reconciling System Requirements and Runtime Behavior”, *Proc. IWSSD’98 - 9th International Workshop on Software Specification and Design*, Isobe, IEEE CS Press, April 1998.

[Fic92] S. Fickas and R. Helm, “Knowledge Representation and Reasoning in the Design of Composite Systems", *IEEE Trans. on Software Engineering*, June 1992, 470-482.

[Fin94] A. Finkelstein, D. Gabbay, A. Hunter, J. Kramer, and B. Nuseibeh, "Inconsistency Handling in Multi-perspective Specifications", IEEE TSE, 20(8): 569-578, 1994.

[Fin96] A. Finkelstein, “The London Ambulance System Case Study”, *succedings of IWSSD8 - 8th Intl. Workshop on Software Specification and Design, ACM Software Engineering Notes*, September 1996.

[Gar99] F.C. Gartner, “Fundamentals of Fault-Tolerant Distributed Computing in Asynchronous Environment”, *ACM Computing Surveys*, Vol. 31, No. 1, March 1999. pp 1-26.

[Gri81] D. Gries, *The Science of Programming*. Springer-Verlag, 1981.

[Gun2K] Carl A. Gunter, Elsa L. Gunter, Michael Jackson, and Pamela Zave. A reference model for requirements and specifications. IEEE Software, May/June 2000.

[Hal98] R.J. Hall, “Explanation-Based Scenario Generation for Reactive System Models”, *Proc. ASE’98*, Hawaii, Oct. 1998.

[Har87] D. Harel, “Statecharts: A Visual Formalism for Complex Systems”, *Science of Computer Programming*, vol. 8, 1987, 231-274.

[Heim96] M.P. Heimdahl and N.G. Leveson, “Completness and Consistency Checking in Hierarchical State-Based Requirements”, *IEEE Transactions on Software Engineering*, Vol. 22, No. 6, June1996, pp. 363-377.

[Heim98] M.P.E Heimdahl, J.M. Thompson and B.J. Czerny, “Specification and Analysis of Intercomponent Communication”, *IEEE Computer*, Vol. 31, No. 4, 1998, pp. 47-54.

[Heit96] C. Heitmeyer, R. Jeffords and B. Labaw, “Automated Consistency Checking of Requirements Specifications”, *ACM Transactions on Software Engineering and Methodology*, Vol. 5, No. 3, July 1996, pp. 231-261.

[Heit96b] Heitmeyer, C.; Mandrioli, D.: editors. Formal methods for real-time computing. New York: John Wiley, 1996.

[Hen80] K.L. Heninger, “Specifying Software Requirements for Complex Systems: New Techniques and Their Application”, *IEEE Transaction on Software Engineering*, Vol. 6, No. 1, January 1980, pp. 2-13.

[Hol97] G. Holtzman, “The Model Checker SPIN”, *IEEE Trans. on Software Engineering* Vol. 23 No. 5, May 1997, 279-295.

[Hun98] A. Hunter and B. Nuseibeh. Managing inconsistent specifications: Reasoning, analysis and action. *ACM Transactions on Software Engineering and Methodology*, 7(4):335--367, 1998.

[Jac96] D. Jackson and C.A. Damon, “Elements of Style: Analyzing a Software Design Feature with a Counterexample Detector”, *Proc. ISTA '96 - Intl. Symp. on Software Testing and Analysis*, ACM Softw. Eng. Notes Vol. 21 No. 3, 1996, 239-249.

[Jac83] M.A. Jackson, *System Development*, Prentice Hall, 1983.

[Jac93] M. Jackson and P. Zave, “Domain Descriptions”, *Proc. RE’93 - 1st Intl. IEEE Symp. on Requirements Engineering*, Jan. 1993, 56-64.

[Jac95] M. Jackson and P. Zave, “Deriving Specifications from Requirements: an Example”, *Proc. ICSE’95 - 17th International Conference on Software Engineering*, April 1995, pp. 15-24.

[Jac95b] D. Jackson and M. Jackson, "Problem Decomposition for Reuse," Technical Report: CMU-CS-95-108, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pa., 1995.

[Jac2K] M. Jackson, *Problem Frames: Analyzing and Structuring Software Development Problems*. Addison-Wesley, 2000.

[Jaf91] M.S. Jaffe et al., “Software Requirements Analysis for Real-Time Process-Control Systems”, *IEEE Transactions on Software Engineering*, vol. 17, no. 3, March 1991, pp. 241-258.

[Jal94] P. Jalote, *Fault Tolerance in Distributed Systems*, Prentice Hall, 1994.

[Jon90] Jones, C.B., *Systematic Software Using VDM*, 2nd ed., Prentice Hall, 1990.

[Jon2K] Edwin de Jong, Jaco van de Pol, and Jozef Hooman. Refinement in requirements specification and analysis: A case study. In *7th IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS)*, pages 290-298, Edinburgh, Scotland, April 2000. IEEE Computer Society.

[Jos96] M. Joseph. *Real-Time Systems: Specification, Verification and Analysis*. Prentice Hall Intl., 1996.

[Kaz99] Kazman, R.; Barbacci, M.; Klein, M.; Carriere, S. J.; &Woods, S. G. "Experience with Performing Architecture Tradeoff Analysis," 54-63. Proceedings of the 21 st International Conference on Software Engineering, Los Angeles, CA, May 1999.

[Ken93] S.J.H. Kent, T.S.E. Maibaum and W.J. Quirk, “Formally Specifying Temporal Constraints and Error Recovery”, *Proc. RE’93 - 1st Intl. IEEE Symp. on Requirements Engineering*, Jan. 1993, pp. 208-215.

[Koy92] R. Koymans, *Specifying message passing and time-critical systems with temporal logic*, LNCS 651, Springer-Verlag, 1992.

[Lamp94] L. Lamport. The temporal logic of actions. ACM Transactions on Programming Languages and Systems, 16(3):872--923, May 1994.

[Lam95] A. van Lamsweerde, R. Darimont and P. Massonet, “Goal-Directed Elaboration of Requirements for a Meeting Scheduler: Problems and Lessons Learned”, *Proc. RE’95 - 2nd Intl. Symp. on Requirements Engineering*, York, IEEE, 1995.

[Lam98a] A. van Lamsweerde and E. Letier, “Integrating Obstacles in Goal-driven Requirements Engineering”, *Proc. ICSE’98 - 20 th Intl Conf. on Software Engineering*, Kyoto, IEEE, April 1998.

[Lam98b] A. van Lamsweerde, R. Darimont and E. Letier, “Managing Conflicts in Goal-driven Requirements Engineering”, *IEEE Transactions on Software Engineering*, Vol. 24, No. 11, Nov. 1998, pp.908-926.

[Lam98b] A. van Lamweerde A., L. Willemet, ‘Inferring Declarative Requirements Specification from Operational Scenarios’, *IEEE Transactions on Software Engineering*, Special Issue on Scenario Management, Vol. 24, No. 12, 1089- 1114, Dec. 1998.

[Lam2Ka] A. van Lamsweerde and E. Letier, “Handling Obstacles in Goal-driven Requirements Engineering”, *IEEE Transactions on Software Engineering*, Special Issue on Exception Handling, 2000.

[Lam2Kb] A. van Lamsweerde, "Formal Specification: a Roadmap". In *The Future of Sofware Engineering*, A. Finkelstein (ed.), ACM Press, 2000.

[Lam2Kc] A. van Lamsweerde, "Requirements Engineering in the Year 00: A Research Perspective", Keynote Paper for ICSE’2000 - 22nd International Conference on Software Engineering, Limerick, ACM Press, 2000

[LAS93] Report of the Inquiry Into the London Ambulance Service, February 1993. The Communications Directorate, South West Thames Regional Authority, ISBN 0-905133-70-6. See also the London Ambulance System home page, http://hsn.lond-amb.sthames.nhs.uk/http.dir/service/organisation/features / info.html.

[Lem95] R. de Lemos, B. Fields and A. Saeed, “Analysis of Safety Requirements in the Context of System Faults and Human Errors”, *Proceedings of the IEEE International Symposium and Workshop on Systems Engineering of Computer Based Systems*. Tucson, Arizona. March 1995. pp. 374-381.

[Lev87] N.G. Leveson and J.L. Stolzy, “Safety Analysis using Petri Nets”, IEEE Trans. Software Eng., vol. 13, no. 3, pp. 386-397, March 1987.

[Lev94] N. G. Leveson, M.P.E. Heimdahl, H. Hildreth, J.D. Reese, “Requirements Specification for Process-Control Systems, I*EEE Transactions on Software Engineering*, Vol. 20, No. 9, September 1994.

[Lev95] N.G. Leveson, *Safeware - System Safety and Computers*, Addison-Wesley, 1995.

[Lon82] London, P.; Feather, M.: Implementing Specification Freedoms. *Science of Computer Programming*, Vol. 2, 1982, pp. 91--131.

[Lut93] R.R. Lutz, “ Analyzing Software Requirements Errors in Safety-Critical Embedded Systems”, Proc. RE’93, pp. 126-133.

[Mai93] T. Maibaum, “Temporal Reasoning over Deontic Specifications,” in J.Ch. Meyer and R.J. Wieringa (Eds.), *Deontic Logic in Computer Science - Normative System Specification*, Wiley, 1993, pp. 141-202.

[Man92] Z. Manna and A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems*, Springer-Verlag, 1992.

[Man96] Z. Manna and the STep Group, “STeP: Deductive-Algorithmic Verification of Reactive and Real-Time Systems”, *Proc. CAV’96 - 8th Intl. Conf. on Computer-Aided Verification*, LNCS 1102, Springer-Verlag, July 1996, 415-418.

[Mas97] P. Massonet and A. van Lamsweerde, "Analogical Reuse of Requirements Frameworks", Proc.RE-97 - 3rd Int. Symp. on Requirements Engineering, Annapolis, 1997, 26-37.

[McM93] K.L. McMillan, *Symbolic Model Checking: An Approach to the State Explosion Problem*, Kluwer, 1993.

[Mei93] J. Ch. Meyer and R.J. Wieringa (Eds.), Deontic Logic in Computer Science - Normative System Specification, Wiley, 1993.

[Mod97] F. Modugno, N.G. Leveson, J.D. Reese, K. Partridge, and S.D. Sandys, Integrated Safety Analysis of Requirements Specifications, Proc. RE-97, 3rd International Symposium on Requirements Engineering, Annapolis, 1997.

[Myl92] Mylopoulos, J., Chung, L., Nixon, B., “Representing and Using Nonfunctional Requirements: A Process-Oriented Approach”, *IEEE Trans. on Software. Engineering*, Vol. 18 No. 6, June 1992, pp. 483-497.

[Myl99] Mylopoulos, J., Chung, L. and Yu E., “From Object-Oriented to Goal-Oriented Requirements Analysis”, *Communication of the ACM*, Vol. 42, No. 1, January 1999, pp. 31-37.

[Myl2K] J. Mylopoulos and J. Castro. Tropos: A framework for requirements-driven software development. In J. Brinkkemper and A. Solvberg, editors, Information Systems Engineering: State of the Art and Research Themes. SpringerVerlag, 2000.

[Par86] D.L. Parnas and P.C. Clements. A rational design process: How and why to fake it. *IEEE Transactions on Software Engineering*, SE-12:251--257, 1986.

[Par95] D.L. Parnas, J. Madey, “Functional documents for computer systems”, *Science of Computer Programming*, Vol. 25, 1995, pp. 41-61.

[Per89] D.E. Perry, “The Inscape Environment”, *Proc. ICSE-11, 11th Intl. Conf. on Software Engineering*, 1989, pp. 2-12.

[Pot91] B. Potter, J. Sinclair and D. Till, *An Introduction to Formal Specification and Z*. Prentice Hall, 1991.

[Pot95] C. Potts, “Using Schematic Scenarios to Understand User Needs”, *Proc. DIS’95 - ACM Symposium on Designing interactive Systems: Processes, Practices and Techniques*, University of Michigan, August 1995.

[Ran95] B. Randel and J. Xu, “The evolution of the recovery block concept”, in: *Software Fault Tolerance*, M.R. Lyu (Ed.), Wiley, 1995.

[Rat96] V. Ratan, K. Partridge, J.D. Reese and N.G. Leveson, Safety Analysis Tools for Requirements Specifications, Compass 96, Gaithersburg, Maryland, June1996.

[Ree97] J.D. Reese and N. Leveson, “Software Deviation Analysis”, *Proc. ICSE’97 - 19th Intl. Conference on Software Engineering*, Boston, May 1997, 250-260

[Rob90] Robinson, W.N., “Negotiation Behaviour During Requirements Specification”, *Proc. 12th International Conference on Software Engineering*, pp 268-276, IEEE, 1990.

[Ros92] D.S. Rosenblum, “Towards a Method of Programming with Assertions”, *Proc. ICSE-14, 14th Intl. Conf. on Software Engineering*, 1992, pp. 92-104.

[Ros77] D.T. Ross and K.E. Schoman, “Structured Analysis for Requirements Definition”, IEEE Transaction on Software Engineering, Vol. 3, No. 1, 1997, pp. 6-15.

[Rub92] K.S. Rubin and A. Goldberg, “Object Behavior Analysis”, Communication of the ACM, Vol. 35, No. 9, September 1992, pp. 48-62.

[Rum91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice-Hall, Englewood Cliffs, New Jersey, 1991.

[Rum98] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1998.

[Smi2K] Smith, G. Stepwise development from ideal specifications. In Edwards, J., editor, Australasian Computer Science Conference (ACSC 00), volume 22(1) of Australian Computer Science Communications, pages 227--233. IEEE Computer Society.

[Som97] I. Sommerville and P. Sawyer, *Requirements Engineering: A Good Practice Guide*. Wiley, 1997.

[Ste74] W.P. Stevens, G.J. Myers and L.L. Constantine, ‘Structured Design’, *IBM Systems Journal*, Vol. 13, No. 2, 1974, pp. 115-139.

[Sut98] A.G. Sutcliffe, N.A. Maiden, S. Minocha and D. Manuel, “Supporting Scenario-Based Requirements Engineering”, *IEEE Trans. Software Eng*. vol. 24, no. 12, Dec.1998, 1072-1088.

[Win99] V. Winter, R. Berg, and J. Ringland, “Bay Area Rapid Transit District, Advance Automated Train Control System: Case Study Description”. Technical Report, Sandia National Labs, 1999. www.sandia.gov/ast/papers/BART_case_study.pdf.

[Yu93] E. Yu, “Modelling Organizations for Information Systems Requirements Engineering”, *Proc. RE'93 - 1st Intl Symp. on Requirements Engineering*, IEEE, Jan 1993, pp. 34-41.

[Yue87] K. Yue, “What Does It Mean to Say that a Specification is Complete?”, *Proc. IWSSD-4, Fourth International Workshop on Software Specification and Design*, Monterey, 1987.

[Zav97] P. Zave and M. Jackson. Four dark corners of requirements engineering. *ACM Trans. Software Eng. and Methodology*, 6(1):1-30, Jan. 1997. 85

Annex A. Proofs of Chapter 4

A. 1. Properties of Agent Runs

Property 1 (interface restriction) -- For all $\sigma_1, \sigma_2 \in$ Run(ag), $s \in$ State(V), if $\sigma_1 \sim_{Voc(ag)} \sigma_2$ then $\sigma_1 + s \in$ Run(ag) iff $\sigma_2 + s \in$ Run(ag)

Proof: Let $\sigma_1, \sigma_2 \in$ Run(ag), and $s \in$ State(V), such that $\sigma_1 \sim_{Voc(ag)} \sigma_2$.

1. Suppose that $\sigma_1 + s \in$ Run(ag), we show that $\sigma_2 + s \in$ Run(ag).

$\sigma_1 + s \in$ Run(ag)
implies {definition of Run(ag)}
$< \sigma_{1|Voc(ag)} , s_{|Ctrl(ag)} > \in$ Next(ag)
implies {$\sigma_1 \sim_{Voc(ag)} \sigma_2$}
$< \sigma_{2|Voc(ag)} , s_{|Ctrl(ag)} > \in$ Next(ag)
implies {$\sigma_2 \in$ Run(ag), definition of Run(ag)}
$\sigma_2 + s \in$ Run(ag)

2. By symmetry, we also have that if $\sigma_2 + s \in$ Run(ag), then $\sigma_1 + s \in$ Run(ag).

Property 2 (control restriction) -- For all $\sigma \in$ Run(ag), $s_1, s_2 \in$ State(V), if $s_1 \sim_{Ctrl(ag)} s_2$ then $\sigma + s_1 \in$ Run(ag) iff $\sigma + s_2 \in$ Run(ag)

Proof: Let $\sigma \in$ Run(ag), and $s_1, s_2 \in$ State(V), such that $s_1 \sim_{Ctrl(ag)} s_2$.

1. Suppose that $\sigma + s_1 \in$ Run(ag), we show that $\sigma + s_2 \in$ Run(ag).

The proof is done by cases, dependent on whether the path σ is empty or not.

Case 1: $\sigma = <>$.
$s_1 \in$ Run(ag)
implies {definition of Run(ag)}
$s_{1|Ctrl(ag)} \in$ Init(ag)
implies {$s_1 \sim_{Ctrl(ag)} s_2$}
$s_{2|Ctrl(ag)} \in$ Init(ag)
implies {definition of Run(ag)}
$s_2 \in$ Run(ag)

Case 2: $\sigma \neq <>$.
$\sigma + s_1 \in$ Run(ag)
implies {definition of Run(ag)}
$< \sigma_{|Voc(ag)} , s_{1|Ctrl(ag)} \in$ Next(ag)
implies {$s_1 \sim_{Ctrl(ag)} s_2$}
$< \sigma_{|Voc(ag)} , s_{2|Ctrl(ag)} \in$ Next(ag)
implies {definition of Run(ag)}
$\sigma + s_2 \in$ Run(ag)

2. By symmetry, we also have that if $\sigma + s_2 \in Run(ag)$, then $\sigma + s_1 \in Run(ag)$.

Property 3 (finitely violable) -- For all $h \in History(V)$,
if $h \notin Behaviour(ag)$ then there exists a finite prefix σ of h such that $\sigma \notin Run(ag)$

Proof: Let $h \in History(V)$, such that $h \notin Behaviour(ag)$.

By definition of Behaviour(ag), if $h \notin Behaviour(ag)$, then

$h(0)_{|Ctrl(ag)} \notin Init(ag)$
or there exists $i \in Nat$ such that
$< h[i-1]_{|Voc(ag)}, h(i)_{|Ctrl(ag)} > \notin Next(ag)$

If $h(0)_{|Ctrl(ag)} \notin Init(ag)$, then $h(0)$ is a finite prefix of h such that $h(0) \notin Run(ag)$. Otherwise, there exists $i \in Nat$ such that $h[i]$ is a finite prefix of h such that $h[i] \notin Run(ag)$.

The following corollary will be used in the proof of Theorem 2 of Chapter 5. Its proof is similar to the proofs of Properties 1 and 2.

Corollary 1. For all $h_1, h_2 \in Hist(V)$,
if $h_1 \sim_{Voc(ag)} h_2$ then $h_1 \in Behaviour(ag)$ iff $h_2 \in Behaviour(ag)$.

A. 2. Semantic Conditions for Realizability

The following lemmas are used in the proof of Theorem 1.

Lemma 1. Let $\Delta(ag) = <Init(ag), Next(ag)>$ be a agent transition system. If $\sigma \in Run(ag)$ then every prefix of σ is also among the agent runs.

Proof. The lemma follows directly from the definition of Run(ag).

Lemma 2. Let $\Delta(ag) = <Init(ag), Next(ag)>$ be a agent transition system. Every finite run $\sigma \in Run(ag)$ can be extended to an infinite behaviour $h \in Behaviour(ag)$.

Proof. The lemma follows from the fact that Next(ag) is a total relation -- see Section 4.2.4.

Lemma 3. Let $G \subseteq History(V)$ and let $\Delta(ag) = <Init(ag, Next(ag)>$ be an agent transition system. If

$Behaviour(ag) = G$,

then for all $\sigma \in Path(V)$,

$\sigma \in Run(ag)$ iff $\sigma \models G$.

Proof:

1. If $\sigma \in Run(ag)$, then $\sigma \models G$.

$\sigma \in Run(ag)$
implies {Lemma 2}
there exists an infinite suffix h of σ such that $h \in Behaviour(ag)$
iff {$Behaviour(ag) = G$}
there exists an infinite suffix h of σ such that $h \models G$
iff {definition of $\sigma \models G$}
$\sigma \models G$.

2. If $\sigma \models G$, then $\sigma \in Run(ag)$.

$\sigma \models G$
iff {definition of $\sigma \models G$}
there exists an infinite suffix h of σ such that $h \models G$
iff {Behaviour(ag) = G}
there exists an infinite suffix h of σ such that $h \in Behaviour(ag)$
implies {Lemma 1}
$\sigma \in Run(ag)$.

Theorem 1 (Semantic Conditions for Realizability). Let $G \subseteq Hist(V)$ such that $G \neq \phi$. G is realizable by an agent ag iff the following conditions hold:

(1) for all $\sigma_1, \sigma_2 \in Path(V)$, $s \in State(V)$,
if $\sigma_1 \sim_{Voc(ag)} \sigma_2$ then $\sigma_1 + s \models G$ iff $\sigma_2 + s \models G$

(2) for all $\sigma \in Path(V)$, $s_1, s_2 \in State(V)$,
if $s_1 \sim_{Ctrl(ag)} s_2$ then $\sigma + s_1 \models G$ iff $\sigma + s_2 \models G$

(3) for all $h \in History(V)$
if $h \not\models G$ then there exists a finite prefix σ of h such that $\sigma \not\models G$

Proof:

1. The fact that conditions (1) to (3) hold when G is realizable follows directly from Lemma 3 and the properties of agent runs in Section 4.2.6.

1.1. If G is realizable by ag then Condition (1) holds.

Let $\sigma_1, \sigma_2 \in Path(V)$, $s \in State(V)$, such that $\sigma_1 \sim_{Voc(ag)} \sigma_2$.

$\sigma_1 + s \models G$
iff {Behaviour(ag) = G, Lemma 3}
$\sigma_1 + s \in Run(ag)$
iff {Property (1), $\sigma_1 \sim_{Voc(ag)} \sigma_2$}
$\sigma_2 + s \in Run(ag)$
iff {Behaviour(ag) = G, Lemma 3}
$\sigma_2 + s \models G$

1.2. If G is realizable by ag then Condition (2) holds.

Let $\sigma \in Path(V)$, and $s_1, s_2 \in State(V)$, such that $s_1 \sim_{Ctrl(ag)} s_2$.

$\sigma + s_1 \models G$
iff {Behaviour(ag) = G, Lemma 3}
$\sigma + s_1 \in Run(ag)$
iff {Property (2), $s_1 \sim_{Ctrl(ag)} s_2$}
$\sigma + s_2 \in Run(ag)$
iff {Behaviour(ag) = G, Lemma 3}
$\sigma + s_2 \models G$

1.3. If G is realizable by ag then Condition (3) holds.

Let $h \in History(V)$ such that $h \not\models G$.

$h \not\models G$
iff {Behaviour(ag) = G}
$h \notin Behaviour(ag)$
iff {Property (3)}
there exists a finite prefix σ of h such that $\sigma \notin Run(ag)$
iff {Behaviour(ag) = G, Lemma 3}
then there exists a finite prefix σ of h such that $\sigma \not\models G$

2. If Conditions (1) to (3) hold, the G is realizable by ag.

To show that these conditions are sufficient for a goal to be realizable, we exhibit an agent transition system characterized by the following properties:

$s_c \in Init(ag)$ iff $s \models G$ for all $s \in State(V)$ such that $s_{|Ctrl(ag)} = s_c$

$< \sigma_m, s_c > \in Next(ag)$ iff $\sigma+s \models G$ or $\sigma \not\models G$ for all σ such that $\sigma_{|Voc(ag)} = \sigma_m$
for all s such that $s_{|Ctrl(ag)} = s_c$

Steps 2.1 and 2.2 show that the set of behaviours generated by this transition system is equal to the set of histories admitted by the goal. Step 2.3 show that this transition system is well-defined, that is, $Init(ag) \neq \phi$, and Next(ag) is a total relation. (Steps 2.1 and 2.2 do not rely on Step 2.3.)

2.1. If $h \in Behaviour(ag)$ then $h \models G$

By contraposition of Condition (3), we show that $h \models G$, by showing that

$\sigma \models G$ for every finite prefixes σ of h.

The proof is done by induction on the length of the prefixes of h.

(*Initial Case*). $h[0] \models G$

$h \in Behaviour(ag)$
implies {satisfaction of Init(ag)}
$h(0)_{|Ctrl(ag)} \in Init(ag)$
iff {definition of Init(ag)}
$s \models G$ for all $s \in State(V)$ such that $s_{|Ctrl(ag)} = h(0)_{|Ctrl(ag)}$
implies {instantiating s to h(0)}
$h(0) \models G$

(*Inductive Step*). if $h[i] \models G$ then $h[i+1] \models G$ for all $i \in Nat$

$h \in Behaviour(ag)$
implies {satisfaction of Next (ag)}
$< h[i]_{|Voc(ag)} , h(i+1)_{|Ctrl(ag)} > \in Next(ag)$
iff {definition of Next(ag)}
$\sigma+s \models G$ or $\sigma \not\models G$ for all σ such that $\sigma_{|Voc(ag)} = h[i]_{|Voc(ag)}$
for all s such that $s_{|Ctrl(ag)} = h(i+1)_{|Ctrl(ag)}$
implies {instantiating σ to h[i] and s to h(i+1)}
$h[i+1] \models G$ or $h[i] \not\models G$.

2.2. If $h \models G$ then $h \in Behaviour(ag)$

By definition, $h \in Behaviour(ag)$ iff

($Satisfaction\ of\ Init$): $h(0)_{|Ctrl(ag)} \in Init(ag)$
($Satisfaction\ of\ next$): $< h[i]_{|Voc(ag)}\ ,\ h(i+1)_{|Ctrl(ag)} > \in Next(ag)$ for all $i \in Nat$

Each condition is proved in turn.

2.2.1. ($Satisfaction\ of\ Init$): if $h \models G$ then $h(0)_{|Ctrl(ag)} \in Init(ag)$

$h \models G$
implies {from definition of $\sigma \models G$}
$h(0) \models G$
implies {Condition (2)}
$s \models G$ for all $s \in State(V)$ such that $h(0)_{|Ctrl(ag)} = s_{|Ctrl(ag)}$
iff {definition of Init(ag)}
$h(0)_{|Ctrl(ag)} \in Init(ag)$

2.2.2. ($Satisfaction\ of\ Next$): if $h \models G$
then $< h[i]_{|Voc(ag)}\ ,\ h(i+1)_{|Ctrl(ag)} > \in Next(ag)$ for all $i \in Nat$

$h \models G$
implies {from definition of $\sigma \models G$}
$h[i]+h(i+1) \models G$
implies {Condition (1)}
$\sigma+h(i+1) \models G$ for all σ such that $\sigma \sim_{Voc(ag)} h[i]$
implies {Condition (2)}
$\sigma+s \models G$ for all σ such that $\sigma \sim_{Voc(ag)} h[i]$
for all s such that $s \sim_{Ctrl(ag)} h(i+1)$
implies {definition of Next}
$< h[i]_{|Voc(ag)}\ ,\ h(i+1)_{|Ctrl(ag)} > \in Next(ag)$

2.3. We now show that this transition system is well-defined, that is, $Init(ag) \neq \phi$, and $Next(ag)$ is a total relation.

2.3.1. $Init(ag) \neq \phi$

Since $G \neq \phi$, there exists h such that $h \models G$.

Form Step 2.2.1, we have that $h(0)_{|Ctrl(ag)} \in Init(ag)$. Therefore, $Init(ag) \neq \phi$.

2.3.2. $Next(ag)$ is a total relation

Let $\sigma_m \in Path(Voc(ag))$. We prove that there exists a $s_c \in State(Ctrl(ag)$ such that

$< \sigma_m, s_c > \in Next(ag)$.

The proof is done by cases.

Case 1. Suppose that $\sigma \not\models G$, for all $\sigma \in Path(V)$ such that $\sigma_{|Vov(ag)} = \sigma_m$.

By definition of Next(ag),

$< \sigma_m, s_c > \in Next(ag)$ for any $s_c \in State(Ctrl(ag)$

Case 2. Suppose that $\sigma \models G$, for some $\sigma \in Path(V)$ such that $\sigma_{|Voc(ag)} = \sigma_m$.

Since $\sigma \models G$, there exists $h \in History(V)$, suffix of σ, such that $h \models G$.

Let $n = length(\sigma)$. We have $h[n]_{|Voc(ag)} = \sigma_m$.

We define:

$s_c = h(n+1)_{|Ctrl(ag)}$

We show that $< \sigma_m , s_c > \in Next(ag)$ as follows:

$h \models G$
implies {Step 2.2.2}
$< h[n]_{|Voc(ag)} , h(n+1)_{|Ctrl(ag)} > \in Next(ag)$
implies {$h[n]_{|Voc(ag)} = \sigma_m$, and $s_c = h(n+1)_{|Ctrl(ag)}$}
$< \sigma_m , s_c > \in Next(ag)$

Annex B. Proofs of Chapter 5

Definition. Let G be an assertion. Voc(G) is defined as the smallest set of state variables X satisfying the following property:

for all h, h' ∈ Hist(V)
if $h \sim_X h'$, then h |= G iff h' |= G

Example. Let G be defined by

$$P \Rightarrow Q \wedge (R \vee \neg R)$$

Then, Voc(G) = {P, Q}.

Lemma 4. Let G be a goal, and $C \subseteq Voc(G)$ and $M = Voc(G) \backslash C$.

For all h ∈ Hist(V),

$$h \models G \quad \text{iff} \quad (h_{|M}, h_{|C}) \in G_{(M,C)}$$

Proof:

1. If h |= G then$(h_{|M}, h_{|C}) \in G_{(M,C)}$.

h |= G
implies {definition of Voc(ag)}
h' |= G for all h' such that $h' \sim_{Voc(G)} h$
implies {$M \cup C = Voc(G)$}
h' |= G for all h' such that $h'_{|M} = h_{|M}$ and $h'_{|C} = h_{|C}$
iff {definition of $G_{(M,C)}$}
$(h_{|M}, h_{|C}) \in G_{(M,C)}$

2. If $(h_{|M}, h_{|C}) \in G_{(M,C)}$ then h |= G.

$(h_{|M}, h_{|C}) \in G_{(M,C)}$
iff {definition of $G_{(M,C)}$}
h' |= G for all h' such that $h'_{|M} = h_{|M}$ and $h'_{|C} = h_{|C}$
implies {instantiating h' to h}
h |= G

Lemma 5. Condition (iv) of Theorem 2 is equivalent to the following condition (iv'):

For all h_m , $h'_m \in \text{dom } G_{(M,C)}$ and $i \geq 0$
if $i = 0$ or $h'_m[i-1] = h_m[i-1]$
then for all $h_c \in \text{Hist}(C)$
if $(h_m, h_c) \in G_{(M,C)}$
then there exists $h'_c \in \text{Hist}(C)$ such that $h'_c[i] = h_c[i]$ and $(h'_m, h'_c) \in G_{(M,C)}$.

Proof:

1. (iv) implies (iv')

Let h_m , $h'_m \in \text{dom } G_{(M,C)}$ and $i \geq 0$ such that $i = 0$ or $h'_m[i-1] = h_m[i-1]$.

Let $h_c \in \text{Hist}(C)$ such that $(h_m, h_c) \in G_{(M,C)}$.

$(h_m, h_c) \in G_{(M,C)}$
implies {definition of $G_{(M, C)}$ (h_m) [i]}
$h_c[i] \in G_{(M, C)}$ (h_m) [i]
implies {Condition (iv) of Theorem 2}
$h_c[i] \in G_{(M, C)}$ (h'_m) [i]
implies {definition of $G_{(M, C)}$ (h'_m) [i]}
exists $h'_c \in \text{Hist}(C)$ such that $h'_c[i] = h_c[i]$ and $(h'_m, h'_c) \in G_{(M,C)}$

2. (iv') implies (iv)

Let h_m , $h'_m \in \text{dom } G_{(M,C)}$ and $i \geq 0$ such that $i = 0$ or $h'_m[i-1] = h_m[i-1]$.

We show that $G_{(M, C)}$ (h_m) [i] $\subseteq G_{(M, C)}$ (h_m') [i]. The inverse inclusion follows by symmetry.

$\sigma_c \in G_{(M, C)}$ (h_m) [i]
iff {definition of $G_{(M, C)}$ (h_m) [i]}
exists $h_c \in \text{Hist}(C)$ such that $h_c[i] = \sigma_c$ and $(h_m, h_c) \in G_{(M,C)}$
implies {Condition (iv')}
exists $h'_c \in \text{Hist}(C)$ such that $h'_c[i] = \sigma_c$ and $(h'_m, h'_c) \in G_{(M,C)}$
iff {definition of $G_{(M, C)}$ (h'_m) [i]}
$\sigma_c \in G_{(M, C)}$ (h'_m) [i]

Theorem 2 -- Let $G \subseteq Hist(V)$ and ag an agent with monitoring and control capabilities given by Mon(ag) and Ctrl(ag), respectively. G is realizable by ag if, and only if, there exists $C \subseteq Voc(G)$ and $M = Voc(G) \backslash Ctrl(ag)$ such that the following conditions hold:

(i) the agent has sufficient monitoring capabilities

$M \subseteq Voc(ag)$

(ii) the agent has sufficient control capabilities

$C \subseteq Ctrl(ag)$

(iii) $G_{(M,C)}$ is a total relation, that is,

for all $h_m \in Hist(M)$ there exists $h_c \in Hist(C)$ such that $(h_m, h_c) \in G_{(M,C)}$

(iv) $G_{(M,C)}$ does not refer to future values of M, that is, the values at time i of variables in C only depend on the previous values of variables in M up to time i -1.

(v) G is finitely violable, i.e.

for all $h \in History(V)$, if $h \not\models G$ then there exists a finite prefix σ of h such that $\sigma \not\models G$

Condition (iv) is formally captured by the following property requiring that if two arbitrary histories of variables in M are equal up to time i - 1, then they accept the same path of variables in C up to time i:

for all $h_m , h_m' \in dom\ G_{(M,C)}$ and $i \geq 0$
if $h_m'[i-1] = h_m[i-1]$ or $i = 0$ then $G_{(M, C)}(h_m)[i] = G_{(M, C)}(h_m')[i]$

Proof: The structure of the proof is similar to the one of Theorem 1.

1. If G is realizable by ag, then Conditions (i) to (v) hold for some set of variables M, $C \subseteq Voc(G)$.

Let $C = Voc(G) \cap Ctrl(ag)$ and $M = Voc(G) \backslash C$. We show that if G is realizable by ag, then conditions (i) to (v) hold.

1.1. If G is realizable by ag, then $M \subseteq Voc(ag)$.

By definition, $M \subseteq Voc(G)$. We will show that $Voc(G) \subseteq Voc(ag)$.

From Corollary 1 of Chapter 4, we have that

for all $h_1, h_2 \in Hist(V)$,
if $h_1 \sim_{Voc(ag)} h_2$ then $h_1 \in Behaviour(ag)$ iff $h_2 \in Behaviour(ag)$.

Since G is realizable by ag, Behaviour(ag) = G. The above formula is therefore equivalent to:

for all $h_1, h_2 \in Hist(V)$,
if $h_1 \sim_{Voc(ag)} h_2$ then $h_1 \models G$ iff $h_2 \models G$.

By definition, Voc(ag) is the smallest set X satisfying the property:

for all $h, h' \in Hist(V)$
if $h \sim_X h'$, then $h \models G$ iff $h' \models G$

Therefore, $Voc(G) \subseteq Voc(ag)$.

1.2. If $G_{(M, C)}$ is realizable by ag, then $C \subseteq Ctrl(ag)$.

This follows directly from the definition of C, that is, $C = Voc(G) \cap Ctrl(ag)$.

1.3. If G is realizable by ag, then $G_{(M, C)}$ is a total relation.

Let $h_m \in Hist(M)$. We will show the existence of an $h_c \in Hist(C)$ such that $(h_m , h_c) \in G_{(M, C)}$.

Since $Ctrl(ag) \cap M = \phi$, $Init(ag) \neq \phi$, and Next(ag) is a total relation, there exists an history $h \in Hist(V)$ such that:

$$h_{|M} = h_m$$
$$h(0)_{|Ctrl(ag)} \in Init(ag)$$

and for all $i > 0$, the states $h(i)_{|Ctrl(ag)}$ are defined by induction such that:

$$< h[i]_{|Voc(ag)} , h(i+1)_{|Ctrl(ag)}> \in Next(ag).$$

By construction, $h \in Behaviour(ag)$. Therefore, since $Behaviour(ag) = G$, and using Lemma 4, we have that $(h_{|M} , h_{|C}) \in G_{(M, C)}$.

Taking $h_c = h_{|C}$, we have $(h_m , h_c) \in G_{(M, C)}$.

1.4. If G is realizable by ag, then $G_{(M, C)}$ does not refers to the future

Using Lemma 5, we show that $G_{(M, C)}$ does not refer to the future by proving Condition (iv').

Let $h_m , h'_m \in dom\ G_{(M,C)}$ and $i \geq 0$ such that $i = 0$ or $h'_m[i-1] = h_m[i-1]$.

Let $h_c \in Hist(C)$ such that $(h_m, h_c) \in G_{(M,C)}$.

We will show the existence of an $h'_c \in Hist(C)$ such that

$$h'_c[i] = h_c[i] \text{ and } (h'_m , h'_c) \in G_{(M, C)}.$$

Let $h \in Hist(V)$ such that $h_{|M} = h_m$ and $h_{|C} = h_c$. Using Lemma 4, we have that

$$h \models G;$$

and since $Behaviour(ag) = G$, we have that

$$h \in Behaviour(ag).$$

Since $Ctrl(ag) \cap M = \phi$, and Next(ag) is a total relation,
there exists an history $h' \in Hist(V)$ such that:

$$h'_{|M} = h'_m$$
$$h'_{|Ctrl(ag)}[i] = h_{|Ctrl(ag)}[i]$$

and for all $j \geq i$, the states $h'(i)_{|Ctrl(ag)}$ are defined by induction such that:

$$< h'[j]_{|Voc(ag)} , h'(j+1)_{|Ctrl(ag)}> \in Next(ag).$$

Taking $h'_c = h'_{|C}$, we show that $h'_c[i] = h_c[i]$ and $(h'_m , h'_c) \in G_{(M, C)}$.

1.4.1. $h'_c[i] = h_c[i]$.

By definition of h'_c,

$h'_c[i] = h'_{|C}[i]$.

Since $h'_{|Ctrl(ag)}[i] = h_{|Ctrl(ag)}[i]$ and $C \subseteq Ctrl(ag)$, we have:

$h'_{|C}[i] = h_{|C}[i]$.

And, by definition of h,

$h_{|C}[i] = h_c[i]$.

Therefore, $h'_c[i] = h_c[i]$.

1.4.2. $(h'_m , h'_c) \in G_{(M, C)}$

$(h'_m , h'_c) \in G_{(M, C)}$
iff $\{h'_{|M} = h'_m, h'_{|C} = h'_c$, Lemma 4$\}$
$h' \models G$
iff {behaviour(ag) = G}
$h' \in$ Behaviour(ag)

We prove that $h' \in$ Behaviour(ag) by showing that it satisfies the initial condition and the next state relation.

1.4.2.1. *Satisfaction of Init*: $h'(0)_{|Ctrl(ag)} \in$ Init(ag).

$h'(0)_{|Ctrl(ag)} = h(0)_{|Ctrl(ag)} \in$ Init(ag) because $h \in$ Behaviour(ag);

1.4.2.2. *Satisfaction of Next*: $< h'[j]_{|Voc(ag)} , h'(j+1)_{|Ctrl(ag)}> \in$ Next(ag) for all $j \in$ Nat.

For all $j < i$:

$< h'[j]_{|Voc(ag)} , h'(j+1)_{|Ctrl(ag)}> = < h[j]_{|Voc(ag)} , h(j+1)_{|Ctrl(ag)}> \in$ Next(ag)
because $h \in$ Behaviour(ag).

For all $j \geq i$:

$< h'[j]_{|Voc(ag)} , h'(j+1)_{|Ctrl(ag)}> \in$ Next(ag) by definition of h'.

1.5. If $G_{(M, C)}$ is realizable by ag, then G is a ‘safety’ property

This has been proved in Step 1.3 of Theorem 1.

2. If Conditions (i) to (v) hold for some for some set of variables M, $C \subseteq Voc(G)$, then G is realizable by ag.

We define a transition system characterized by the following properties:

$s_c \in Init(ag)$ iff $s \models G$ for all $s \in State(V)$ such that $s_{|Ctrl(ag)} = s_c$

$< \sigma_m, s_c > \in Next(ag)$ iff $\sigma+s \models G$ or $\sigma \not\models G$ for all σ such that $\sigma_{|Voc(ag)} = \sigma_m$ and for all s such that $s_{|Ctrl(ag)} = s_c$

Steps 2.1 and 2.2 show that the set of behaviours generated by this transition system is equal to the set of histories admitted by the goal. Step 2.3 show that this transition system is well-defined, that is, $Init(ag) \neq \phi$, and Next(ag) is a total relation. (Steps 2.1 and 2.2 do not rely on Step 2.3.)

2.1. If $h \in Behaviour(ag)$ then $h \models G$

The proof is similar to Step 2.1 in the proof of Theorem 1; it follows from the definition of Init(ag) and Next(ag), and uses Condition (v) only.

2.2. If $h \models G$ then $h \in Behaviour(ag)$

By definition, $h \in Behaviour(ag)$ iff it satisfies the initial condition and the next state relation. Each condition is proved in turn.

2.2.1. (*Satisfaction of Init*): if $h \models G$, then $h(0)_{|Ctrl(ag)} \in Init(ag)$

$h(0)_{|Ctrl(ag)} \in Init(ag)$
iff {definition of Init(ag)}
$s \models G$ for all $s \in State(V)$ such that $s_{|Ctrl(ag)} = h(0)_{|Ctrl(ag)}$
iff {definition of $s \models G$}
for all $s \in State(V)$ such that $s_{|Ctrl(ag)} = h(0)_{|Ctrl(ag)}$
there exists h' such that $h'(0) = s$ and $h' \models G$

Let $s \in State(V)$ such that

$s_{|Ctrl(ag)} = h(0)_{|Ctrl(ag)}$.

Since $C \subseteq Ctrl(ag)$, we also have that:

$s_{|C} = h(0)_{|C}$ (1)

We will define an h' such that $h'(0) = s$ and $h' \models G$.

We have that:

$h \models G$
implies {Lemma 4}
$(h_{|M}, h_{|C}) \in G_{(M,C)}$
implies {Condition (iv') (instantiating i to 0)}
for all $h'_m \in dom\ G_{(M,C)}$
there exists h'_c such that $h'_c(0) = h_{|C}(0)$ and $(h'_m, h'_c) \in G_{(M,C)}$
implies {Condition (iii), i.e. $dom\ G_{(M,C)} = Hist(M)$}
for all $h'_m \in Hist(M)$
there exists h'_c such that $h'_c(0) = h_{|C}(0)$ and $(h'_m, h'_c) \in G_{(M,C)}$

Instantiating this last formula with an $h'_m \in Hist(M)$ such that

$h'_m(0) = s_{|M}$ (2)

yields that there exists h'_c such that

$h'_c(0) = h_{|C}(0)$ and $(h'_m, h'_c) \in G_{(M,C)}$ (3)

We define $h' \in Hist(V)$ as follows:

$h'(0) = s$ (4)
$h'(i)_{|M} = h'_m(i)$ for all $i \geq 1$
$h'(i)_{|C} = h'_c(i)$ for all $i \geq 1$

We have to show that $h' \models G$.

By construction of h', we have:

$h'_{|M} = h'_m$
because: $h'(0)_{|M} = s_{|M} = h'_m(0)$ {4, 2}
$h'(i)_{|M} = h'_m(i)$ for all $i \geq 1$ {4}
$h'_{|C} = h'_c$
because: $h'(0)_{|C} = s_{|C} = h_{|C}(0) = h'_c(0)$ {4, 1, 3}
$h'(i)_{|M} = h'_m(i)$ for all $i \geq 1$ {4}

And, since $(h'_m, h'_c) \in G_{(M,C)}$, we have:

$(h'_m, h'_c) \in G_{(M,C)}$
implies {$h'_{|M} = h'_m$, and $h'_{|C} = h'_c$}
$(h'_{|M}, h'_{|C}) \in G_{(M,C)}$
implies {Lemma 4}
$h' \models G$

2.2.2. (*Satisfaction of Next*):
if $h \models G$, then $< h[i]_{|Voc(ag)} , h(i+1)_{|Ctrl(ag)} > \in Next(ag)$ for all $i \in Nat$

Let $i \in Nat$. We have:

$< h[i]_{|Voc(ag)} , h(i+1)_{|Ctrl(ag)} > \in Next(ag)$
iff {definition of Next(ag)}
$\sigma+s \models G$ or $\sigma \not\models G$ for all σ such that $\sigma_{|Voc(ag)} = h[i]_{|Voc(ag)}$
for all s such that $s_{|Ctrl(ag)} = h(i+1)_{|Ctrl(ag)}$

Let σ such that

$\sigma_{|Voc(ag)} = h[i]_{|Voc(ag)}$

Since $M \subseteq Voc(ag)$, and $C \subseteq Ctrl(ag)$, we also have that:

$\sigma_{|M} = h[i]_{|M}$ (5)
$\sigma_{|C} = h[i]_{|C}$ (6)

Let s such that

$s_{|Ctrl(ag)} = h(i+1)_{|Ctrl(ag)}$

Since $C \subseteq Ctrl(ag)$, we also have that:

$$s_{|C} = h(i+1)_{|C} \qquad (7)$$

We will show that $\sigma+s \models G$ by finding an h' such that $h'[i] = \sigma$ and $h'(i+1) = s$ and $h' \models G$.

We have:

$h \models G$
implies {Lemma 4}
$(h_{|M}, h_{|C}) \in G_{(M,C)}$
implies {Condition (iv')}
for all $h'_m \in \text{dom } G_{(M,C)}$ such that $h'_m[i] = h_{|M}[i]$
there exists h'_c such that $h'_c[i+1] = h_{|C}[i+1]$ and $(h'_m, h'_c) \in G_{(M,C)}$
implies {Condition (iii), i.e. $\text{dom } G_{(M,C)} = Hist(M)$}
for all $h'_m \in Hist(M)$ such that $h'_m[i] = h_{|M}[i]$
there exists h'_c such that $h'_c[i+1] = h_{|C}[i+1]$ and $(h'_m, h'_c) \in G_{(M,C)}$

Instantiating this last formula with an $h'_m \in Hist(M)$ such that

$$h'_m[i] = h_{|M}[i] \qquad (8)$$
$$h'_m(i+1) = s_{|M} \qquad (9)$$

yields that there exists h'_c such that

$$h'_c[i+1] = h_{|C}[i+1] \qquad (10)$$
$$(h'_m, h'_c) \in G_{(M,C)}$$

We define $h' \in Hist(V)$ as follows:

$$h'[i] = \sigma \qquad (11)$$
$$h'(i+1) = s$$
$$h'(j)_{|M} = h'_m(j) \quad \text{for all } j \geq i+1$$
$$h'(j)_{|C} = h'_c(i) \quad \text{for all } j \geq i+1$$

We have to show that $h' \models G$.

By construction of h', we have:

$h'_{|M} = h'_m$
because: $h'[i]_{|M} = \sigma_{|M} = h_{|M}[i] = h'_m[i]$ {11, 5, 8}
$h'(i+1)_{|M} = s_{|M} = h'_m(i+1)$ {11, 9}
$h'(j)_{|M} = h'_m(j)$ for all $j \geq i+1$ {11}
$h'_{|C} = h'_c$
because: $h'[i]_{|C} = \sigma_{|C} = h_{|C}[i] = h'_c[i]$ {11, 6, 10}
$h'(i+1)_{|C} = s_{|C} = h(i+1)_{|C} = h'_c(i+1)$ {11, 7, 10}
$h'(i)_{|M} = h'_m(i)$ for all $j \geq i+1$ {11}

Since $(h'_m, h'_c) \in G_{(M,C)}$, we have:

$(h'_m, h'_c) \in G_{(M,C)}$
implies $\{h'_{|M} = h'_m$, and $h'_{|C} = h'_c\}$
$(h'_{|M}, h'_{|C}) \in G_{(M,C)}$
implies {Lemma 4}
$h' \models G$

2.3. The proofs that Init(ag) is not empty and that Next(ag) is a total relation are similar to the corresponding proofs for Theorem 1.