Handling Obstacles in Goal-Oriented Requirements Engineering

Axel van Lamsweerde, *Member*, *IEEE*, and Emmanuel Letier

Abstract—Requirements engineering is concerned with the elicitation of high-level goals to be achieved by the envisioned system, the refinement of such goals and their operationalization into specifications of services and constraints and the assignment of responsibilities for the resulting requirements to agents such as humans, devices, and software. Requirements engineering processes often result in goals, requirements, and assumptions about agent behavior that are too ideal; some of them are likely not to be satisfied from time to time in the running system due to unexpected agent behavior. The lack of anticipation of exceptional behaviors results in unrealistic, unachievable, and/or incomplete requirements. As a consequence, the software developed from those requirements will not be robust enough and will inevitably result in poor performance or failures, sometimes with critical consequences on the environment. This paper presents formal techniques for reasoning about obstacles to the satisfaction of goals, requirements, and assumptions elaborated in the requirements engineering process. A first set of techniques allows obstacles to be generated systematically from goal formulations and domain properties. A second set of techniques allows resolutions to be generated once the obstacles have been identified thereby. Our techniques are based on a temporal logic formalization of goals and domain properties; they are integrated into an existing method for goal-oriented requirements elaboration with the aim of deriving more realistic, complete, and robust requirements specifications. A key principle in this paper is to handle exceptions at requirements engineering time and at the goal level, so that more freedom is left for resolving them in a satisfactory way. The various techniques proposed are illustrated and assessed in the context of a real safety-critical system.

Index Terms—Goal-oriented requirements engineering, high-level exception handling, obstacle-based requirements transformation, defensive requirements specification, specification refinement, lightweight formal methods.

1 Introduction

Requirements engineering (RE) is the branch of software engineering concerned with the real world goals for, functions of, and constraints on software systems. It is also concerned with the relationship of these factors to precise specifications of software behavior and to their evolution over time and across software families. This general definition, borrowed from [87], stresses the leading part played by goals during requirements elaboration. Goals drive the elaboration of requirements to support them [78], [15], [80]; they provide a completeness criterion for the requirements specification—the specification is complete if all stated goals are met by the specification [86]; they provide a rationale for requirements—a requirement exists because of some underlying goal which provides a base for it [15], [83]; goals represent the roots for detecting conflicts among requirements and for resolving them eventually [76], [49]; goals are generally more stable than the requirements to achieve them [3]. In short, requirements "implement" goals much the same way as programs implement design specifications.

Goals are to be achieved by the various agents operating together in the *composite* system; such agents include software components that exist or are to be developed, external devices, and humans in the environment [22], [27]. The elicitation of functional and nonfunctional goals, their organization into a coherent structure and their operationalization into requirements to be assigned to the various agents is, thus, a central aspect of requirements engineering [16], [65]. Various techniques have been proposed to support this process. *Qualitative reasoning* techniques may be used to determine the degree to which high-level goals are satisficed/denied by lower-level goals and requirements [64]. When goals can be formalized, *formal reasoning* techniques are expected to do more. For example, the correctness of goal refinements may be verified [17]; more constructively, such refinements may be derived formally [16], [24], [17]. Formal goal models may be used to detect and resolve conflicts among goals [49]. Planning techniques may be used to generate admissible scenarios showing that some desirable goal is not achieved by the system specified and propose resolution actions [4], [27]. Conversely, declarative goal specifications may be inferred inductively from operational specifications of scenarios [50].

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 [47], [71], [26]. This general problem is not really handled by current requirements elaboration methods.

• The authors are with the Département d'Ingénierie Informatique, Université Catholique de Louvain, 2 Place Sainte Barbe, B-1348 Louvain-la-Neuve, Belgium. E-mail: {avl, eletier}@info.ucl.ac.be.

Manuscript received 13 Mar. 1999; revised 20 Dec. 1999; accepted 28 Jan. 2000.
Recommended for acceptance by A. Romanovsky.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number 111492.

Consider an ambulance dispatching system, for example; a first-sketch goal such as

Achieve[MobilizedAmbulancePromptlyAtIncident]

is overideal and likely to be violated from time to time—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; and so forth. 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 e-mail 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 on the environment.

The purpose of this paper is to introduce systematic techniques for deidealizing goals, assumptions, and requirements, and to integrate such techniques in a goal-oriented requirements elaboration method in order to derive more complete and realistic requirements from which more robust systems can be built.

Our approach is based on the concept of *obstacle* first introduced in [71]. 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 gets true the goal may not be achieved. Thus, the term "obstacle" is 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 [54] or *threats* that may obstruct security goals [1]; or in later phases of the software lifecycle—such as *faults* that may prevent a program from achieving its specification [14], [29].

The paper 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 operators 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., WrongBeliefAbout Deadline or ReviewRequestLost (under responsibility of Reviewer agents), UnprocessablePostscriptFile (under responsibility of Author agents), etc. 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 may no longer occur, etc.

A key principle here is to handle abnormal agent behavior at requirements engineering time and *at the goal level*. This principle is consistent with recommendations from analysis of software requirements errors [55]. Exception handling techniques are usually introduced at later stages of the software lifecycle, such as architectural design or programming, 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 [2], [9], [70], [13], [79], [40], [14], [7], [29]. In contrast, we perform systematic obstacle analysis at the much earlier stage of requirements engineering, from goal formulations, so that more freedom is left on adequate ways of handling obstacles to goals—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 fairly different.

The integration of obstacle analysis into the requirements engineering process is detailed in the paper in the context of the KAOS methodology for goal-oriented requirements elaboration [16], [47], [17]. In [49], we have shown that obstacle analysis can be seen as a degenerate case of conflict analysis; an obstacle amounts to a condition for conflict between N goals within the domain under consideration, where $N = 1$. As a consequence, there are generic similarities between the respective identification and resolution techniques. 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. As will be seen in the paper, the generic identification/resolution mechanisms yield different instantiations and specializations for obstacle analysis and for conflict analysis.

The rest of the paper is organized as follows: Section 2 summarizes some background material on KAOS that will be used in the sequel. Section 3 introduces obstacles to goals and provides a formal characterization of this concept, including the notion of completeness of a set of obstacles. Section 4 discusses a modified goal-oriented requirements elaboration process that integrates obstacle analysis. Section 5 presents techniques for generating obstacles from goal formulations. Section 6 then presents techniques for transforming goals, requirements and/or assumptions so as to resolve the obstacles generated. The various techniques presented in the paper are illustrated and assessed in Section 7 by an obstacle analysis of a real safety-critical system for which failure stories have been reported [51], [28]. We discuss some related work in Section 8 before concluding in Section 9.

2 Goal-Oriented RE with KAOS

The KAOS methodology is aimed at supporting the whole process of requirements elaboration—from the high-level goals to be achieved to the requirements, objects, and operations to be assigned to the various agents in the composite system. The methodology provides a specification language, an elaboration method, and tool support. To make the paper self-contained, we recall some of the features that will be used later in the paper; see [16], [47], [17], [18] for details.

2.1 Concepts and Terminology

An **object** is a thing of interest in the composite system whose instances may evolve from state to state. Objects are characterized by attributes and invariant assertions. They may be organized in inheritance hierarchies. An *entity* is an autonomous object. A *relationship* is an object dependent on other objects it links. An *event* is an instantaneous object.

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/postconditions, which capture additional strengthenings to ensure that the requirements are met.

An **agent** is an active object that acts as processor for some operations. An agent *performs* an operation if it is allocated to it; the agent *monitors/controls* an object if the states of the object are observable/controllable by it. Agents may be humans, devices, programs, etc.

A **goal** is an objective the composite system should meet; it captures a set of desired behaviors of the composite system. *AND-refinement* links relate a goal to a set of subgoals (called refinement); this means that satisfying all subgoals in the refinement is a sufficient condition for satisfying the goal. *OR-refinement* links relate a goal to an alternative set of refinements; this means that satisfying one of the refinements is a sufficient condition for satisfying the goal. The goal refinement structure for a given system can be represented by an AND/OR directed acyclic graph [66]. Goals *concern* the objects to which they refer. A goal may additionally be characterized by a *priority* attribute whose values specify the extent to which the goal is mandatory or optional.

Goals are classified according to the category of requirements they will drive about the agents concerned. Functional goals result in functional requirements. For example, SatisfactionGoals are functional goals concerned with satisfying agent requests; InformationGoals are goals concerned with keeping agents informed about object states. Likewise, nonfunctional goals result in nonfunctional requirements. For example, AccuracyGoals are nonfunctional goals concerned with maintaining the consistency between the state of objects in the environment and the state of their representation in the software; other subcategories include SafetyGoals, SecurityGoals, PerformanceGoals, and so on.

Goal refinement ends up when terminal goals are reached; these are goals assignable to individual agents. A terminal goal can thus be formulated in terms of states controllable by some individual agent. A *requirement* is a terminal goal assigned to an agent in the software-to-be. An *assumption* is a terminal goal assigned to an agent in the environment. Unlike requirements, assumptions cannot be enforced in general. Terminal goals are in turn AND/OR *operationalized* by operations and objects through strengthenings of their domain pre/postconditions and invariants, respectively, and through obligations expressed by trigger conditions. Alternative ways of assigning responsible agents to a terminal goal are captured through OR *responsibility* links. The actual assignment of an agent to the operations that operationalize the terminal goal is captured in corresponding *performs* links.

A **domain property** is a property about objects or operations in the environment which holds independently of the software-to-be. Domain properties include physical laws [69], regulations, constraints imposed by environmental agents [54]—in short, indicative statements of domain knowledge [36], [88]. In KAOS, domain properties are captured by domain invariants attached to objects and by domain pre/postconditions attached to operations.

A **scenario** is a domain-consistent sequence of state transitions controlled by corresponding agent instances; domain consistency means that the operation associated with a state transition is applied in a state satisfying its domain precondition together with the various domain invariants attached to the corresponding objects, with a resulting state satisfying its domain postcondition.

2.2 The Specification Language

Each construct in the KAOS language has a two-level generic structure: an outer semantic net layer [10] for *declaring* a concept, its attributes and its various links to other concepts; an inner formal assertion layer for *formally defining* the concept. The declaration level is used for conceptual modeling (through a concrete graphical syntax), requirements traceability (through semantic net navigation) and specification reuse (through queries) [18]. The assertion level is optional and used for formal reasoning [16], [17], [59], [26], [49], [50].

The generic structure of a KAOS construct is instantiated to specific types of links and assertion languages according to the specific type of the concept being specified. For example, consider the following goal specification for an ambulance dispatching system:

Goal *Achieve* [AmbulanceMobilization]
Concerns Call, Ambulance, Incident
Refines AmbulanceIntervention
RefinedTo IncidentFiled, AmbulanceAllocated, AllocatedAmbulanceMobilized
InformalDef *For every responded call about an incident, an ambulance able to arrive at the incident scene within 11 minutes should be mobilized. The ambulance mobilization time should be less than 3 minutes [ORCON standard, 3005].*
FormalDef $\forall$ cl: Call, inc: Incident
Responded (cl) $\wedge$ About (cl, inc)
$\Rightarrow \Diamond_{\leq 3m} \exists\, a$: Ambulance
Mobilized (a, inc)
$\wedge \bullet$ [Available (a) $\wedge$ TimeDist (a.Loc, inc.Loc) $\leq$ 11]

The declaration part of this specification introduces a concept of type "goal," named AmbulanceMobilization, stating a target property that should eventually hold ("Achieve" verb), referring to objects such as Call or Ambulance, refining the parent goal AmbulanceIntervention, refined into subgoals IncidentFiled, AmbulanceAllocated, and AllocatedAmbulanceMobilized and defined by some informal statement. (The semantic net layer is represented in textual form in this paper for reasons of space limitations; the reader may refer to [18] to see what the alternative graphical concrete syntax looks like.)

The optional assertion part in the specification above defines the goal Achieve[AmbulanceMobilization] in formal terms using a real-time temporal logic inspired from [45]. In this paper, we will use the following classical operators for temporal referencing [57]:

$\circ$ (in the next state)	$\bullet$ (in the previous state)
$\Diamond$ (some time in the future)	$\blacklozenge$ (some time in the past)
$\Box$ (always in the future)	$\blacksquare$ (always in the past)
$\mathcal{W}$ (always in the future *unless*)	$\mathcal{U}$ (always in the future *until*)

Formal assertions are interpreted over historical sequences of states. Each assertion is in general satisfied by some sequences and falsified by some other sequences. The notation

$$(H, i) \models P$$

is used to express that assertion P is satisfied by history H at time position i (i $\in$ T), where T denotes a linear temporal structure assumed to be discrete for sake of simplicity. We will also use the notation H $\models$ P for (H, 0) $\models$ P.

States are global; the *state* of the composite system at some time position i is the aggregation of the local states of all its objects at that time position. The state of an individual object instance *ob* at some time position is defined as a mapping from *ob* to the set of values of all *ob*'s attributes and links at that time position. In the context of KAOS requirements, an historical sequence of states defines a behavior produced by a scenario.

The semantics of the above temporal operators is then defined as usual [57], e.g.,

$(H, i) \models \circ P$ iff $(H, next(i)) \models P$
$(H, i) \models \Diamond P$ iff $(H, j) \models P$ for some $j \geq i$
$(H, i) \models \Box P$ iff $(H, j) \models P$ for all $j \geq i$

$(H, i) \models P\mathcal{U}Q$ iff there exists a $j \geq i$ such that $(H, j) \models Q$ and for every k, $i \leq k < j$, $(H, k) \models P$

$(H, i) \models P\mathcal{W}Q$ iff $(H, i) \models P\mathcal{U}Q$ or $(H, i) \models \Box P$

Note that $\Box$ P amounts to P$\mathcal{W}$**false**. We will also use the standard logical connectives $\wedge$ (and), $\vee$ (or), $\neg$ (not), $\rightarrow$ (implies), $\leftrightarrow$ (equivalent), $\Rightarrow$ (strongly implies), $\Leftrightarrow$ (strongly equivalent), with

$P \Rightarrow Q$ iff $\Box (P \rightarrow Q)$
$P \Leftrightarrow Q$ iff $\Box (P \leftrightarrow Q)$

Note thus that there is an implicit outer $\Box$-operator in every strong implication.

Besides the agent-related classification of goals introduced in Section 2.1, goals in KAOS are also classified according to the pattern of temporal behavior they capture:

Achieve:	$C \Rightarrow \Diamond T$
Cease:	$C \Rightarrow \Diamond \neg T$
Maintain:	$C \Rightarrow T\mathcal{W}N$, $C \Rightarrow \Box T$
Avoid:	$C \Rightarrow \neg T\mathcal{W}N$, $C \Rightarrow \Box \neg T$

In these patterns, C, T, and N denote some current, target, and new condition, respectively. (We avoid the classical safety/liveness terminology here to avoid confusions with SafetyGoals.)

In requirements engineering we often need to introduce real-time restrictions. Bounded versions of the above temporal operators are therefore introduced, in the style advocated by [45], such as

$\Diamond_{\leq d}$ (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 T is enriched with a metric domain D and a temporal distance function *dist:* TxT $\rightarrow$ D which has all desired properties of a metrics [45]. We will take

T:	the set of naturals
D:	{ d | there exists a natural *n* such that d = n$\times$*u*}, where *u* denotes some chosen time unit
dist(i, j):	$\mid j - i \mid \times u$

Multiple units can be used—e.g., *s* (second), *m* (minute, see the AmbulanceMobilization goal above), *d* (day), etc; these are implicitly converted into some smallest unit. The o-operator then yields the nearest subsequent time position according to this smallest unit.

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

$(H, i) \models \Diamond_{\leq d} P$ iff $(H, j) \models P$ for some $j \geq i$ with dist(i, j) $\leq$ d

$(H, i) \models \Box_{<d} P$ iff $(H, j) \models P$ for all $j \geq i$ such that dist(i, j) < d

In the above goal declaration of AmbulanceMobilization, the conjunction of the assertions formalizing the subgoals IncidentFiled, AmbulanceAllocated, and AllocatedAmbulanceMobilized must entail the formal assertion of the parent goal AmbulanceMobilization they refine together. Every formal goal refinement thus generates a corresponding proof obligation [17].

In the formal assertion of the goal AmbulanceMobilization, the predicate Mobilized(a,inc) means that, in the current state, an instance of the Mobilized relationship links variables a and inc of sort Ambulance and Incident, respectively. The Mobilized relationship and Ambulance entity are defined in other sections of the specification, e.g.,

Entity Ambulance
  Has Loc: *Location*, Dest: *Location*,
  ...
Relationship Mobilized
  Links Ambulance {**card** 0:1}, Incident {**card** 0:N}
  InformalDef *An ambulance is mobilized for some incident iff a crew is assigned to it and its destination is the incident's location.*
  DomInvar $\forall$a: Ambulance, inc: Incident
    Mobilized (a, inc) $\Leftrightarrow$ ($\exists$ cr: Crew) Assigned (cr, a) $\wedge$ a.Dest = inc.Loc

The Crew type might in turn be declared by

Agent Crew
  Has Free: *Boolean*, Paramedics: *Boolean*
  ...

In the declarations above, Loc is declared as an attribute of the entity Ambulance (this attribute was used in the formal definition of the goal AmbulanceMobilization); Free is declared as an attribute of the agent Crew.

As mentioned earlier, operations are specified formally by pre and postconditions in the state-based style [72], e.g.,

Operation Mobilize
  Input Incident {**arg** inc}
  Output Ambulance {**res** *amb*}, Mobilized
  DomPre $\neg$ ($\exists$ a: Ambulance) Mobilized (a, inc)
  DomPost Mobilized (*amb*, inc)

Note that the invariant defining the Mobilized relationship is not a requirement, but a domain property; it specifies what being mobilized does precisely mean in the domain. The pre- and postcondition of the operation Mobilize above are domain properties as well; they capture corresponding elementary state transitions in the domain, namely, from a state where no ambulance is mobilized to a state where some ambulance is mobilized. The software requirements are found in the terminal goals assigned to agents in the software-to-be and in the additional pre-, post-, and trigger conditions that need to strengthen the corresponding domain conditions in order to ensure all such goals [16], [47]. Assuming the AmbulanceMobilization goal is assigned to the dispatching software, one would derive the following strengthenings from the above formal assertion for that goal:

Operation Mobilize
  ...
  RequiredPre for AmbulanceMobilization:
    Available (*amb*) $\wedge$ TimeDist (*amb*.Loc, inc.Loc)$\leq$ 11
  RequiredTrig for AmbulanceMobilization:
    $\blacksquare_{\leq 3m}$($\exists$ cl: Call) Responded (cl) $\wedge$ About (cl, inc)

The trigger condition captures an obligation to trigger the operation as soon as the condition gets true and provided the domain precondition is true. The specification will be consistent provided the trigger condition and required preconditions are together true in the operation's initial state.

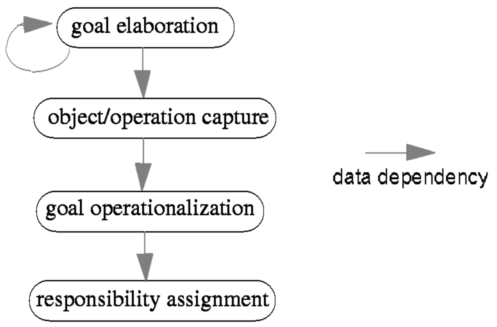

Fig. 1. Goal-oriented requirements elaboration.

2.3 The Elaboration Method

Fig. 1 outlines the major steps that may be followed to elaborate KAOS specifications from high-level goals. (Section 4 will discuss how obstacle analysis gets into this process model.)

- *Goal elaboration.* Elaborate the goal AND/OR structure by defining goals and their refinement links until assignable goals are reached. The process of identifying goals, defining them precisely, and relating them through refinement links is in general a combination of top-down and bottom-up subprocesses [47]; offspring goals are identified by asking HOW questions about goals already identified whereas parent goals are identified by asking WHY questions about goals and operational requirements already identified.
- *Object capture.* Identify the objects involved in goal formulations, define their conceptual links, and describe their domain properties by invariants.
- *Operation capture.* Identify object state transitions that are meaningful to the goals. Goal formulations refer to desired or forbidden states that are reachable through state transitions; the latter correspond to applications of operations. The principle is to specify such state transitions as domain pre- and postconditions of operations thereby identified and to identify the agents that could perform these operations.
- *Operationalization.* Derive strengthened pre-, post-, and trigger conditions on operations and strengthened invariants on objects, in order to ensure that all terminal goals are met. A number of formal derivation rules are available to support the operationalization process [16].
- *Responsibility assignment.* 1) Identify alternative responsibilities for terminal goals; 2) make decisions among refinement, operationalization, and responsibility alternatives, so as to reinforce nonfunctional goals [64]—e.g., goals related to reliability, performance, cost reduction, load reduction, etc; 3) assign the operations to agents that can commit to guarantee the terminal goals in the alternatives

selected. The boundary between the system and its environment is obtained as a result of this process and the various terminal goals become requirements or assumptions dependent on the assignment made.

The steps above are ordered by data dependencies; they may be running concurrently, with possible backtracking at every step.

3 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.

3.1 Obstacles to Goals

Semantically speaking, a goal defines a set of desired behaviors, where a behavior is a temporal sequence of states (see Section 2.2). A positive scenario is a sequence of state transitions, controlled by corresponding agent instances, that produces such a desired behavior (see Section 2.1). Goal refinement yields sufficient subgoals for the goal to be achieved.

Likewise, an obstacle defines a set of undesirable behaviors; a negative scenario produces a behavior in this set. 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. $\{O, Dom\} \not\models \mathbf{false}$ (*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. In terms of behaviors, the consistency condition is semantically equivalent to the following condition:

2′. There exists a scenario S producing a behavior H such that

$$H \models O \qquad (feasibility).$$

This condition now states that the obstacle specification is satisfiable through one behavior at least, produced by a (domain-consistent) scenario of agent cooperation.

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]
RefinedTo SatisfiedWhenAvailable, CopyEventuallyAvailable, RequestPending
FormalDef $\forall$ bor: Borrower, b: Book
Requesting (bor, b)
$\Rightarrow \Diamond$ ($\exists$ bc: BookCpy) [Copy (bc, b) $\land$ Gets (bor, bc)]

An obstructing obstacle to that goal might be specified by the following assertion:

$\Diamond$ $\exists$ bor: Borrower, b: Book
Requesting (bor, b)
$\land \Box$ ($\forall$ bc: BookCpy) [Copy (bc, b) $\Rightarrow \neg$ 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$\land\neg$ Q)). This obstructing assertion is satisfiable, e.g., through the classical starvation scenario [19] 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 $\land$ PressureTooLow $\Rightarrow$ AlarmRaised

It is easy to see that condition 1 would be satisfied by the candidate obstacle

PressureTooLow $\land$ Startup $\Rightarrow \neg$ AlarmRaised
$\land$ $\Diamond$ [Running $\land$ PressureTooLow $\land$ Startup]

which logically entails the negation of the goal above; however, this candidate is inconsistent with the 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 in the paper.

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.

3.2 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 Safety goals.

A set of obstacles $O_1, \ldots, O_n$ to goal G in *Dom* is *domain-complete* with respect to G iff the following condition holds:

$\{\neg O_1, \ldots, \neg O_n, \mathrm{Dom}\} \models G$ (*domain-completeness*)

This condition intuitively means that if none of the obstacles in the set may occur then the goal is necessarily 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 [37] after a real plane incident. The goal

MovingOnRunway $\Rightarrow$ ReverseThrustEnabled

can be AND-refined, using the milestone refinement pattern [17], 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 dependent on whether the following property is known in the domain:

MovingOnRunway
$\wedge$ WheelsOut $\wedge \neg$ WheelsBlocked $\wedge \neg$ Aquaplaning
$\Rightarrow$ WheelsTurning
(D4)

Obstacle completeness thus really depends on what valid properties are known in the domain.

3.3 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 $O_1, \ldots, O_n$ is an *AND-refinement* of an obstacle O iff the following conditions hold:

1. $\{ O_1 \wedge O_2 \wedge \ldots, \wedge O_n, \text{Dom}\} \models O$ (*entailment*)
2. $\{ O_1 \wedge O_2 \wedge \ldots, \wedge O_n, \text{Dom}\} \not\models$ **false** (*consistency*).

In general, one is interested in minimal AND-refinements, in which case the following condition has to be added:

3. for all i: $\{\wedge_{j \neq i} O_j, \text{Dom}\} \not\models O$ (*minimality*).

A set of obstacles $O_1, \ldots, O_n$ is an *OR-refinement* of an obstacle O iff the following conditions hold:

1. for all i: $\{O_i, \text{Dom}\} \models O$ (*entailment*)
2. for all i: $\{O_i, \text{Dom}\} \not\models$ **false** (*consistency).*

In general, one is interested in complete OR-refinements in which case the domain-completeness condition has to be added:

3. $\{\neg O_1 \wedge \ldots \wedge \neg O_n, \text{Dom}\} \models \neg O$ (*completeness).*

In the plane landing example above, the set {O1, O2, O3} is a complete OR-refinement of the higher-level obstacle *N-Ass* with respect to a domain comprising all the properties listed.

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, j: $\{O_i, O_j, \text{Dom}\} \models$ **false** (*disjointness).*

Section 5.3 will present a rich set of complete and disjoint obstacle refinement patterns.

Chaining the definitions in Sections 3.1 and 3.3 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.

3.4 Classifying Obstacles

As mentioned in Section 2.1, goals are classified by type of requirements they will drive about the agents concerned. For each goal category, corresponding obstacle categories may be defined. For example,

- Nonsatisfaction obstacles are obstacles that obstruct the satisfaction of agent requests (that is, Satisfaction goals);
- Noninformation 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 [1], WrongBelief obstacles form a subcategory of Inaccuracy obstacles, etc.

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 5.4.

3.5 Goal Obstruction vs. Goals Divergence

In the context of handling conflicts between multiple goals, we have introduced in [49] 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

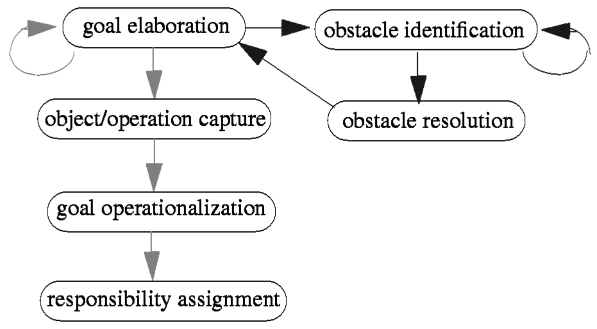

Fig. 2. Obstacle analysis in goal-oriented requirements elaboration.

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 as well (see Section 5.4). As will be seen below, the common generic principles for problem identification/resolution yield specific instantiations and specializations for obstacle analysis. For example, the goal regression procedure can be simplified (see Section 5.1); the completion procedure is specific to obstacle analysis (see Section 5.2); obstruction refinement patterns are different from divergence patterns (see Section 5.3).

4 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 [47], [71]. 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 outlined in Section 2.3 is depicted in Fig. 2. (As in Fig. 1, the arrows indicate data dependencies.) The main difference is the *obstacle analysis loop* introduced in the upper right part.

During elaboration of the goal graph by elicitation and by refinement, obstacles are generated from goal specifications. Such obstacles may be recursively refined—see the right circle arrow in Fig. 2. (Section 5 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 [21]: the generation of alternative resolutions and the selection of one among the alternatives considered. (Section 6 will discuss different operators 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 subobstacles in order to identify precise circumstances whose feasibility might be assessed through negative scenarios of agent behavior. 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 6, the generation of alternative resolutions corresponds to the application of different strategies for resolving obstacles. The strategies include obstacle elimination, with substrategies such as obstacle prevention, goal substitution, agent substitution, goal deidealization, or object transformation; obstacle reduction; and obstacle tolerance, with substrategies such as obstacle mitigation or goal restoration. (Some of these strategies have been studied in other contexts of handling problematic situations—e.g., deadlocks in parallel systems [12]; exceptions and faults in fault-tolerant systems [2], [13], [40], [29]; feature interaction in telecommunication systems [42]; inconsistencies in software development [67]; or conflicts between requirements [77], [49]).

- The range of strategies to consider and the selection of a specific strategy to apply will depend on the likelihood of occurrence of the obstacle, on the impact of such occurrence (in the number of goals being obstructed by the obstacle), and on the severity of the consequences of such 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 paper.
- 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 [21].

Goal-obstacle analysis iteration. *When should the intertwined processes of goal elaboration and obstacle analysis stop?*

The goal-obstacle analysis loop in Fig. 2 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.

Some of the issues above will be addressed in a more specific way for the obstacle analysis of the London Ambulance System in Section 7.

5 Generating Obstacles

According to the definition in Section 3.1, 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.

Step 2 corresponds to a classical consistency checking problem in logic; it can be carried out using deductive verification techniques (e.g., [58], [68]). Alternatively, one may check the satisfiability of the candidate obstacle in the domain by finding out some negative scenario (see the feasibility condition in Section 3.1). This can be done manually [71], with some formal support as shown below, or using automated techniques based on planning [27] or model checking [34], [60], [38]; in the latter case some operational model of the system needs to be available.

We therefore concentrate on Step 1 and present techniques for deriving candidate obstacles whose domain consistency/ feasibility needs to be subsequently checked. We successively discuss:

- a formal calculus of preconditions for obstruction,
- the use of formal obstruction patterns to shortcut formal derivations, and
- the use of identification heuristics based on obstacle classifications as a cheap, informal alternative to formal techniques.

5.1 Regressing Goal Negations

The first technique is based on the obstruction condition defining an obstacle in Section 3.1. 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 [30] for declarative representations. Variants of this procedure have been used in AI planning [85], in explanation-based learning [46], and in requirements engineering to identify divergent goals [49]. 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 to meetings they are aware of and which fit their constraints:

Goal *Achieve* [InformedParticipantsAttendance]
FormalDef $\forall$ m: Meeting, p: Participant
Intended (p, m) $\wedge$ Informed (p, m) $\wedge$ Convenient (p, m)
$\Rightarrow \Diamond$ Participates (p, m)

The initialization step of the regression procedure consists of taking the negation of that 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 decribed in Section 5.3, we will thereby obtain two subobstacles that could be named MeetingPostponedIndefinitely and LastMinuteImpediment, respectively. Scenarios satisfying their respective assertion are straightforward in this case.

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 [46] can be described as follows:

```
Initial step:
  take O := ¬ G
Inductive step:
  let A ⇒ C be the domain rule selected,
    with C matching some subformula L in O whose
    occurrences in O are all positive;
  then $\mu$ := mgu(L, C);
       O := O[L / A.$\mu$]
```

This procedure relies on the following definitions and notations:

- for a formula scheme φ(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 [57, p. 203]:

If all occurrences of u in φ(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, dependent on whether the obstacles obtained are meaningful and precise enough 1) to easily identify scenarios satisfying them and 2) to see appropriate ways of resolving them through strategies discussed in Section 6.

In the example above, only one iteration was performed. Regressing obstacle (O1) above further through a domain property like

Convenient (p, m) $\Rightarrow$ m.Date **in** p.Constraints
$\wedge$ m.Location **in** p.Constraints

would have produced finer subobstacles 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′) $\forall$ m: Meeting, p: Participant
Participates (p, m) $\Rightarrow \exists$ M: Belief_p(m.Date = M) $\wedge$ m.Date = M

The deontic Belief_{ag} construct in this formalization is often used to capture Accuracy goals and Inaccuracy obstacles; it is linked to the Knows_{ag} construct by the following property:

$$\mathrm{Knows}_{ag}(P) \equiv \mathrm{Belief}_{ag}(P) \wedge P$$

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.

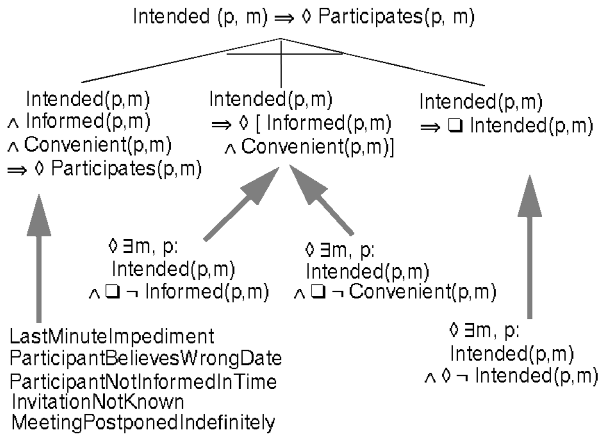
Intended (p, m) ⇒ ◊ Participates(p, m)
Intended(p,m) ∧ Informed(p,m) ∧ Convenient(p,m) ⇒ ◊ Participates(p,m)
Intended(p,m) ⇒ ◊ [Informed(p,m) ∧ Convenient(p,m)]
Intended(p,m) ⇒ □ Intended(p,m)
◊ ∃m, p: Intended(p,m) ∧ □ ¬ Informed(p,m)
◊ ∃m, p: Intended(p,m) ∧ □ ¬ Convenient(p,m)
LastMinuteImpediment
ParticipantBelievesWrongDate
ParticipantNotInformedInTime
InvitationNotKnown
MeetingPostponedIndefinitely
◊ ∃m, p: Intended(p,m) ∧ ◊ ¬ Intended(p,m)

Fig. 3. Goal refinement and obstacles derived by regression.

Regressing the goal negation (NG) above through property (D′) now yields the following new obstacle:

(O2) $\Diamond \exists$ m: Meeting, p: Participant
Intended (p, m) $\wedge$ Informed (p, m) $\wedge$ Convenient (p, m)
$\wedge \Box \forall$ M: $\neg$ [Belief_p (m.Date = M) $\wedge$ 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 ParticipantNotInformedInTime, InvitationNotKnown, etc.

The examples above exhibit a **simplified procedure** for generating obstacles to *Achieve* goals of the form C $\Rightarrow \Diamond$ T:

1. Negate the goal, which yields a pattern $\Diamond$ (C $\wedge \Box \neg$ 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 and 3, recursively.

A dual version of this simplified procedure can be used for goals having the *Maintain* patterns C $\Rightarrow$ T, C $\Rightarrow \Box$ T, or C $\Rightarrow$T*W*N. For the plane landing example in Section 3.2, 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*, the requirements engineer may *incrementally elicit necessary conditions* for *T* to hold 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, Fig. 3 shows a goal AND-refinement tree, derived by instantiation of a frequent refinement pattern from [17], together with corresponding obstacles that were generated by regression (universal quantifiers have been left implicit).

5.2 Completing a Set of Obstacles

The domain-completeness condition in Section 3.2 suggests a procedure for completing a set of obstacles $O_1, \ldots, O_n$ already identified for some goal *G*.

As noted in Section 3.1, *G* has the general form $\Box$ GC whereas O_i has the general form $\Diamond OC_i$. The completion procedure can be described as follows:

1. Form the complementary assertion
$$O^* = \Diamond(\neg GC \wedge \neg OC_1 \wedge \ldots \wedge \neg OC_k);$$
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*, O1, ..., Ok} obtained by Step 1 satisfies the domain-completeness condition in Section 3.2 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 $\Diamond$ (P $\wedge$ P1) and a domain property is found having the form P $\Rightarrow$ P1. A one-step regression then yields O = $\Diamond$ P.

Back to the plane landing example in Section 3.2, Step 1 of the completion procedure applied to the assumption

MovingOnRunway $\Rightarrow$ WheelsTurning

and the obstructing obstacles

$\Diamond$ (MovingOnRunway $\wedge$ $\neg$ WheelsOut) (O1)
$\Diamond$ (MovingOnRunway $\wedge$ WheelsBlocked) (O2)
$\Diamond$ (MovingOnRunway $\wedge$ Aquaplaning) (O3)

yields

O* = $\Diamond$ (MovingOnRunway $\wedge$ $\neg$ WheelsTurning
$\wedge$ WheelsOut $\wedge$ $\neg$ WheelsBlocked $\wedge$ $\neg$ Aquaplaning)

This candidate obstacle is inconsistent with the domain if property (D4) is found in *Dom* (see Section 3.2). 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.

5.3 Using Obstruction Refinement Patterns

As introduced in Section 3.3, obstacles may be AND/OR-refined into subobstacles. AND-refinements yield more "primitive" obstacles, that is, obstacles for which 1) negative scenarios can be found more easily to show their feasibility and 2) 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 5.1 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. Fig. 3 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 MeetingNever-Convenient, respectively). The latter subobstacles may be refined in turn. Similarly, the obstacle ParticipantBelieves-WrongDate that was derived in Section 5.1 could be OR-refined into alternative subobstacles like WrongDateCom-municated, ParticipantConfusesDates, etc.

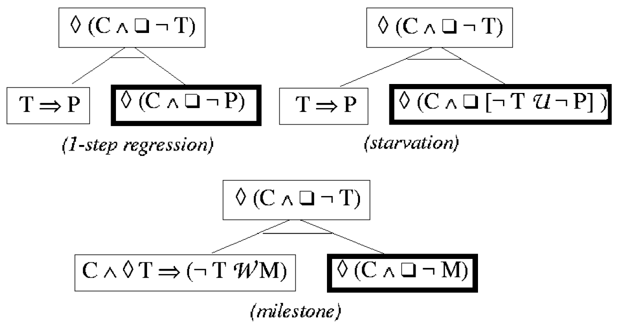
$\Diamond (C \wedge \Box \neg T)$
$T \Rightarrow P$
$\Diamond (C \wedge \Box \neg P)$
(1-step regression)
$\Diamond (C \wedge \Box \neg T)$
$T \Rightarrow P$
$\Diamond (C \wedge \Box [\neg T \, \mathcal{U} \neg P])$
(starvation)
$\Diamond (C \wedge \Box \neg T)$
$C \wedge \Diamond T \Rightarrow (\neg T \, \mathcal{W} M)$
$\Diamond (C \wedge \Box \neg M)$
(milestone)

Fig. 4. AND-refinement patterns for obstacles to goals C $\Rightarrow$ $\Diamond$ T.

The AND/OR refinement of obstacles may be seen as a formal, *goal-oriented* form of fault-tree analysis [54] or threat-tree analysis [1]. 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 5.1 is a first technique to achieve this; alternatively, one may use obstacle refinement patterns to shortcut the formal deriva-tions involved in the regression procedure.

The general principle is similar to goal refinement patterns [17] and divergence detection patterns [49]. 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 5.1. 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 [58]. As we will see, the notion of correctness is different for AND- and OR-refinement patterns. We discuss them successively.

5.3.1 AND-Refinement Patterns

Figs. 4, 5, and 6 show a sample of frequent AND-refinement patterns for obstacles that obstruct *Achieve* and *Maintain* goals, respectively.

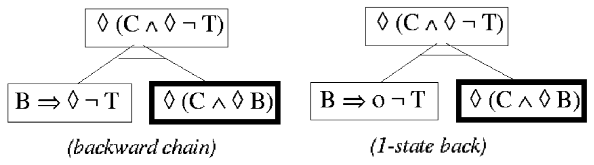

Fig. 5. AND-refinement patterns for obstacles to goals $C \Rightarrow \Box T$.

The *root* assertion in each AND-tree corresponds to the negation of the goal being obstructed. (Remember that there is an implicit $\Box$-operator in every strong implication; this causes the outer $\Diamond$-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 $T \Rightarrow P$. In the milestone pattern, it defines a necessary milstone 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 patterns in Figs. 4 and 6 correspond to the regression procedure in Section 5.1 where only one iteration is performed.

As an example of using the starvation pattern in Fig. 4, consider a general resource management system and the goal

$\forall$ u: User, r: Resource
Requesting (u, r) $\Rightarrow \neg \exists$ u′$\neq$ u: Allocated (r, u′)

The domain property

Allocated (r, u) $\Rightarrow \neg \exists$ u′$\neq$ u: Allocated (r, u′)

suggests reusing the starvation pattern with instantiations

C: Requesting (u, r)
T: Allocated (r, u) , *P*: $\neg \exists$ u′$\neq$ u: Allocated (r, u′)

The following starvation obstacle has been thereby derived:

$\Diamond \exists$ u: User, r: Resource
Requesting (u, r)
$\wedge \Box$ [$\neg$ Allocated(r, u) $\mathcal{U} \exists$ u′$\neq$ u: Allocated (r, u′)]

As an example of using the 1-step regression pattern in Fig. 6, consider the LAS ambulance dispatching system [51] 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

$\forall$ a: Ambulance, inc: Incident
Allocation (a, inc)$\Rightarrow$Allocation (a, inc)$\mathcal{W}$Intervention (a, inc)

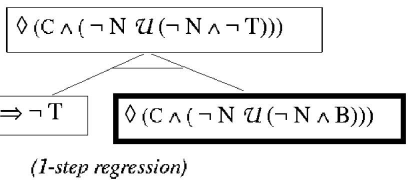

Fig. 6. AND-refinement patterns for obstacles to goals $C \Rightarrow T\mathcal{W}N$.

TABLE 1
Patterns of Obstacles to Goals $R \Rightarrow \Diamond S$

	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	$R \wedge \Diamond S \Rightarrow \Diamond [P \wedge (P \mathcal{W} S)]$	$\Diamond [R \wedge (\neg S \mathcal{U} \Box \neg P)]$
	$R \wedge \Diamond S \Rightarrow \Diamond [P \wedge (P \mathcal{W} S)]$	$\Diamond [R \wedge (\neg S \mathcal{U} (\neg S \wedge \Box \neg P))]$

We know from the domain that an ambulance can be allocated to at most one incident at a time:

Allocation (a, inc) $\Rightarrow \neg \exists$ inc′ $\neq$ 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*: $\exists$ inc′ $\neq$ inc: Allocation (a, inc′)

The following subobstacle is thereby derived:

$\Diamond \exists$ a: Ambulance, inc: Incident
Allocation (a, inc)
$\wedge \neg$ Intervention (a, inc) $\mathcal{U}$
$\neg$ Intervention (a, inc) $\wedge \exists$ inc′ $\neq$ 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 1, 2, 3, and 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 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 4 may be seen to correspond to the backward construction of a fault-tree from a state machine [74]; *p* and *q* are intended to be state predicates there.

All AND-refinement patterns in Tables 1, 2, 3, and 4 were proved correct using STeP [58]—by this we mean that the

TABLE 2
Patterns of Obstacles to Goals $P \Rightarrow \bullet Q$

	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	$C \Rightarrow \mathbf{o} \neg Q$	$\Diamond [P \wedge \Diamond C]$

entailment and consistency conditions in Section 3.3 were formally verified.

Section 7 will illustrate the use of various patterns from Tables 1, 2, 3, and 4.

5.3.2 Complete OR-Refinement Patterns

Fig. 7 shows a pattern for refining the obstruction of an *Achieve* goal $R \Rightarrow \Diamond S$ into a complete set of disjoint alternative subobstacles (see Section 3.3 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.

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 [47]:

$\forall$ 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 Fig. 7 with the following instantiations:

C: ConstraintsRequested(p,m) *T:* ConstraintsProvided(p,m)
M: RequestReaches(p,m)

The milestone pattern then generates the following domain property:

TABLE 3
Patterns of Obstacles to Goals $\Box (P \rightarrow Q)$

	Assertions	Subobstacle
	$Q \wedge C \Rightarrow \mathbf{o} \neg Q$, $P \wedge C \Rightarrow \mathbf{o} P$	$\Diamond [P \wedge Q \wedge C]$
	$Q \wedge C \Rightarrow \mathbf{o} \neg Q$, $\neg P \wedge C \Rightarrow \mathbf{o} P$	$\Diamond [\neg P \wedge Q \wedge C]$
	$\neg P \wedge C \Rightarrow \mathbf{o} P$, $\neg Q \wedge C \Rightarrow \mathbf{o} \neg Q$	$\Diamond [\neg P \wedge \neg Q \wedge C]$

TABLE 4
Patterns of Obstacles to Goals $\Box q$

	Assertion	Subobstacle
back state	$p \Rightarrow \mathbf{o} \neg q$	$\Diamond p$

$\forall$ m: Meeting, p: Participant
ConstraintsRequested(p,m) $\wedge \Diamond$ ConstraintsProvided (p,m)
$\Rightarrow$ [$\neg$ ConstraintsProvided(p,m) $\mathcal{W}$RequestReaches(p,m)]

together with a complete set of alternative subobstacles to the goal above:

$\Diamond \exists$ m: Meeting, p: Participant
ConstraintsRequested (p,m) $\wedge \Box \neg$ RequestReaches (p,m)

or

$\Diamond \exists$ m: Meeting, p: Participant
ConstraintsRequested (p,m) $\wedge$
$\neg$ RequestReaches (p,m) $\mathcal{U}$
RequestReaches (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.

These examples suggest that *the more an obstacle is refined the closer one gets to an explicit scenario*. Obstacle refinement patterns may thus be used for suggesting feasible scenarios as well.

A more extensive set of complete and disjoint OR-refinement patterns is given in Tables 5 and 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 Section 7.

All OR-refinement patterns in Tables 5 and 6 were proved correct using STeP [58]—by this we mean that the entailment, consistency, disjointness, and domain-completeness conditions in Section 3.3 were formally verified. In the latter case, the formulas in the assertion column were taken as generic domain property forming *Dom*.

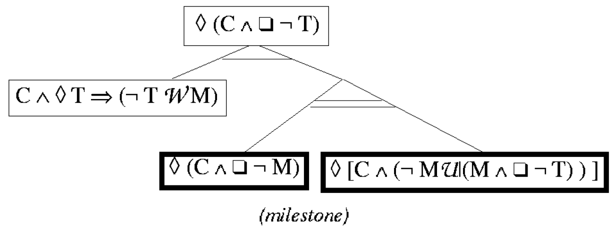

Fig. 7. OR-refinement pattern for obstacles to goals $C \Rightarrow \Diamond T$.

TABLE 5
Obstacle OR-Refinement for Goals $R \Rightarrow \Diamond S$

Assertion	Obstacle	Obstacle	Obstacle
$S \Leftrightarrow P \land Q$	$\Diamond [R \land \Box \neg P]$	$\Diamond [R \land \Box \neg Q]$	$\Diamond [R \land P \land Q \land \Box \neg (P \land Q)]$
$S \Rightarrow P$	$\Diamond [R \land \Box \neg P]$	$\Diamond [R \land \Diamond P \land \Box \neg S]$	
$S \Rightarrow P$	$\Diamond [R \land \neg S \mathcal{U} \Box \neg P]$	$\Diamond [R \land \Diamond P \land \Box \neg S]$	
$S \Rightarrow P$	$\Diamond [R \land \Box (\neg S \mathcal{U} \neg P)]$	$\Diamond [R \land \Diamond (P \mathcal{W} (P \land S)) \land \Box \neg S]$	
$R \land \Diamond S \Rightarrow P$	$\Diamond [R \land \neg P]$	$\Diamond [R \land P \land \Box \neg S]$	
$R \land \Diamond S \Rightarrow P \mathcal{W} S$	$\Diamond [R \land \neg S \mathcal{U} (\neg P \land \neg S)]$	$\Diamond [R \land P \mathcal{W} S \land \Box \neg S]$	
$R \land \Diamond S \Rightarrow P \mathcal{W} (P \land S)$	$\Diamond [R \land \neg S \mathcal{U} \neg P]$	$\Diamond [R \land \Box \neg S \land P \mathcal{W} (P \land S)]$	
$R \land \Diamond S \Rightarrow \neg S \mathcal{W} M$	$\Diamond [R \land \Box \neg M]$	$\Diamond [R \land \neg M \mathcal{U} (M \land \Box \neg S)]$	
$B \Rightarrow \Box \neg S$	$\Diamond [R \land \neg S \mathcal{U} B]$	$\Diamond [R \land \Box \neg S \land \neg B \mathcal{W} S]$	
$P \Rightarrow \Box \neg S \land \blacksquare \neg S$	$\Diamond [R \land P]$	$\Diamond [R \land \Box \neg S \land \Box \neg P]$	

5.4 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 [54] or, more generally, to the use of safety checklists [39], [83].

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 2.1). Here are a few examples to illustrate the approach.

***If** an agent has to monitor/control 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 heuristics might have helped identifying obstacles like ParticipantBelievesWrongDate—see Section 5.1; 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:

- WrongInfoProvided. The necessary information provided by another agent about the object state is incorrect (possible refinements for this obstacle are, e.g., too high or too low values for an object attribute).
- InfoCorrupted. The information from the provider has been corrupted by another agent.
- 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 [71].
- 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 the ModeConfusion obstacle subcategory in which pilot agents become confused about what the cockpit software agent is doing; obstacles in this category receive increasing attention as

TABLE 6
Obstacle OR-Refinement for Goals $\Box (P \rightarrow Q)$

Assertion	Obstacle	Obstacle
$Q \Leftrightarrow Q1 \land Q2$	$\Diamond [R \land \neg Q1]$	$\Diamond [R \land \neg Q2]$

they have been recognized to be responsible for a significant number of critical incidents [11].

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 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* InformationGoal *category is considered,* ***then*** *consider obstacles like*

MessageUndelivered,
MessageDeliveredAtWrongPlace,
MessageDeliveredAtWrongTime,
MessageCorrupted,

and so on.

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 [59].

6 Resolving Obstacles

The generated obstacles need to be resolved in some way or another. As discussed in Section 4, 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 *strategies* that may be applied. They can be classified into three broad classes dependent on whether the obstacle is eliminated (Section 6.1), reduced (Section 6.2), or tolerated (Section 6.3).

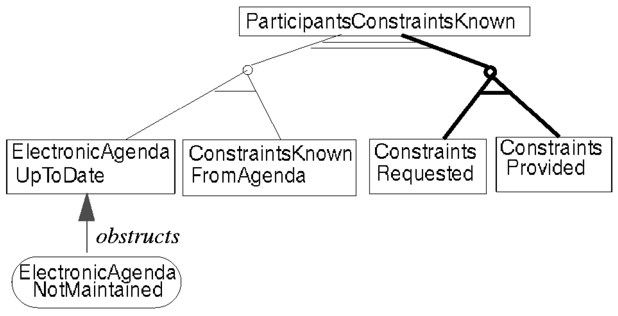
ParticipantsConstraintsKnown
ElectronicAgenda UpToDate
ConstraintsKnown FromAgenda
Constraints Requested
Constraints Provided
obstructs
ElectronicAgenda NotMaintained

Fig. 8. Goal substitution.

Some of these strategies have been studied in other contexts of handling problematic situations—e.g., deadlocks in parallel systems [12]; exceptions and faults in fault-tolerant systems [2], [13], [40], [29]; feature interaction in telecommunication systems [42]; inconsistencies in software development [67]; or conflicts between requirements [77], [49]. The objective here is to specialize such strategies 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.

6.1 Obstacle Elimination

Eliminating an obstacle requires one of the conditions defining an obstacle in Section 3.1 to be inhibited; the obstruction should be avoided or the obstacle should be made inconsistent/unfeasible within the domain. The strategies below address one of the conditions or the other.

6.1.1 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 the 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 Fig. 8); 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 the participants and the email system).

Choosing an alternative goal refinement will in general result in a different design for the composite system.

6.1.2 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 fairly 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 e-mail, phone calls, etc.

In the electronic reviewing example, one could introduce a software agent for checking that no occurrence of the reviewer's name is found in the review (to overcome the obstacle NonAnonymousReview), a software agent for checking destination tables (to overcome the obstacle MessageSentToWrongPerson), etc.

Agent substitution may entail goal substitution and vice versa.

6.1.3 Obstacle Prevention

This strategy resolves the obstruction by *adding a new goal* requiring that the obstacle be avoided.

Remember that a goal G has the general form $\Box$ GC whereas an obstacle O to G has the general form $\Diamond$OC. To prevent O from being ever satisfied, the following *Avoid* goal is thus introduced:

$$G^*: \ \Box \neg \text{OC}$$

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 Fig. 3. The prevention strategy yields the new goal Avoid [MeetingForgotten]. The latter may then be refined into a requirement Achieve [MeetingReminded] under responsibility of the meeting 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 it possible to infer the obstacle O by regression. In such cases, the domain theory will be updated instead of the goal structure.

Obstacle anticipation is a substrategy 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:

$$\text{OC} \Rightarrow \blacksquare_{\leq d} \text{ P}$$

In such a case, the obstacle prevention goal may be refined by introducing the subgoal

$$G^*: \ \text{P} \Rightarrow \Diamond_{\leq d} \neg \text{ P}$$

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.

6.1.4 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 on an example first.

Consider the obstacle ParticipantNotInformedInTime in Fig. 3 which obstructs the goal

$$\text{Intended (p, m)} \land \text{Informed (p, m)} \land \text{Convenient (p, m)} \Rightarrow \Diamond \text{ Participates (p, m)}$$

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:

$$\text{Intended(p,m)} \land \text{Informed}\mathbf{InTime}\text{(p,m)} \land \text{Convenient(p,m)} \Rightarrow \Diamond \text{ Participates (p, m)}$$

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:

$$\text{Informed}\mathbf{InTime}\text{(p, m)} \equiv \blacksquare_{\leq(\text{m.Date}-\text{Nd})} \text{ Informed (p, m)}$$

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 Fig. 3 will produce a strengthened goal in the middle of the tree, namely,

$$\text{Intended(p,m)} \Rightarrow \Diamond[\text{Informed}\mathbf{InTime}\text{(p,m)} \land \text{Convenient(p,m)}]$$

The deidealization procedure is similar to the one used for weakening divergent goals [49]. It is simpler here as only one goal assertion has to be considered in the weakening process. The procedure has two steps:

1. *Weaken* the goal specification to obtain a more liberal version that covers the obstacle. Syntactic 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 [23] may be seen as a particular form of syntactic 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 [17].

Goal deidealization patterns may also be used as formal support to the deidealization process. Given the obstructed goal and the obstructing obstacle, they yield deidealized

TABLE 7
Deidealization Patterns for *Achieve* Goals

Goal	Obstacle	Deidealized goal
$R \Rightarrow \Diamond S$	$\Diamond [R \wedge \neg P]$	$R \wedge P \Rightarrow \Diamond S$
$R \Rightarrow \Diamond S$	$\Diamond [R \wedge \Box \neg P]$	$R \wedge (P \mathcal{W} S) \Rightarrow \Diamond S$
$R \Rightarrow \Diamond S$	$\Diamond [R \wedge (\neg S\, \mathcal{U} \neg P)]$	$R \wedge (P \mathcal{W} S) \Rightarrow \Diamond S$
$R \Rightarrow \Diamond S$	$\Diamond [R \wedge (\neg S\, \mathcal{U} \Box \neg P)]$	$R \wedge \Box \Diamond P \Rightarrow \Diamond S$
$R \Rightarrow \Diamond S$	$\Diamond [R \wedge \Box (\neg S\, \mathcal{U} \neg P)]$	$R \wedge \Diamond (P\, \mathcal{W} (P \wedge S)) \Rightarrow \Diamond S$

versions of the goal. To illustrate the approach, Table 7 gives some patterns for some of the obstacles from Table 1.

At the end of Section 5.3.1, we considered the resource management *Achieve* goal

$$\Diamond \text{ u: User, r: Resource} \\ \text{Requesting (u, r)} \Rightarrow \Diamond \text{ Allocated (r, u)},$$

and we generated the starvation obstacle

$$\Diamond \exists \text{ u: User, r: Resource} \\ \text{Requesting (u, r)} \\ \wedge \Box [\neg \text{ Allocated (r, u)}\ \mathcal{U}\ \exists u' \neq u\text{: Allocated (r, u')}]$$

The goal and the starvation obstacle match the last row of Table 7; we thereby generate the deidealized goal specification

$$\forall \text{ u: User, r: Resource} \\ \text{Requesting (u,r)} \wedge \Diamond [\neg \exists u' \neq u\text{:Allocated (r,u')}] \\ \mathcal{W} \text{ Allocated (r,u)} \\ \Rightarrow \Diamond \text{ Allocated (r, u)}$$

The new goal version states that if the user requests the resource and the resource is subsequently kept unallocated unless allocated to her/it, then the resource is eventually allocated to her/it. The new condition $P\mathcal{W}S$ 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.

6.1.5 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 3.1) or no longer obstructing the goal (see the obstruction condition in Section 3.1).

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 videoconferencing 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 5.1).

6.2 Obstacle Reduction

The difference between this class of strategies and the previous one is that here one tries to *reduce the occurrences of the obstacle* instead of eliminating them completely.

Strategies 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, etc.

6.3 Obstacle Tolerance

In cases where the obstacle cannot be thoroughly avoided or where avoiding it is simply too costly or not worthy, one may specify which behaviors will be admissible or tolerated in the presence of the obstacle.

6.3.1 Goal Restoration

A first strategy 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. Thus, this new goal takes the *Achieve* form

$$G^*: \quad OC \Rightarrow \Diamond G$$

This strategy could be followed for the obstacle PaperLost that obstructs the goal Achieve[ReviewReturned]. A subgoal refining the restoration goal will be Achieve [LostPaperResent].

6.3.2 Obstacle Mitigation

Another alternative strategy 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.

Weak mitigation consists in ensuring some weakened version G' of the obstructed goal G whenever the obstacle condition OC becomes true. Thus, a weak mitigation goal has the form

$$G^*: \quad OC \Rightarrow G'$$

where G' is a deidealized version of G obtained using the specification transformations described in Section 6.1.4.

To illustrate this, consider the obstacle LastMinute Impediment generated in Section 5.1. The introduction of the weak mitigation goal

Achieve [ImpedimentNotified]

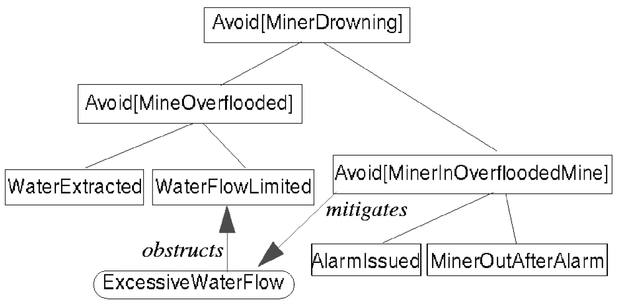

Fig. 9. Obstacle mitigation.

will ensure a weaker version of the goal InformedParticipantsAttendance in Section 5.1, namely,

Intended (p, m) ∧ Informed (p, m) ∧ Convenient (p, m)
⇒ ◊ [Participates (p, m) ∨ *Excused (p, m)*]

(Note that in this case an obstacle prevention alternative to such weak mitigation would yield a goal like Achieve [MeetingReplanned].)

Strong mitigation consists in ensuring some parent goal G' of G whenever the obstacle condition OC becomes true, in spite of G being obstructed. Thus a strong mitigation goal has the form

G^* :   OC ⇒ G′

where the obstructed goal G is a subgoal of G'.

Fig. 9 illustrates this on a mine pump system example [41]. The goal Avoid[MinerInOverfloodedMine] strongly mitigates the obstacle ExcessiveWaterFlow that obstructs the goal WaterFlowLimited by guaranteeing that the parent goal Avoid[MinerDrowning] will be satisfied.

The distinction between strong and weak mitigation somewhat corresponds, at the requirements engineering level, to two different, sometimes confused notions of fault tolerance [13]: one where the program meets its specification in spite of faults, and the other where the program meets a weaker version of the specification.

6.3.3 Do-Nothing

For noncritical 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.

7 Obstacle Analysis for a Real Safety-Critical System

The purpose of this section is to illustrate and assess the various techniques presented in this paper through obstacle analysis of a real safety-critical system for which failure stories have been reported [51], [28].

From the documentation available in the Inquiry Report on the London Ambulance System [51], we reverse engineered the goal graph for this system. We started from the commonsense, high-level goal IncidentResolved and refined it progressively. In addition, a number of goals were elicited from explicit or implicit formulations in the Inquiry Report. Formalizing goal specifications and domain properties and applying formal refinement patterns [17] led us to find out missing goals. Objects and their relationships, operations, and domain properties emerged gradually as goals were refined. Agents were identified as active objects among them—the Computer-Aided Despatch software (CAD), the Automatic Vehicle Location System (AVLS), the resource allocator (RA), the control assistant who handles emergency calls (CA), the radio operator, the communication infrastructure, the station printer, the mobile data terminal (MDT), and the ambulance crew. Goal refinement terminated when requirements assigned to the CAD and assumptions assigned to the agents in the environment were obtained as terminal goals. Figs. 10, 11, and 12 provide excerpts from the goal structure. Note the importance of Accuracy goals (bottom of Fig. 10 and Fig. 12).

Obstacles were then derived systematically for each terminal goal. Many of them were formalized; a mix of regression, obstruction patterns and informal heuristics from Section 5 were used. We then compared 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 occured during those failures (notably, Inaccuracy problems), they also cover many other problems that could (but did not) occur—see the comparison tables below. Finally, we explored the space of possible resolutions by application of the strategies discussed in Section 6.

7.1 Obstacle Generation

Let us illustrate some of the formal derivations first. Consider the terminal goal *IncidentResolvedByIntervention* appearing at level 2 of the goal tree in Fig. 9:

Goal *Achieve* [IncidentResolvedByIntervention]
UnderResponsibility AmbulanceCrew
Refines IncidentResolved
FormalDef ∀ a: Ambulance, inc: Incident
Intervention (a, inc) ⇒ ◊ Resolved (inc)

Applying the regression procedure, we negate this goal to produce the high-level obstacle IncidentNotResolvedByIntervention:

◊ a: Ambulance, inc: Incident
Intervention (a, inc) ∧ □ ¬ Resolved (inc)

We regress this obstacle through domain properties that provide necessary conditions for incident resolution:

Resolved (inc) ⇒
(∀ p: Patient) Injured (p, inc) →
(∀ r: Resource) CriticallyNeeds (p, r) →
(∃ ru: ResourceUnit) Unit (ru, r) ∧ UsedOn (ru, p)

and

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:

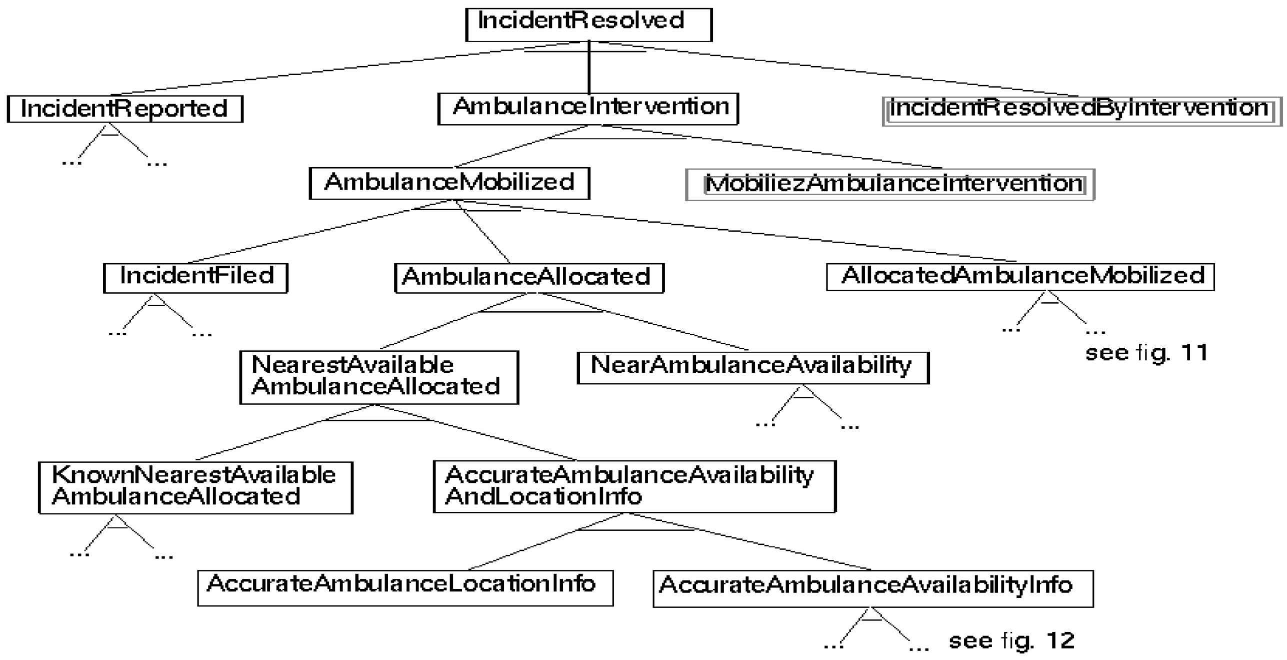
IncidentResolved
IncidentReported
AmbulanceIntervention
IncidentResolvedByIntervention
AmbulanceMobilized
MobiliezAmbulanceIntervention
IncidentFiled
AmbulanceAllocated
AllocatedAmbulanceMobilized
see fig. 11
NearestAvailable AmbulanceAllocated
NearAmbulanceAvailability
KnownNearestAvailable AmbulanceAllocated
AccurateAmbulanceAvailability AndLocationInfo
AccurateAmbulanceLocationInfo
AccurateAmbulanceAvailabilityInfo
see fig. 12

Fig. 10. Refinement of the LAS root goal.

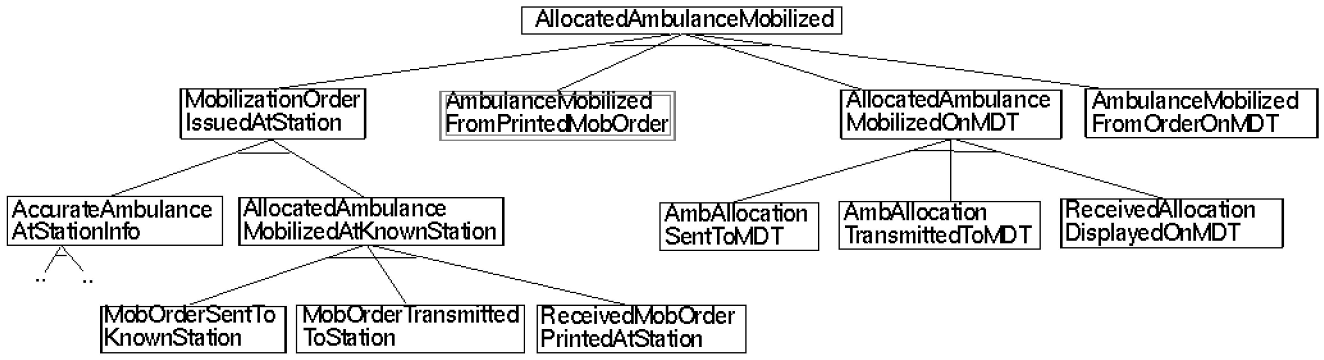
AllocatedAmbulanceMobilized
MobilizationOrder IssuedAtStation
AmbulanceMobilized FromPrintedMobOrder
AllocatedAmbulance MobilizedOnMDT
AmbulanceMobilized FromOrderOnMDT
AccurateAmbulance AtStationInfo
AllocatedAmbulance MobilizedAtKnownStation
AmbAllocation SentToMDT
AmbAllocation TransmittedToMDT
ReceivedAllocation DisplayedOnMDT
MobOrderSentTo KnownStation
MobOrderTransmitted ToStation
ReceivedMobOrder PrintedAtStation

Fig. 11. Refinement of the goal *AllocatedAmbulanceMobilized.*

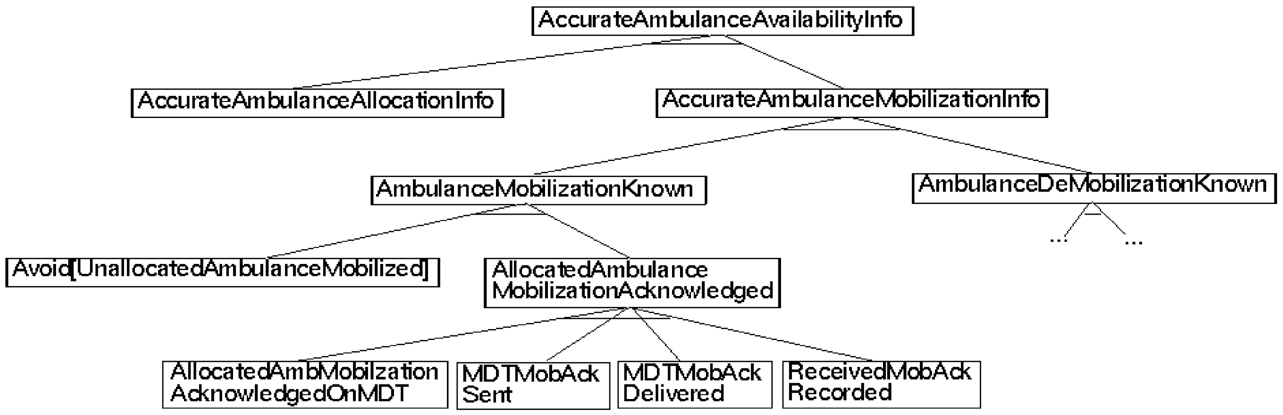
AccurateAmbulanceAvailabilityInfo
AccurateAmbulanceAllocationInfo
AccurateAmbulanceMobilizationInfo
AmbulanceMobilizationKnown
AmbulanceDeMobilizationKnown
Avoid[UnallocatedAmbulanceMobilized]
AllocatedAmbulance MobilizationAcknowledged
AllocatedAmbMobilzation AcknowledgedOnMDT
MDTMobAck Sent
MDTMobAck Delivered
ReceivedMobAck Recorded

Fig. 12. Refinement of the goal *AccurateAmbulanceAvailabilityInfo*.

Obstacle CriticalCareNotGivenToPatient
FormalDef $\Diamond \exists$ a: Ambulance, inc: Incident
Intervention (a, inc)
$\wedge \Box$ ($\exists$p: Patient, r: Resource)
Injured (p, inc) $\wedge$ CriticallyNeeds (p, r)
$\wedge \neg$ (ru: ResourceUnit) Unit (ru, r) $\wedge$ UsedOn (ru, p)

and

Obstacle PatientNotAdmittedToHospital
FormalDef $\Diamond \exists$ a: Ambulance, inc: Incident
Intervention (a, inc)
$\wedge \Box \exists$ p: Patient
Injured (p, inc) $\wedge$ ($\neg \exists$ h: Hospital) AdmittedAt (p, h)

Regressing the first subobstacle CriticalCareNotGivenToPatient through the domain property

Intervention (a, inc)
$\wedge$ Injured (p, inc) $\wedge$ UsedOn (ru, p)
$\Rightarrow$ InAmbulance (ru, a)
$\wedge \neg$ ($\exists$ p′: Patient) p′ $\neq$ p $\wedge$ UsedOn (ru, p′)

yields the new subobstacle:

Obstacle InsufficientResourceInAmbulance
FormalDef $\Diamond\ \exists$ a: Ambulance, inc: Incident
Intervention (a, inc)
$\wedge \Box \exists$ p: Patient, r: Resource
Injured (p, inc) $\wedge$ CriticallyNeeds (p, r)
$\wedge$ Intervention (a, inc)
$\wedge$ ($\forall$ ru: ResourceUnit) Unit (ru, r) $\rightarrow$
InAmbulance (ru, a) $\rightarrow$
($\exists$ p′: Patient) p′ $\neq$ p $\wedge$ 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 $\Diamond\ \exists$ a: Ambulance, inc: Incident
Intervention (a, inc)
$\wedge \Box \exists$ p: Patient, r: Resource
Injured (p, inc) $\wedge$ CriticallyNeeds (p, r)
$\wedge$ Intervention (a, inc)
$\wedge$ ($\exists$ ru: ResourceUnit) Unit (ru, r) $\wedge$ InAmbulance (ru,a)
$\wedge \neg$ ($\exists$ p′: Patient) p′ $\neq$ p $\wedge$ UsedOn (ru, p′)

Further refinement of the latter subobstacle by, e.g., use of the heuristics in Section 5.3, 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.

Consider now the terminal goal *MobilizedAmbulanceIntervention* appearing at level 3 of the goal tree in Fig. 9:

Goal*Achieve* [MobilizedAmbulanceIntervention]
UnderResponsibility AmbulanceCrew
Refines AmbulanceIntervention
FormalDef $\forall$ a: Ambulance, inc: Incident
Mobilized (a, inc) $\wedge$ TimeDist (a.Loc, inc.Loc) $\leq$ 11
$\Rightarrow \Diamond_{\leq 11\mathrm{m}}$ Intervention (a, inc)

This *Achieve* goal suggests instantiations

R: Mobilized (a, inc)
S: Intervention (a, inc)

Negating the goal yields a high-level obstacle:

Obstacle MobilizedAmbulanceNotInTimeAtDestination
FormalDef $\Diamond\ \exists$ a: Ambulance, inc: Incident
Mobilized (a, inc) $\wedge$ TimeDist (a.Loc, inc.Loc) $\leq$ 11
$\wedge \Box_{\leq 11\mathrm{m}} \neg$ Intervention (a, inc)

The nonpersistence obstruction patterns in Table 1 suggest looking for domain properties taking the form

R $\wedge \Diamond$ S $\Rightarrow P\mathcal{W}$ (P $\wedge$ S)

The latter involve a persistent condition *P* that must continuously hold, from the time of *R* to the time of *S*, for *R* to lead to *S*. Given the instantiations for *R* and *S*, two candidates *P* are suggested from *R* to satisfy the above persistence condition:

P1: Mobilized (a, inc)
P2: TimeDist (a.Loc, inc.Loc) < TimeDist ($\underline{\text{a.Loc}}$, inc.Loc)

(The underline notation is used to denote the previous state.) These candidates produce two persistence conditions that are domain properties indeed: The former says that if a sufficiently close ambulance is mobilized and intervenes at the location within 11 minutes, then it 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 nonpersistence pattern in Table 1 to generate the two following obstacles (one for each persistent condition):

Obstacle AmbulanceMobilizationRetracted
FormalDef $\Diamond\ \exists$a: Ambulance, inc: Incident
Mobilized (a, inc) $\wedge$ TimeDist (a.Loc, inc.Loc) $\leq$ 11
$\wedge$ ($\neg$ Intervention (a, inc) $\mathcal{U}_{\leq 11\mathrm{m}} \neg$ Mobilized (a, inc))

and

Obstacle
MobilizedAmbulanceStoppedOrInWrongDirection
FormalDef $\Diamond\ \exists$ a: Ambulance, inc: Incident
Mobilized (a, inc) $\wedge$ TimeDist (a.Loc, inc.loc) $\leq$ 11
$\wedge$ ($\neg$ Intervention(a, inc) $\mathcal{U}_{\leq 11\mathrm{m}}$
TimeDist (a.Loc, inc.Loc) $\geq$ TimeDist ($\underline{\text{a.Loc}}$, inc.Loc))

(In the above assertions, $P\mathcal{U}_{\leq \mathrm{d}}Q$ stands for $P\mathcal{U}Q \wedge \Diamond_{\leq \mathrm{d}}Q$.)

Further refinement of these formal obstacles based on regression, patterns, and heuristics from Section 5 yield the following obstacle OR-refinement tree:

```
MobilizedAmbulanceNotInTimeAtDestination
   ← AmbulanceMobilization Retracted
      ← MobilizedAmbulanceDestinationChanged
         ← LocationConfusedByCrew
      ← MobilizedAmbulanceDestinationForgotten
      ← AmbulanceMobilizationCancelled
   ← MobilizedAmbulanceStoppedOrInWrongDirection
      ← AmbulanceStopped
         ← AmbulanceBreakdownOrAccident
         ← AmbulanceStoppedInTraffic
      ← AmbulanceInWrongDirection
         ← AmbulanceLost
            ← CrewInUnfamiliarTerritorry
         ← TrafficDeviation
```

For the terminal goal *AmbulanceMobilizedFromPrintedMob Order* appearing as subgoal of the root goal *AmbulanceMobilized* in Fig. 10, 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 [51].

We have compared the set of obstacles generated systematically using our techniques, for the goal structure in Figs. 10, 11, and 12, with the scenarios that actually occurred during the two major system failures in October-November 1992, as reported in [51]. While our obstacles cover the various problems that occured during those failures, they also cover many other potential problems that could (but did not) occur. Tables 8 and 9 summarize the obstacles generated for the various terminal goals in Fig. 10. The tables provide, for each requirement/assumption, the responsible agent assigned to it, the (sub)obstacles derived, and features of the satisfying scenarios that occurred during the reported system failures. 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*) unfeasible.

7.2 Obstacle Resolution

We now discuss various resolution strategies from Section 6 for some of the obstacles generated.

Let us first consider the obstacle MobOrderTakenByOtherAmbulance seen at the end of Section 7.1 to obstruct the goal *AmbulanceMobilizedFromPrintedMobOrder*.

The *obstacle mitigation* strategy 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,

MobilizationByOtherAmbulanceSignaledToRadioOperator,

assigned to AmbulanceCrew, and

MobilizationStatusUpdated,

assigned to RadioOperator. (An alternative refinement and assignment would consist in letting the change be signalled to the MDT instead).

The *obstacle prevention* strategy 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 alternatives have of course to be evaluated by the stakeholders involved.

As suggested in Section 6.1.5, the *domain transformation* strategy to resolve the same obstacle would result here in transforming the MobOrder object so that it does not mention the incident location anymore; the latter information would only be given by the MDT inside the ambulance. (Such resolution would however be fairly risky if MDT's are likely to break down.)

The *goal substitution* strategy 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 MobOrder objects no longer have an attribute indicating the target ambulance).

Finally, the *obstacle reduction* strategy might consist here in trying to change ambulance crew practice by a reward/dissuasion system.

Another interesting example of obstacle reduction concerns the obstacle CrewInUnfamiliarTerritorry refining AmbulanceLost (see Section 7.1). The obstacle reduction consists in dividing the city into geographic divisions and allocating ambulances to incidents within the same division. This policy was found in the original system, abandoned in the "Pan London" system that failed and restored in the newly designed system. This corresponds to goal substitution as well; the goal DivisionalAmbulanceAllocated is chosen as an alternative to the goal (PanLondon)AmbulanceAllocated in Fig. 9.

We now illustrate the *goal restoration* strategy. Consider the obstacle

MDTMobOrderIgnored

that appears at the bottom of Table 8 in Appendix 2. A low-level restoration goal would be to generate an audible signal to make crews aware of the mobilization order. An alternative, higher-level resolution would consist in introducing a higher-level restoration goal

FailedMobilizationRecovered

to resolve the higher-level obstacle

AllocatedAmbulanceNotMobilized

TABLE 8
Obstacles to Subgoals of the Goal *Achieve[AllocatedAmbulanceMobilized]*

Agent	Goal	Obstacle	Oct/Nov'92 scenario
CAD	MobOrderSentTo KnownStation	MobOrderNotSent	*no PSTN line free*
		MobOrderSentToWrongStation	
		MobOrderSentToWrongAmbulance	
		MobOrderSentWith- WrongDestination	
		InvalidMobOrderSent	
	AmbAllocationSentTo MDT	AmbAllocationNotSentToMDT	
		AmbAllocationSentToWrongMDT	
		AmbAllocatioSentWith- WrongDestination	
		InvalidAmbAllocationSentToMDT	
Communic.- Infrastructure	MobOrderTransmitted ToStation	MobOrderNotTransmitted	*radio congestion, radio blackspot*
		MobOrderDeliveredAtWrongStation	
		MobOrderCorruptedDuring- Transmission $\longrightarrow$ WrongDestination $\longrightarrow$ WrongAmbulance $\longrightarrow$ InvalidMobOrder	
	AmbAllocation TransmittedToMDT	AmbAllocationNotTransittedToMDT	
		AmbAllocationTransmittedAt- WrongMDT	
		AmbAllocationCorruptedDuring- Transmission $\longrightarrow$ WrongDestination $\longrightarrow$ OtherValidMsgDelivered $\longrightarrow$ InvalidMsgDelivered	
Station Printer	ReceivedMobOrder PrintedAtStation	ReceivedMobOrderNotPrinted	
		PrintedMobOrderUnreadable	
MDT	ReceivedAllocation DisplayedOnMDT	ReceivedAllocationNotDisplayed- OnMDT	
		IncorrectDestinationDisplayed	
Ambulance- Crew	AmbulaceMobilized FromPrintedMobOrder	AmbNotMobilizedFromMobOrder- AtStation $\longleftarrow$ MobOrderIgnored $\longleftarrow$ AmbNotAt Station $\longleftarrow$ AmbNotAvailable	
		MobOrderTakenByOtherAmbulance $\longleftarrow$ MobOrderConfuision $\longleftarrow$ AllocatedAmbNotAvailable $\longleftarrow$ AllocatedAmbNotAtStation $\longleftarrow$ established work practice	*crews take different vehicle from those allocated by CAD*
		LocationConfusedByCrew	
	AmbulanceMobilized FromMobOrderOnMDT	AmbulanceNotMobilizedFrom- MobOrderonMDT $\longleftarrow$ MDTMobOrderIgnored $\longleftarrow$ CrewNotInAmbulance $\longleftarrow$ AmbulanceNotAvailable	
		AmbulanceMobilizedWithDifferent- DestinationThanMDTDestination $\longleftarrow$ LocationConfusedByCrew $\longleftarrow$ OtherMobilizationDestination- Pending $\longleftarrow$ MDTDestDifferentFrom- DestOnMobOrderAtStation	

TABLE 9
Obstacles to Subgoals of the Goal *Achieve[AmbulanceMobilizationKnown]*

Agent	Goal	Obstacle	Oct/Nov'92 scenario
Ambulance-Crew	Avoid Unallocated Ambulance Mobilized	MobOrderTakenBy-OtherAmbulance	*crews use different vehicle*
	Allocated AmbMobilization AcknowledgedOnMDT	AmbulanceCrewForgetTo-AcknowledgeMobilization	*crews don't press status buttons*
		AmbCrewPushWrongButton-ToAcknowledgeMobilization	*crews press buttons in wrong order*
MDT	MDTMobAckSent	MDTMobAckNotSent	
		MDTSendsOtherMsg	
		SentMobAckErroneous	
Communic.-Infrastructure	MDTMobAckDelivered	MDTMobAckNotDelivered	*communication bottelneck, radio blackspot*
		MDTMobAckCorrupted	
CAD	ReceivedMobAckRecorded	ReceivedMobAckIgnored	*failure of system to catch all data*
		ReceivedMobAckConfusedWith-OtherMsg	
		ReceivedMobAckRecordedFor-WrongAmbulance	

This goal would restore the higher-level goal AllocatedAmbulanceMobilized through the following goal refinement tree:

FailedMobilizationRecovered
$\leftarrow$ AmbulanceMobilizationKnown
$\leftarrow$ ...
$\leftarrow$ UnrespondedAllocationRestored
$\leftarrow$ UnrespondedAllocationSignaled
$\leftarrow$ SignaledUnrespondedAllocReallocated

For the CrewPushWrongButton subobstacle in Table 9, a restoration goal under responsibility of MDT might be to signal an error if the pushed button is not the one expected.

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 nonpersistence pattern from Table 1:

$\Diamond\, \exists$ a: Ambulance, inc: Incident
Mobilized (a, inc)
$\wedge$ ($\neg$ Intervention (a, inc) $\mathcal{U}$ Breakdown (a))

Using the third deidealization pattern in Table 7, we obtain the weakened version for that goal:

$\forall$ a: Ambulance, inc: Incident
Mobilized (a, inc)
$\wedge$ ($\neg$ Breakdown(a) $\mathcal{W}$Intervention (a, inc))
$\Rightarrow \Diamond$ Intervention (a, inc)

The propagation will result in strengthened companion goals like

$\forall$ inc: Incident, p: Person
Reported (inc, p) $\Rightarrow$
$\Diamond\, \exists$ a: Ambulance,
Mobilized (a, inc)
$\wedge$ [$\neg$ Breakdown(a) $\mathcal{W}$Intervention (a, inc)]

to be refined and deidealized in turn.

7.3 Discussion

Many of the technical problems with the LAS were caused by incomplete identification and resolution of obstacles. These problems have to be identified and resolved at requirements engineering time, not at programming time, when it is too late. The techniques presented in this paper provide formal and heuristic support for generating high-level exceptions and their resolutions in a systematic way. Requirements engineers can then concentrate their efforts on assessing with stakeholders which resolution is the most appropriate for their domain.

Regression and formal patterns were seen to help identifying not only obstacles, but also the companion domain properties that are necessary to derive them.

Our experience in using these techniques for the LAS and other systems revealed a number of issues that are worth pointing out.

- 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 paper 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 EncodedMDTAvailability-NotTransmitted.

  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 [64], [49] may therefore be required.

8 Related Work

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 [14], [29]—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 [14], 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 the one required by its specification [2], [29]. 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 [40], or to precisely define the acceptable level of system behavior degradation when faults occur, if the former objective is not realizable [13].

The notion of ideal fault-tolerant component provides a basis for structuring software systems [2], [73]. 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 [6], [29]. What is meant for a program to tolerate a certain class of fault is formally defined in [6]. 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 [7]. The method is based on the principle of adding detector and corrector components to intolerant programs in a stepwise and noninterfering manner. Various forms of fault-tolerance are discussed in [29]; 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, [9] describes language mechanisms for handling violations of assumptions in a database. Using such mechanisms, programs can be designed to detect and handle exeptional 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 [55]. 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 [26].) 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 [52], [5]. 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 [8] for a comparison of this work with ours.)

Some work has also been done at specification level. The JSD method [35] 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. Our techniques for generating and resolving obstacles at the goal level are intended to fill that void.

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 [72].

Throughout this paper, 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 [32], [33] are aimed at checking whether the set of conditions guarding transitions in a state machine covers all possible cases.

Model checking techniques generate counterexamples showing that a temporal logic specification is violated by a finite state machine specification [34], [60]. In the same vein, planning techniques can be used to exhibit scenarios showing the inconsistency between an abstract property and an operational model [4], [27], [31]. 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 the one hand, we want to conduct the analysis at the goal level for reasons explained throughout the paper; model checking requires the availability of an operational description of the target system, or of relational specifications [38] 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 [38] 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 [53] 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 [74], [63]. Several other techniques have also been proposed to generate other standard hazard analysis models from RSML specifications [74], [63]. 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 more abstract 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. They allow 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 [54], in that they embed experience about known forms of obstruction. General criteria correponding to such checklists have been identified in [39]. These criteria cover exceptional circumstances such as unexpected inputs, computer errors, environmental disturbances, etc. Good RE practices also consider checklists that cover unexpected inputs, operator errors, and other faults or exceptional circunstances [83]. Our heuristics are in fact closer to HAZOP-like guidewords that can be used to elicit hazards [54]; 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 [75].

Our work builds on Potts' paper which was the first to introduce the notion of obstacle as a dual notion to goals [71]. 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 paper, see Section 5.4). Obstacle resolution is not studied there.

Sutcliffe et al. also build on Potts' work by proposing additional heuristics for identifying possible exceptions and errors in such interaction scenarios—e.g., scenarios in which events occur in the wrong order, or in which incorrect information is transmitted [84]. Influencing factors such as agent motivation and workload are also used there 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 [84] 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 centered around the concept of scenario. It provides no formal guidance compared with the range of obstacle generation/resolution techniques that can be precisely defined through rigorous reasoning on declarative specifications of goals.

Deontic logics are formalisms that allow one to specify and reason about normal and abnormal situations by means of modal operators such as permission and obligation [62]. Such logics have been proposed for system specification, allowing one to specify what should happen if an abnormal situation occurs [56], [44]. However, such approaches do not provide any guidance for elaborating the requirements, in particular the requirements dealing with the abnormal situations. In contrast, our approach for resolving obstacles is based on goals which serve as a rationale for introducing new requirements to deal with the abnormal situations.

The principle of pattern-directed specification and reasoning, as we applied it in [17] for formal goal refinement and in this paper for obstacle refinement, has gained recent interest in the formal analysis community. For example, Dwyer et al. discuss their experience in building and reusing a rich library of temporal patterns that codify in high-level terms property specifications to be input to analysis tools such as model checkers [20].

Our initial ideas were presented in [48] which this paper significantly expands on—notably, by a full treatment of obstacle completeness and AND/OR refinement, a much more extensive set of patterns, many more heuristics, more resolution strategies, and the application to a real safety-critical system. We are also investigating an alternative, dynamic approach in which system deviations from requirements/assumptions are monitored and reconciled at runtime [26].

9 Conclusion

In order to get high-quality software, it is of utmost importance to reason about agent behavior during requirements elaboration—not only software agents, but also the agents in the environment like devices, operators, users, etc. This point has been recently reempahsized in the context of requirements engineering [81].

The key principle underlying this paper 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.

Another important message is our preference for a constructive approach to requirements elaboration, over a posteriori analysis of possibly poor requirements. It is better to construct hopefully complete, realistic, and achievable requirements than to correct poor ones. In the process discussed in this paper, goal-oriented elaboration of requirements and systematic obstacle analysis proceed hand-in-hand.

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 strategies to eliminate, reduce, or tolerate the obstacle. Domain knowledge was seen to play an important role in some of these techniques; however, as we pointed out, such knowledge may be elicited gradually during obstacle analysis.

The techniques were applied to a significant safety-critical system for which failures have been reported; this provided some basis for assessing them and raising important questions and open issues. Our techniques also allowed us to formally generate the 17 obstacles informally identified in [71] for the meeting scheduler benchmark [25], plus a dozen more. The space of resolutions was even broader. Within a potentially large space of obstacles and resolutions, the requirements engineer has to decide which ones are meaningful to the system considered and need to receive careful attention.

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.

We hope to have convinced the reader through the variety of examples given that the techniques proposed are general, systematic, and effective in generating and resolving subtle obstacles. Our plan is to integrate these techniques in the KAOS/GRAIL environment [18] in the near future so that large-scale experimentation on industrial projects from our tech transfer institute can take place.

Acknowledgments

The work reported herein was partially supported by the "Communauté Française de Belgique" (FRISCO project, Actions de Recherche Concertées Nr. 95/00-187 - Direction générale de la Recherche). The authors would like to thank Colin Potts for his insightful discussions and suggestions that originally triggered this work and Alessandra Russo for pointing out an error in a regression for the London Ambulance System. The authors are also grateful to the reviewers for helpful comments and feedback.

References

[1] E.J. Amoroso, *Fundamentals of Computer Security.* Prentice Hall, 1994.

[2] T. Anderson and P.A. Lee, *Fault Tolerance: Principles and Practice.* Prentice Hall, 1981.

[3] A.I. Anton, W.M. McCracken, and C. Potts, "Goal Decomposition and Scenario Analysis in Business Process Reengineering," *Proc. Conf. Advanced Information Systems Eng. (CAISE '94)*, pp. 94–104, 1994.

[4] J.S. Anderson and S. Fickas, "A Proposed Perspective Shift: Viewing Specification Design as a Planning Problem," *Proc. Fifth Int'l Workshop Software Specification and Design*, pp. 177-184, 1989.

[5] T. Anderson, R. de Lemos, and A. Saeed, "Analysis of Safety Requirements for Process Control Systems," *Predictably Dependable Computing Systems,* B. Randell, J.C. Laprie, B. Littlewood, and H. Kopetz, eds., Springer-Verlag, 1995.

[6] A. Arora and M.G. Gouda, "Closure and Convergence: A Foundation of Fault-Tolerant Computing," *IEEE Trans. Software Eng.*, vol. 19, no. 11, pp. 1,015–1,027, 1993.

[7] A. Arora and S. Kulkarni, "Component-Based Design of Multitolerant Systems," *IEEE Trans. Software Eng.*, vol. 24, no. 1, pp. 63–78, Jan. 1998.

[8] D.M. Berry, "The Safety Requirements Engineering Dilemma," *Proc. Ninth Int'l Workshop Software Specification and Design,* Apr. 1998.

[9] A. Borgida, "Language Features for Flexible Handling of Exceptions in Information Systems," *ACM Trans. Database Systems,* vol. 10, no. 4, pp. 565–603, Dec. 1985.

[10] *Readings in Knowledge Representation,* R.J. Brachman and H.J. Levesque eds., Morgan Kaufmann, 1985.

[11] R.W. Butler, S.P. Miller, J.N. Potts, and V.A. Carreno, "A Formal Methods Approach to the Analysis of Mode Confusion," *Proc. 17th Digital Avionics Systems Conference,* Nov. 1998, http://shemesh.larc.nasa.gov/fm/fm-now-mode-confusion.html.

[12] E.C. Coffman, M.J. Elphick, and J. Shoshani, "System Deadlocks," *ACM Computing Surveys*, vol. 3, no. 2, pp. 67–68, June 1971.

[13] F. Cristian, "Understanding Fault-Tolerant Distributed Systems," *Comm. ACM*, Feb. 1991.

[14] F. Cristian, "Exception Handling," *Software Fault Tolerance,* M.R. Lyu ed., 1995.

[15] A. Dardenne, S. Fickas, and A. van Lamsweerde, "Goal-Directed Concept Acquisition in Requirements Elicitation," *Proc. Sixth Int'l Workshop Software Specification and Design*, pp. 14–21, 1991.

[16] A. Dardenne, A. van Lamsweerde, and S. Fickas, "Goal-Directed Requirements Acquisition," *Science of Computer Programming,* vol. 20, pp. 3–50, 1993.

[17] R. Darimont and A. van Lamsweerde, "Formal Refinement Patterns for Goal-Driven Requirements Elaboration," *Proc. Fourth ACM SIGSOFT Symp. Foundations of Software Eng.*, pp. 179-190, Oct. 1996.

[18] R. Darimont, E. Delor, P. Massonet, and A. van Lamsweerde, "GRAIL/KAOS: An Environment for Goal-Driven Requirements Engineering," *Proc. 20th Int'l Conf. Software Eng. (ICSE '98)*, vol. 2, pp. 58–62, Apr. 1998.

[19] E.W. Dijkstra, "Hierarchical Ordering of Sequential Processes," *Acta Informatica*, vol. 1, pp. 115–138, 1971.

[20] M.B. Dwyer, G.S. Avrunin, and J.C. Corbett, "Patterns in Property Specifications for Finite-State Verification," *Proc. 21st Int'l Conf. Software Eng. (ICSE '99),* May 1999.

[21] S. Easterbrook, "Resolving Requirements Conflicts with Computer-Supported Negotiation," M. Jirotka and J. Goguen eds., pp. 41–65, Academic Press, 1994.

[22] M. Feather, "Language Support for the Specification and Development of Composite Systems," *ACM Trans. on Programming Languages and Systems*, vol. 9, no. 2, pp. 198–234, Apr. 1987.

[23] M. Feather, "Cardinality Evolution in Specifications," *Proc. Eighth Conf. Knowledge-Based Software Eng. (KBSE '93)*, Sept. 1993.

[24] M. Feather, "Towards a Derivational Style of Distributed System Design," *Automated Software Eng.*, vol. 1, no. 1, pp. 31-60, 1995.

[25] M. Feather, S. Fickas, A. Finkelstein, and A. van Lamsweerde, "Requirements and Specification Exemplars," *Automated Software Eng.*, vol. 4, no. 4, Oct. 1997.

[26] M. Feather, S. Fickas, A. van Lamsweerde, and C. Ponsard, "Reconciling System Requirements and Runtime Behavior," *Proc. Ninth Int'l Workshop Software Specification and DesignI (WSSD '98),* Apr. 1998.

[27] S. Fickas and R. Helm, "Knowledge Representation and Reasoning in the Design of Composite Systems," *IEEE Trans. Software Eng.*, 470–482, June 1992.

[28] A. Finkelstein, "The London Ambulance System Case Study," *Succ. Eighth Int'l Workshop Software Specification and Design (IWSSD 8),* Sept. 1996.

[29] F.C. Gartner, "Fundamentals of Fault-Tolerant Distributed Computing in Asynchronous Environment," *ACM Computing Surveys,* vol. 31, no. 1, pp. 1–26, Mar. 1999.

[30] D. Gries, *The Science of Programming.* Springer-Verlag, 1981.

[31] R.J. Hall, "Explanation-Based Scenario Generation for Reactive System Models," *Proc. Automated Software Eng. (ASE '98),* Oct. 1998.

[32] M.P. Heimdahl and N.G. Leveson, "Completeness and Consistency in Hierarchical State-Based Requirements," *IEEE Trans. Software Eng.*, vol. 22, no. 6, pp. 363–377, June 1996.

[33] C. Heitmeyer, R. Jeffords, and B. Labaw, "Automated Consistency Checking of Requirements Specificatons," *ACM Trans. Software Eng. and Methodology,* vol. 5, no. 3, pp. 231–261, July 1996.

[34] G. Holtzman, "The Model Checker SPIN," *IEEE Trans. Software Eng.*, vol. 23, no. 5, pp. 279-295, May 1997.

[35] M.A. Jackson, *System Development.* Prentice Hall, 1983.

[36] M. Jackson and P. Zave, "Domain Descriptions," *Proc. First Int'l IEEE Symp. Requirements Eng. (RE '93)*, pp. 56–64, Jan. 1993.

[37] M. Jackson, *Software Requirements & Specifications—A Lexicon of Practice, Principles and Pejudices.* ACM Press, Addison-Wesley, 1995.

[38] D. Jackson and C.A. Damon, "Elements of Style: Analyzing a Software Design Feature with a Counterexample Detector," *Proc. Int'l Symp. Software Testing and Analysis (ISTA '96),* vol. 21, no. 3, pp. 239–249, 1996.

[39] M.S. Jaffe et al., "Software Requirements Analysis for Real-Time Process-Control Systems," *IEEE Trans. Software Eng.*, vol. 17, no. 3, pp. 241–258, Mar. 1991.

[40] P. Jalote, *Fault Tolerance in Distributed Systems.* Prentice Hall, 1994.

[41] *Real-Time Systems: Specification, Verification and Analysis*, M. Joseph ed., Prentice Hall, 1995.

[42] D.O. Keck and P.J. Kuehn, "The Feature and Service Interaction Problem in Telecommunication Systems: A Survey," *IEEE Trans. Software. Eng.*, vol. 24, no. 10, pp. 779–796, Oct. 1998.

[43] S.E. Keller, L.G. Kahn, and R.B. Panara, "Specifying Software Quality Requirements with Metrics," *System and Software Requirements Eng.*, R.H. Thayer and M. Dorfman, eds., pp. 145–163, 1990.

[44] S.J.H. Kent, T.S.E. Maibaum, and W.J. Quirk, "Formally Specifying Temporal Constraints and Error Recovery," *Proc. First Int'l Symp. Requirements Eng. (RE '93)*, pp. 208–215, Jan. 1996.

[45] R. Koymans, *Specifying Message Passing and Time-Critical Systems with Temporal Logic.* Springer-Verlag, 1992.

[46] A. van Lamsweerde, "Learning Machine Learning," *Introducing a Logic Based Approach to Artificial Intelligence*, A. Thayse ed., vol. 3, pp. 263–356, 1991.

[47] A. van Lamsweerde, R. Darimont, and P. Massonet, "Goal-Directed Elaboration of Requirements for a Meeting Scheduler: Problems and Lessons Learned," *Proc. Second Int'l Symp. Requirements Eng. (RE '95)*, 1995.

[48] A. van Lamsweerde and E. Letier, "Integrating Obstacles in Goal-Driven Requirements Eng.," *Proc. 20th Int'l Conf. Software Eng. (ICSE '98)*, vol. 1, pp. 53–63, Apr. 1998.

[49] A. van Lamsweerde, R. Darimont, and E. Letier, "Managing Conflicts in Goal-Driven Requirements Engineering," *IEEE Trans. Software Eng.,* vol. 24, no. 11, pp. 908–926, Nov. 1998.

[50] A. van Lamsweerde and L. Willemet, "Inferring Declarative Requirements Specifications from Operational Scenarios," *IEEE Trans. Software. Eng.*, vol. 24, no. 12, pp. 1,089–1,114, Dec. 1998.

[51] "Inquiry Into the London Ambulance Service,"Technical Report, ISBN 0-905133-70-6, The Communications Directorate, South West Thames Regional Authority, Feb. 1993, http://hsn.londamb.sthames.nhs.uk/http.dir/service/organisation/featurs/info.html.

[52] R. de Lemos, B. Fields, and A. Saeed, "Analysis of Safety Requirements in the Context of System Faults and Human Errors," *Proc. IEEE Int'l Symp. and Workshop Systems Eng. of Computer Based Systems,* pp. 374–381, Mar. 1995.

[53] N.G. Leveson and J.L. Stolzy, "Safety Analysis using Petri Nets," *IEEE Trans. Software Eng.*, vol. 13, no. 3, pp. 386–397, Mar. 1987.

[54] N. Leveson, *Safeware—System Safety and Computers.* Addison-Wesley, 1995.

[55] R. Lutz, "Analyzing Software Requirements Errors in Safety-Critical, Embedded Systems," *Proc. First Int. Symp. Requirements Eng. (RE '93),* pp. 126–133, Jan. 1996.

[56] T. Maibaum, "Temporal Reasoning over Deontic Specifications," *Deontic Logic in Computer Science—Normative System Specification,* J. Ch. Meyer and R.J. Wieringa eds., Wiley, 1993.

[57] Z. Manna and A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems,* Springer-Verlag, 1992.

[58] Z. Manna and the STep Group, "STeP: Deductive-Algorithmic Verification of Reactive and Real-Time Systems," *Proc. Eighth Int'l Conf. Computer-Aided Verification (CAV '96),* pp. 415–418, July 1996.

[59] P. Massonet and A. van Lamsweerde, "Analogical Reuse of Requirements Frameworks," *Proc. Third Int'l Symp. Requirements Eng. (RE '97),* pp. 26–37, 1997.

[60] K.L. McMillan, *Symbolic Model Checking: An Approach to the State Explosion Problem.* Kluwer, 1993.

[61] B. Meyer, "On Formalism in Specifications," *IEEE Software,* vol. 2, no. 1, pp. 6–26, Jan. 1985.

[62] *Deontic Logic in Computer Science—Normative System Specification,* J.Ch. Meyer and R.J. Wieringa eds.,Wiley, 1993.

[63] F. Modugno, N.G. Leveson, J.D. Reese, K. Partridge, and S.D. Sandys, "Integrated Safety Analysis of Requirements Specifications," *Proc. Third Int'l Symp. Requirements Eng. (RE '97),* 1997.

[64] J. Mylopoulos, L. Chung, and B. Nixon, "Representing and Using Non-Functional Requirements: A Process-Oriented Approach," *IEEE Trans. Software. Eng.*, vol. 18, no. 6, pp. 483–497, June 1992.

[65] J. Mylopoulos, L. Chung, and E. Yu, "From Object-Oriented to Goal-Oriented Requirements Analysis," *Comm. ACM,* vol. 42, no. 1, pp. 31–37, Jan. 1999.

[66] N.J. Nilsson, *Problem Solving Methods in Artificial Intelligence.* McGraw-Hill, 1971.

[67] B. Nuseibeh, "To Be and Not to Be: On Managing Inconsistency in Software Development," *Proc. Eighth Int'l Workshop Software Specification and Design (IWSSD 8),* pp. 164–169, 1996.

[68] S. Owre, J. Rushby, and N. Shankar, "Formal Verification for Fault-Tolerant Architectures: Prolegomena to the Design of PVS," *IEEE Trans. Software Eng.*, vol. 21, no. 2, pp. 107–125, Feb. 1995.

[69] D.L. Parnas and J. Madey, "Functional Documents for Computer Systems," *Science of Computer Programming,* vol. 25, pp. 41–61, 1995.

[70] D.E. Perry, "The Inscape Environment," *Proc. 11th Int'l Conf. Software Eng. (ICSE 11),* pp. 2–12, 1989.

[71] C. Potts, "Using Schematic Scenarios to Understand User Needs," *Proc. ACM Symp. Designing Interactive Systems: Processes, Practices, and Techniques (DIS '95),* Aug. 1995.

[72] B. Potter, J. Sinclair, and D. Till, *An Intro. to Formal Specification and Z,* second ed., Prentice Hall, 1996.

[73] B. Randel and J. Xu, "The evolution of the recovery block concept," *Software Fault Tolerance,* M.R. Lyu, ed., Wiley, 1995.

[74] V. Ratan, K. Partridge, J.D. Reese, and N.G. Leveson, "Safety Analysis Tools for Requirements Specifications," *Proc. Compass 96,* June 1996.

[75] J.D. Reese and N. Leveson, "Software Deviation Analysis," *Proc. 19th Int'l Conf. Software Eng. (ICSE '97),* pp. 250–260, May 1997.

[76] W.N. Robinson, "Integrating Multiple Specifications Using Domain Goals," *Proc. Fifth Int'l Workshop Software Specification and Design (IWSSD 5),* pp. 219–225, 1989.

[77] W.N. Robinson and S. Volkov, "A Meta-Model for Restructuring Stakeholder Requirements," *Proc. 19th Int'l Conf. Software Eng. (ICSE 19),* pp. 140–149, May 1997.

[78] D.T. Ross and K.E. Schoman, "Structured Analysis for Requirements Definition," *IEEE Trans. Software Eng.*, vol. 3, no. 1, pp. 6–15, 1977.

[79] D.S. Rosenblum, "Towards a Method of Programming with Assertions," *Proc., 14th Int'l Conf. Software Eng. (ICSE 14),* pp. 92–104, 1992.

[80] K.S. Rubin and A. Goldberg, "Object Behavior Analysis," *Comm. ACM,* vol. 35, no. 9, pp. 48–62, Sept. 1992.

[81] K. Ryan and S. Greenspan, "Requirements Engineering Group Report," *Succeedings Eighth Int'l Workshop Software Specification and Design (IWSSD 8),* pp. 22–25, Sept. 1996.

[82] A. Saed, R. de Lemos, and T. Anderson, "Robust Requirements Specifications for Safety-Critical Systems," *Proc. 12th Int'l Conf. Safety, Reliability, and Security (SAFECOMP '93),* 1993.

[83] I. Sommerville and P. Sawyer, *Requirements Engineering: A Good Practice Guide.* Wiley, 1997.

[84] A.G. Sutcliffe, N.A. Maiden, S. Minocha, and D. Manuel, "Supporting Scenario-Based Requirements Engineering," *IEEE Trans. Software Eng.*, vol. 24, no. 12, pp. 1,072–1,088, Dec. 1998.

[85] R. Waldinger, "Achieving Several Goals Simultaneously," *Machine Intelligence,* E. Elcock and D. Michie, eds., vol. 8, 1977.

[86] K. Yue, "What Does It Mean to Say That a Specification is Complete?," *Proc. Fourth Int'l Workshop Software Specification and Design (IWSSD 4),* 1987.

[87] P. Zave, "Classification of Research Efforts in Requirements Engineering," *ACM Computing Surveys,* vol. 29, no. 4, pp. 315–321, 1997.

[88] P. Zave and M. Jackson, "Four Dark Corners of Requirements Engineering," *ACM Trans. Software Eng. and Methodology,* pp. 1–30, 1997.

Axel van Lamsweerde is a professor of computing science at the Université Catholique de Louvain, Belgium. He received the MS degree in mathematics from the same university and the PhD degree in computing science from the University of Brussels. From 1970 to 1980, he was research associate with the Philips Research Laboratory in Brussels where he worked on proof methods for parallel programs and knowledge-based approaches to automatic programming. Then he became a professor of software engineering at the Universities of Namur and, Brussels, until he joined UCL in 1990. He is co-founder of the CEDITI technology transfer institute partially funded by the European Union. He has also been a visitor at the University of Oregon and at the Computer Science Laboratory of SRI International, Menlo Park.

Dr. van Lamsweerde's professional interests are in lightweight formal methods and tools for assisting software engineers in knowledge-intensive tasks. His current focus is on constructive, technical approaches to requirements engineering and, more generally, on formal reasoning about software engineering products and processes. His recent papers can be found at http://www.info.ucl.ac.be/people/avl.html.

Dr. van Lamsweerde is an ACM fellow and a member of the IEEE. He was program chair of the Third European Software Engineering Conference (ESEC '91), program co-chair of the Seventh IEEE Workshop on Software Specification and Design (IWSSD 7), and program co-chair of the ACM-IEEE Sixteenth International Conference on Software Engineering (ICSE 16). He is a member of the editorial boards of the *Automated Software Engineering Journal* and the *Requirements Engineering Journal*. Since 1995, he also has been editor-in-chief of the *ACM Transactions on Software Engineering and Methodology* (TOSEM).

Emmanuel Letier received the engineering degree in applied mathematics from the Université Catholique de Louvain, Belgium. He is currently involved in PhD research at the Département d'Ingénierie Informatique of the same university. His interests are in techniques for formal reasoning about agents during requirements elaboration.