Integrating Obstacles in Goal-Driven Requirements Engineering

Axel van Lamsweerde and Emmanuel Letier

Département d'Ingénierie Informatique
Université catholique de Louvain
B-1348 Louvain-la-Neuve (Belgium)
{avl, eletier}@info.ucl.ac.be

ABSTRACT

Requirements engineering is concerned with the elicitation of high-level goals to be achieved by the system envisioned, the refinement of such goals and their operationalization into services and constraints, and the assignment of responsibilities for the resulting requirements to agents such as humans, devices, and software.

Requirements engineering processes may often result in requirements and assumptions about agent behaviour that are too ideal; some of them are likely to be violated from time to time in the running system due to unexpected agent behaviour. 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 inevitably result in poor performance, sometimes with critical consequences on the environment.

This paper proposes systematic techniques for reasoning about obstacles to the satisfaction of goals, requirements, and assumptions elaborated in the requirements engineering process. These techniques are integrated into an existing method for goal-driven requirements elaboration with the aim of deriving more complete and realistic requirements.

The concept of obstacle is first defined precisely. Formal techniques and domain-independent heuristics are then proposed for identifying obstacles from goal/assumption formulations and domain properties. The paper then discusses techniques for resolving obstacles by transformation of the goals, requirements and assumptions elaborated so far in the process, or by introduction of new ones.

Numerous examples are given throughout the paper to suggest how the techniques can be usefully applied in practice.

KEYWORDS

Goal-driven requirements engineering, obstacle-driven requirements transformation, defensive requirements specification, specification refinement, lightweight formal methods.

Proceedings ICSE'98 - 20th International Conference on Software Engineering, Kyoto, ACM-IEEE, April 1998.

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 [31], stresses the leading part played by goals during requirements elaboration. There are multiple reasons why goals must be made explicit in the requirements engineering process. Goals drive the elaboration of requirements to support them; they provide a completeness criterion for the requirements specification --the specification is complete if all stated goals are met by the specification [30]; goals provide an explanation to clients of the rationale underpinning requirements; they represent the roots for detecting conflicts among requirements and for resolving them eventually [26]; they generally represent the most stable information in the requirements product. In short, requirements "implement" goals much the same way as programs implement design specifications.

Goals are to be achieved by the various agents operating altogether in the *composite* system; such agents include software components that exist or are to be developed, external devices, and humans in the environment [8, 10]. The elicitation of 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 [4]. 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 [21]. When goals can be formalized, *formal reasoning* techniques are expected to do more. For example, 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 [2, 10]. Formal goal models may also be used for formal verification of correctness of goal refinements/operationalizations or, more constructively, for formal derivation of such refinements/operationalizations [4, 9, 5].

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 external agents like humans or devices [16, 23]. This problem is not really handled by current requirements elaboration methods.

Consider a system to support the electronic reviewing process for a scientific journal, for example; a first-sketch goal such as Achieve[ReviewReturnedInFourWeeks] or an assumption such as ReviewerReliable are straightforward examples of overideal statements that are likely to be violated on occasion; the same might be true for a security goal such as Maintain[ReviewerAnonymity]. In a resource management system, a goal such as Achieve[RequestedResourceUsed] or an assumption such as RequestPendingUntilUse are also overideal as requesting agents may change their mind and no longer wish to use the requested resource even if the latter becomes available. In a meeting scheduler system, a goal such as Achieve[ParticipantsTimeConstraintsProvided] is likely to be violated, e.g., for participants that do not check their email regularly thereby missing invitations to meetings and requests for providing their time constraints. In a control system, a goal such as Maintain[AlarmIssuedWhenAbnormalCondition] might be violated sometimes due to unavailable data, device failure or deactivation by malicious agents.

Overidealization of goals, requirements and assumptions results in run-time inconsistencies between the specification of the system and its actual behaviour. The lack of anticipation of exceptional circumstances in the environment thus leads to unrealistic, unachievable and/or incomplete requirements. As a consequence, the software developed from those requirements will inevitably result in poor performance, 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 into goal-driven requirements elaboration methods in order to derive more complete and realistic requirements.

In [23], the concept of *obstacle* is proposed as a dual notion to goals. While goals capture desired properties, obstacles capture undesired ones. (In the particular case of safety goals, obstacles amount to hazards [17].) Scenarios of interaction between software and human agents are suggested in [23] as a means for identifying potential obstacles that may block the fulfillment of specified goals.

This paper elaborates on the notion of obstacle and proposes a set of heuristics and formal techniques for (i) systematic identification of obstacles from goal specifications and domain properties, and (ii) transformation of goals, requirements and assumptions to overcome or mitigate the obstacles identified.

Back to the example of the ideal goal named Achieve[ReviewReturnedInFourWeeks], our aim is to derive obstacle specifications from a precise specification of this goal and from properties of the domain; one would thereby expect to obtain obstacles such as, e.g., WrongBeliefAboutDeadline or ReviewRequestLost (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; or by changing agent assignments so that the obstacle may no longer occur.

A key principle here is to tackle risks of unexpected agent behaviour at specification time and *at goal level.* Standard exception handling techniques are introduced at later stages of the software lifecycle, such as operational specification, architecturing or implementation (e.g., [22, 27]), where the boundary between the software and its environment has been decided and remains frozen. In contrast, we perform systematic obstacle analysis at a much earlier stage, from goal formulations, so that more freedom is left on adequate ways to handle goal obstacles --like, e.g., alternative agent assignments resulting in different system proposals, in which more or less functionality is automated and in which the interaction between the software and its environment may be quite different.

The integration of obstacle analysis into the requirements engineering process is detailed in the paper in the context of the KAOS methodology for goal-driven requirements elaboration [4, 16, 5]. Section 2 provides some background material on KAOS that will be used in the sequel. Section 3 introduces obstacles, provides a precise characterization of this concept, and presents a modified goal-driven requirements elaboration process that integrates obstacle analysis. Section 4 presents techniques for obstacle identification from goal formulations. Section 5 then presents techniques for transforming goals, requirements and/or assumptions so as to overcome or mitigate obstacles.

2. GOAL-DRIVEN RE WITH KAOS

The KAOS methodology is aimed at supporting the whole process of requirements elaboration --from the high-level goals to be achieved by the agents in the composite system to the operations, objects and constraints to be assigned to software and environmental agents. Thus WHY, WHO and WHEN questions are addressed in addition to the usual WHAT questions addressed by standard specification techniques.

KAOS provides a multi-paradigm specification language, a goal-directed elaboration method, and meta-level knowledge used for local guidance during method enactment. An environment supporting the KAOS methodology is now available and has been used in various large-scale, industrial projects [6]. Hereafter we introduce some of the features that will be used later in the paper; see [4] and [16] for details.

2.1 The Specification Language

The KAOS language combines semantic nets [3] for the conceptual modelling of goals, assumptions, agents, objects and operations in the system; temporal logic [18, 14] for the specification of goals and objects; and state-based specifications [24] for the specification of operations. Unlike most specification languages, KAOS supports a strict separation of requirements from domain descriptions [12].

2.1.1 The Underlying Ontology

The following types of concepts will be used in the sequel.

- *Object:* an object is a thing of interest in the composite system whose instances may evolve from state to state. It is in general specified in a more specialized way, that is, as an *entity, relationship,* or *event* dependent on whether the object is an autonomous, subordinate, or instantaneous object, respectively. Objects are characterized by attributes and invariant assertions. Inheritance is of course supported.
- *Action:* an action is an input-output relation over objects; action applications define state transitions. Actions are characterized by pre-, post- and trigger conditions.
- *Agent:* an agent is another kind of object which acts as processor for some actions. An agent *performs* an action if it is effectively allocated to it; the agent *HasAccess/HasControl* to/over an object if the states of the object are observable/controllable by it. Agents can be humans, devices, programs, etc.
- *Goal:* a goal is an objective the composite system should meet. *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. Goals often *conflict* with others. Goals *concern* the objects they refer to.
- *Constraint:* a constraint is an implementable goal, that is, a goal that can be formulated in terms of states controllable by some individual agent. Goals must be eventually AND/OR *refined* into constraints. Constraints in turn are AND/OR *operationalized* by actions and objects through strengthenings of their pre-, post-, trigger conditions and invariants, respectively. Alternative ways of assigning responsible agents to a constraint are captured through AND/OR *responsibility* links; the actual assignment of agents to the actions that operationalize the constraint is captured in the corresponding *performance* links.
- *Assumption:* an assumption is a fact taken for granted about agents in the environment. Unlike goals, assumptions need not be refined nor enforced. They often appear as auxiliary assertions needed to prove the correctness of refinements or operationalizations. Assumptions are tentatively true and are likely to change.
- *Scenario:* a scenario is a domain-consistent composition of applications of actions by corresponding agent instances; domain-consistency means that the actions are applied in states satisfying their domain precondition together with the various domain invariants attached to the corresponding objects, with resulting states satisfying their domain postcondition. The composition modes include sequential, alternative, repetitive, and parallel composition.

2.1.2 Language Constructs

Each construct in the KAOS language has a generic two-level structure: an outer semantic net layer for *declaring* a concept, its attributes and its various links to other concepts; an inner formal assertion layer for *formally defining* the concept. The generic structure is instantiated to specific types of links and assertion languages according to the specific type the concept is an instance of.

For example, consider the following goal specification for a meeting scheduler system:

Goal *Achieve* [ParticipantsConstraintsKnown]
  Concerns Meeting, Participant, Scheduler, ...
  RefinedTo ConstraintsRequested, ConstraintsProvided
  InformalDef *A meeting scheduler should know the constraints of the various participants invited to the meeting within C days after appointment*
  FormalDef $\forall$ m: Meeting, p: Participant, s: Scheduler
    Invited (p, m) $\wedge$ Scheduling (s, m)
    $\Rightarrow \Diamond_{\leq Cd}$ Knows (s, p.Constraints)

The declaration part of this specification introduces a goal named ParticipantsConstraintsKnown, refering to objects such as Participant or Scheduler, refined into two subgoals, and defined by some informal statement.

The assertion defining this goal formally is written in a real-time temporal logic borrowed from [14]. In this paper we will use some classical operators for temporal referencing [18]: **o** (in the next state), $\bullet$ (in the previous state), $\Diamond$ (eventually), $\blacklozenge$ (some time in the past), $\square$ (always in the future), $\blacksquare$ (always in the past), $\mathcal{U}$ (always in the future *until*), $\mathcal{W}$ (always in the future *unless*). Real-time restrictions are indicated by subscripts [14]; e.g., $\Diamond_{\leq Cd}$ means "some time in the future within *C* day time units".

In the formal assertion above, the predicate Invited(p,m) means that, in the current state, an instance of the Invited relationship links variables p and m of sort Participant and Meeting, respectively. The Invited relationship, Participant agent and Meeting entity are declared in other sections of the specification, e.g.,

Agent Participant
  CapableOf CommunicateConstraints, ...
  Has Constraints: **Tuple** [ExcludedDates: **SetOf** [*TimeInterval*],
    PreferredDates: **SetOf** [*TimeInterval*]]

Relationship Invited
  Links Participants {**card**: 0:N}, Meeting {**card**: 1:N}
  DomInvar $\forall$p: Participant, m: Meeting
    Invited (p, m) $\Leftrightarrow$ p $\in$ Requesting[-,m].ParticipantsList

In the declarations above, Constraints is declared as an attribute of Participant (this attribute was used in the formal definition of ParticipantsConstraintsKnown).

As mentioned earlier, operations are specified formally by pre- and postconditions, for example,

Action DetermineSchedule
  Input Requesting, Meeting {**Arg**: m}; **Output** Meeting {**Res**: m}
  DomPre Requesting (-,m) $\wedge \neg$ Scheduled (m)
  DomPost Feasible (m) $\Rightarrow$ Scheduled (m)
    $\wedge \neg$ Feasible (m) $\Rightarrow$ DeadEnd (m)

Note that the invariant defining Invited is not a requirement, but a domain description [12]; the pre-/postcondition of DetermineSchedule above are domain descriptions as well. The effective requirements are found in the *constraints* refining the goals, and in the additional pre-/postconditions and invariants that *strengthen* the corresponding domain assertions so as to ensure all constraints specified.

In a KAOS specification, the declaration level is very useful for conceptual modeling (through a concrete graphical syntax), requirements traceability (through semantic net navigation) and specification reuse (through queries). The assertion level is optional and used for formal reasoning.

2.2 The Elaboration Method

The following steps may be followed to systematically elaborate KAOS specifications from high-level goals.

- *Goal elaboration:* elaborate the goal AND/OR structure by defining goals and their refinement/conflict links until implementable constraints are reached; offspring goals are identified by asking HOW questions whereas parent goals are identified by asking WHY questions.
- *Object capture:* identify the objects that are concerned by goals and describe their domain properties.
- *Action capture:* identify object state transitions that are meaningful to the goals, specify them as domain pre- and postconditions of actions, and identify agents that could have those actions among their capabilities.
- *Operationalization:* derive strengthenings on action pre-/postconditions and on object invariants in order to ensure that all constraints are met; requirements that operationalize the goals are obtained thereby.
- *Responsibility assignment:* identify alternative responsibilities for constraints; make decisions among refinement, operationalization, and responsibility alternatives (with process-level objectives such as resolving conflicts, reducing costs, increasing reliability, avoiding overloading agents, etc.); assign the actions to agents that can commit to guaranteeing the requirements in the alternatives selected.

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

2.3 Using Meta-Level Knowledge

At each step of the goal-driven method, domain-independent knowledge can be used for local guidance and validation in the elaboration process.

- A rich taxonomy of goals, objects and actions is provided together with rules to be observed when specifying concepts of the corresponding sub-type. We give a few examples of such taxonomies.
  - Goals are classified by pattern of temporal behavior they require:

    Achieve: $P \Rightarrow \Diamond Q$ or *Cease:* $P \Rightarrow \Diamond \neg Q$

    Maintain: $P \Rightarrow \square Q$ or *Avoid:* $P \Rightarrow \square \neg Q$
  - Goals are also classified by type of requirements they will drive with respect to the agents concerned (e.g., SatisfactionGoals are functional goals concerned with satisfying agent requests; InformationGoals are goals concerned with keeping agents informed about object states; other categories include SafetyGoals, SecurityGoals, AccuracyGoals, etc.).

  Such taxonomies are associated with heuristic rules that may guide the elaboration process, e.g.,
  - SafetyGoals are AvoidGoals to be refined in HardConstraints;
  - ConfidentialityGoals are AvoidGoals on *Knows* predicates.
- Tactics capture heuristics for driving the elaboration or for selecting among alternatives, e.g.,
  - Refine goals so as to reduce the number of agents involved in the achievement of each subgoal;
  - Favor goal refinements that introduce less conflicts.

3. GOAL OBSTRUCTION BY OBSTACLES

While goals capture desired properties, obstacles capture undesired ones. This section defines obstacles and their relationship to goals more precisely; the integration of obstacle-based specification in the goal-driven requirements elaboration method above is then discussed.

3.1 Defining Obstacles

Semantically speaking, a goal defines a set of desired behaviours; a behaviour is defined as a temporal sequence of state transitions controlled by corresponding agent instances. A positive scenario is an instance of this set. Goal refinement yields sufficient subgoals for the goal to be achieved.

Likewise, an obstacle defines a set of undesirable behaviours; a negative scenario is an instance of 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. (The same can be said for assumption obstruction.)

Definition. Let *GA* be a goal or an assumption, and *Dom* be a set of domain descriptions that captures the knowledge available about the domain. An assertion *O* is said to be an *obstructing obstacle* to *GA* iff the following conditions hold:

1. $\{O, Dom\} \vdash \neg GA$ (*obstruction*)
2. $Dom \nvdash \neg O$ (*domain consistency*)
3. there exists a scenario *S* such that $S \models O$ (*feasibility*)

Condition (1) states that the negation of the goal/assumption can be inferred from the logical theory that comprises the obstacle specification and the set of domain properties available; condition (2) states that the negation of the obstacle cannot be logically inferred from the domain theory; condition (3) states that the obstacle specification is satisfiable through one behaviour at least --in other words, one must find at least one scenario of agent cooperation that establishes the obstacle. Clearly, it makes no sense to reason about obstacles that are either inconsistent with the domain or cannot occur through some feasible agent behaviour.

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
Assuming RequestPending
FormalDef $\forall$ bor: Borrower, b: Book
Requesting (bor, b)
$\Rightarrow \Diamond$ ($\exists$ bc: BookCpy) [Copy (bc, b) $\wedge$ Gets (bor, bc)]

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

$\exists$ bor: Borrower, b: Book
$\Diamond$ { Requesting (bor, b)
$\wedge \Box$ ($\forall$ bc: BookCpy) [Copy (bc, b) $\Rightarrow \neg$ Gets (bor, bc)] }

This assertion trivially satisfies condition (1) as it amounts to the negation of the goal; it is consistent in a standard library domain; condition (3) can be satisfied, e.g., through the classical starvation scenario [7] in which, each time a copy of a requested book becomes available, that copy gets borrowed in the next state by a borrower different from the requesting agent.

To illustrate the need for condition (2), consider the following goal for some device control system (expressed in propositional terms for simplicity):

$\Box$ [Running $\wedge$ PressureTooLow $\Rightarrow$ AlarmRaised]

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

$\Box$ [Startup $\wedge$ PressureTooLow $\Rightarrow \neg$ AlarmRaised]
$\wedge \Diamond$ [Running $\wedge$ Startup $\wedge$ PressureTooLow]

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

Given some goal formulation, defensive requirements specification would require as many obstacles as possible to be identified; completeness is desirable --at least for high-priority goals. A *complete* set of obstacles O1, ..., On to a goal/ assumption GA is characterized by the condition

$$\{\neg \text{O1}, ..., \neg \text{On}, \text{Dom}\} \models \text{GA}$$

The problem here is that the space of potential obstacles may sometimes be very large for some goals/assumptions; the deductive closure of the domain theory available may be quite large. Section 4 will present techniques for identifying meaningful obstacles in a systematic way.

3.2 Integrating Obstacles in the RE Process

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 external agents [16, 23]. Our aim is to anticipate exceptional behaviors in the environment in order to derive more complete and realistic requirements.

A defensive extension of the goal-driven process outlined in Section 2.2 is depicted in Fig. 1. The main difference is the obstacle analysis loop introduced in the upper right part. During elaboration of the goal graph by elicitation (asking WHY questions) and by refinement (asking HOW questions), obstacles are derived from goal specifications. Obstacles identified are then resolved which results in a goal structure updated with new goals and/or transformed versions of existing ones. These goals in turn may refer to new objects/actions and require specific operationalizations.

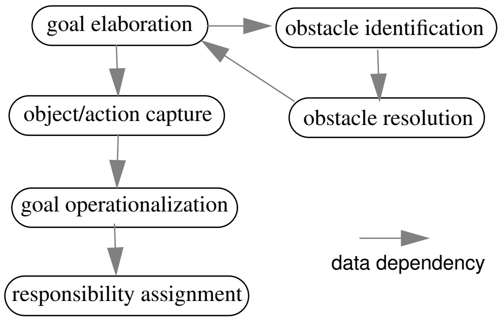

Fig. 1 - Obstacle analysis in goal-driven requirements elaboration

Some questions arising here are: (i) from which goals in the goal graph should obstacles be identified? (ii) for some given goal, how extensive need obstacle identification be? (iii) for some given obstacle, how drastic need obstacle resolution be? We discuss these questions in turn.

The more specific the goal/assumption is, the more specific its obstructing obstacles will be. A high-level goal will produce high-level obstacles which will need to be refined significantly into sub-obstacles in order to identify precise circumstances whose feasibility needs to be assessed through negative scenarios of agent behaviour. 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 in the goal refinement process, that is, from *constraints* assignable to agents and from the assumptions possibly needed to derive them.

The extensiveness of obstacle identification will depend on the type and priority of the goal being obstructed. For example, obstacle identification needs to be fairly extensive for SafetyGoals. Domain-specific cost-benefit analysis needs to be carried out to decide when the obstacle identification process should terminate.

As will be seen in Section 5, various strategies can be followed to avoid or mitigate obstacles. Which way to follow will depend on the severity of the obstacle and of the consequences of its occurrence, and on the likelihood of its occurrence. Forward evaluation of consequences and their impact, together with cost-benefit analysis, need to be deployed in order to provide a definite answer. This important issue will not be considered further in this paper.

4. IDENTIFYING OBSTACLES

According to the definition given in Section 3.1, the identification of obstacles to some given goal/assumption proceeds by iteration of three steps:

(1) Given the goal/assumption specification, find some assertion that may obstruct it;

(2) Check that the candidate obstacle thereby obtained is consistent with the domain theory available;

(3) Determine the satisfiability of this candidate obstacle by

finding out some feasible negative scenario.

Step (2) is a classical consistency checking problem in logic; it will not be considered further in this paper. Step (3) can be done manually [23] or using automated techniques such as [2, 10]. We will therefore concentrate on step (1) in this paper and present techniques for deriving obstacles whose consistency and feasibility need to be subsequently checked.

4.1 Regressing Goal Negations

The first technique is directly based on the first, obstruction condition defining an obstacle (see Section 3.1). Given the goal/assumption assertion GA, it consists of calculating preconditions for obtaining the negation ¬ GA from the domain theory. Every precondition obtained defines an obstacle. This may be achieved by a form of goal regression [29], which is the counterpart of Dijkstra's precondition calculus [11] for declarative representations.

Let us illustrate the idea on a simple example first. 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 ∀ m: Meeting, p: Participant
Intended (p, m) ∧ Informed (p, m) ∧ Convenient (p, m)
⇒ ◊ Participates(p, m)

(Real-time subscripts to temporal operators are dropped in order to make the presentation simpler.)

The initialization step consists of taking the negation of that goal which yields

(NG) ◊ ∃ m: Meeting, p: Participant
Intended (p, m) ∧ Informed (p, m) ∧ Convenient (p, m)
∧ ❑ ¬ Participates(p, m)

(The initialization may produce precise, feasible obstacles in some cases, see other examples below).

Suppose now that the domain theory contains the following property that partially defines the concept of participation:

∀ m: Meeting, p: Participant
Participates(p, m) ⇒ Holds (m) ∧ Convenient (p, m)

or, equivalently,

(D) ∀ m: Meeting, p: Participant
¬ [Holds (m) ∧ Convenient (p, m)] ⇒ ¬ 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) ◊ ∃m: Meeting, p: Participant
Intended (p, m) ∧ Informed (p, m) ∧ Convenient (p, m)
∧ ❑ [Holds (m) ⇒ ¬ Convenient (p, m)]

This obstacle could be named LastMinuteImpediment: it refers to a participant invited to a meeting whose date/location was first convenient to her and then no longer convenient when the meeting took place. A scenario satisfying this assertion is straightforward in this case.

Assuming the domain theory takes the form of a set of rules A ⇒ C, the general procedure is as follows [15].

Initial step: take O := ¬ GA
Inductive step: let A ⇒ C be the domain rule selected,
with C matching some literal L in O;
then μ := mgu (L, C);
O := O [L / A.μ]

(where 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, and F[F1/F2] denotes the result of replacing every occurrence of F1 in formula F by F2).

Every iteration in the procedure above produces potentially finer obstacles; it is up to the specifier to decide when to stop, depending on whether the obstacles obtained are meaningful and precise enough. In the example above only one iteration was performed. Regressing obstacle (O1) above further through a domain description like

Convenient (p, m) ⇒ m.Date **in** p.Constraints
∧ m.Location **in** p.Constraints

would have produced two sub-obstacles, 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 that can be derived 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 about participation in meetings:

(D') ∀ m: Meeting, p: Participant
Participates(p, m) ⇒ ∃ M: Belief_p(m.Date = M) ∧ m.Date = M

(where the deontic Belief_{ag} and Knows_{ag} operators for some agent *ag* are defined by: Knows_{ag}(P) ≡ Belief_{ag}(P) ∧ P).

Regressing the goal negation (NG) through property (D') now yields the following new obstacle that could be named ParticipantKnowsWrongDate:

(O2) ◊ ∃m: Meeting, p: Participant
Intended (p, m) ∧ Informed (p, m) ∧ Convenient (p, m)
∧ ❑ ∀M: ¬ [Belief_p(m.Date = M) ∧ m.Date = M]

Further backtracking on other applicable rules would generate other obstacles obstructing the goal *Achieve* [InformedParticipantsAttendance] such as, e.g., ParticipantNotInformedInTime, InvitationNotKnown, etc.

The examples above exhibit a simplified procedure for goals having the *Achieve* pattern P ⇒ ◊ Q:

1. Negate the goal, which yields a pattern ◊ (P ∧ ❑ ¬ Q);
2. Find *necessary* conditions for the target condition *Q* in the domain theory;
3. Replace the negated target condition in the pattern resulting from step 1 by the negated necessary conditions found; each such replacement yields a potential obstacle. If needed, apply steps 2, 3 recursively.

A dual simplified procedure can be used for goals having the *Maintain* pattern P ⇒ ❑ Q.

In practice, the domain theory does not necessarily need to be very rich at the beginning. Given a target condition *Q* in the goal P ⇒ ◊ Q, the requirements engineer may *incrementally* elicit necessary conditions for *Q* 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. 2 shows a goal AND-refinement tree, derived by instantiation of a frequent refinement pattern from [5], together with corresponding obstacles that were generated by regression (universal quantifiers have been left implicit).

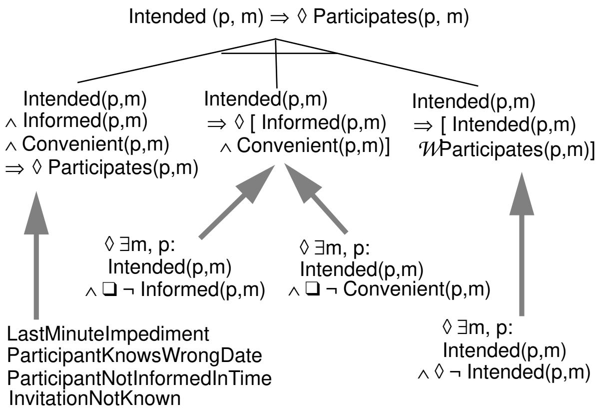

Fig. 2 - Goal refinement and obstacles derived by regression

4.2 Refining Obstacles

Goals are AND/OR refined into subgoals until constraints assignable to agents are obtained. Likewise, obstacles can be AND/OR refined into subobstacles until "primitive" obstacles are obtained. By primitive, we mean obstacles for which (i) negative scenarios can be found easily to show their feasibility, and (ii) effective ways to resolve them can be envisioned.

Obstacle OR-refinement yields *sufficient* subobstacles to establish the obstacle; each OR-refinement of an obstacle obstructs the goal obstructed by this obstacle (see the definitions of refinement and obstruction in Sections 2.1.1 and 3.1, respectively).

The AND/OR refinement of obstacles proceeds exactly the same way as goals --except that only a few alternative OR-refinements of goals are generally considered whereas, in the case of obstacles, one may wish to identify as many alternative subobstacles as possible (see Section 3).

Section 4.1 already contained examples of obstacle refinements; the obstacle LastMinuteImpediment was 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. 2 shows an example of OR-refinement of the obstacle obstructing the goal in the middle of the goal tree; this obstacle, not explicitly represented there, has been formally OR-refined into the two subobstacles in the middle (which could be named MeetingNeverNotified and MeetingNeverConvenient). The latter subobstacles may be refined in turn. Similarly, the obstacle ParticipantKnowsWrongDate that was derived in Section 4.1 could be OR-refined into subobstacles like WrongDateCommunicated, ParticipantConfusesDates, etc. The constraint ParticipantsConstraintsProvided that appeared in Section 2.1.2 is obstructed by the obstacle ConstraintsNotProvided; the latter can be OR-refined into subobstacles like ConstraintsNotSent (with EmailNotCheckedRegularly, RequestForgotten, ParticipantTooBusy as new OR-refinements), InaccurateConstraintsSent (with WrongDateRange, ConstraintsConfusion as new OR-refinements), UnprocessableConstraints (with MessageCorrupted, WrongFormat as new OR-refinements), etc.

The AND/OR refinement of obstacles may be seen as a goal-driven form of fault-tree analysis [17]. It can be done in a same informal way through interaction with domain experts and clients; some formal support can however be provided. Beside iterative regression through domain descriptions, one may identify *obstacle refinement patterns* that are formally proved correct *once for all* and reused in multiple contexts by instantiation --in the same spirit as goal refinement patterns [5]. Figures 3 and 4 show a sample of refinement patterns for obstacles that obstruct *Maintain* and *Achieve* goals, respectively (the outer ◊-operator has been left implicit). The correctness of these patterns was proved formally using the STeP verification tool [19].

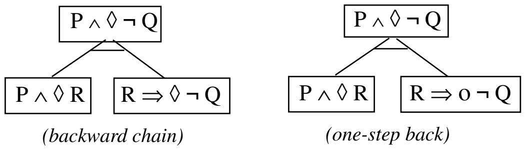

Fig.3 - Refinement patterns for obstacles to goal P ⇒ ❑ Q

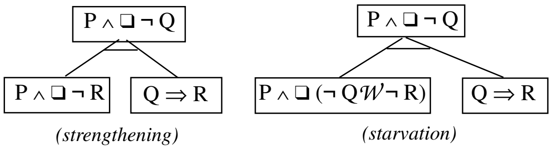

Fig.4 - Refinement patterns for obstacles to goal P ⇒ ◊ Q

As an example of reusing the one-step back pattern in Fig. 3, consider a telecommunication system and the goal stating that calls from a phone to certain numbers specified by the subscriber should be blocked [32]. This constraint, under the responsibility of the Originating Call Screening feature/agent, may be formalized by

∀ d1, d2: DirectoryNumber
Forbidden (d1, d2) ⇒ ❑ ¬ Connected (d1, d2)

The domain property

Calling (d1,d3) ∧ CallForwarding (d3,d2) ⇒ o Connected (d1, d2)

suggests reusing the one-step back pattern with the following instantiations:

P: Forbidden (d1, d2)
Q: ¬ Connected (d1, d2)
R: Calling (d1,d3) ∧ CallForwarding (d3,d2)

The following subobstacle has thereby been derived:

∃ d1, d2, d3: DirectoryNumber
Forbidden (d1, d2) ∧ ◊ Calling (d1,d3) ∧ CallForwarding (d3,d2)

As another example, consider a general resource management system and the goal

∀ u: User, r: Resource
Requesting (u, r) ∧ Available (r) ⇒ ◊ Gets (u, r)

Using the starvation pattern in Fig. 4 with instantiations

P: Requesting (u, r) $\wedge$ Available (r)

Q: Gets (u, r) , R: $\neg \exists$ u' u: Gets (u', r)

together with the domain property

Gets (u, r) $\Rightarrow \neg \exists$ u' u: Gets (u', r)

one derives the starvation obstacle

Requesting(u,r) $\wedge$ Available(r) $\wedge \Box$ ($\neg$ Gets(u,r) $\mathcal{W}$ $\exists$u' u: Gets(u',r))

The various examples above show that *the more an obstacle is refined the closer one gets to an explicit scenario*. Obstacle refinement patterns may thus be used for deriving feasible scenarios as well.

4.3 Using Obstacle Classifications

As mentioned in Section 2.3, goals can be classified by type of requirements they will drive with respect to the agents concerned. For each goal category, specific obstacle categories may be defined, e.g., Starvation obstacles to Satisfaction goals; Misinformation and Forgetting obstacles to Information goals; Hazard obstacles to Safety goals; Threat obstacles to Security goals; etc. Knowing the category of a goal may prompt a search for obstructing obstacles in the corresponding categories. More specific goal subcategories (like Confidentiality or Integrity subcategories of Security goals [1]) will result in more focussed search of corresponding obstacles.

On another hand, one may use domain-independent heuristics for identifying obstacles from goal formulations. Here are a few examples of such heuristics.

Agent failure: can an agent assigned to some constraint fail to guarantee it through the actions it/she has to perform? If so, why? Can an agent fail to have access to the objects it/she has to control? Can agents fail to communicate? HAZOP-like guidewords can also be used to direct more questions [17], such as: can an agent produce values for an object attribute it/she controls that are higher/lower than expected? Can it/she guarantee assigned constraints only partially? Can there be side-effects to performed actions?

Instance confusion [23]: can multiple instances of a same type of object be confused by some agent? In the meeting scheduler example, this heuristics might have helped identifying several obstacles among those derived formally, e.g., participants confusing meetings and dates, meeting initiators confusing participants which results in wrong people being invited, confusion in constraints, etc. In an ambulance dispatching system [28], an obstacle like an ambulance going to a wrong accident could be identified thereby.

Wrong agent beliefs: can object states recorded in the memory of some agent be different from the actual states of those objects? In the meeting scheduler example, this heuristics might have helped identifying obstacles like ParticipantKnowsWrongDate (see above); for an electronic reviewing process an obstacle like ReviewerKnowsWrongDeadline could be identified in a similar way.

Obstacles can also be identified by analogy with obstacles in similar systems, using analogical reuse techniques [20].

5. RESOLVING OBSTACLES

Once feasible and domain-consistent obstacles have been identified, they need to be resolved in some way or another depending on the severity of the obstacle and its consequences, and on the likelihood of its occurrence. The result will be an updated goal structure (see Fig. 1). Various resolution strategies can be followed by the requirements engineer. The first one is to just tolerate the obstacle because the consequences of its occurences are not serious enough to transform goals, add new ones, or consider alternative designs. For intolerable obstacles, we propose three other strategies that go into the latter direction.

5.1 Deidealizing Goals

A goal obstructed by an obstacle may be transformed to make the obstacle disasppear. Let us suggest the technique on an example first.

Consider the obstacle ParticipantNotInformedInTime in Fig. 2 which obstructs the goal

Intended (p, m) $\wedge$ Informed (p, m) $\wedge$ Convenient (p, m)
$\Rightarrow \Diamond$ Participates(p, m)

The idea is to make the obstructed goal more liberal, that is, to weaken it so that it covers the obstacle. In this case the goal weakening is achieved by strengthening its antecedent:

Intended (p, m) $\wedge$ Informed**InTime** (p, m) $\wedge$ Convenient (p, m)
$\Rightarrow \Diamond$ Participates(p, m)

The predicate Informed**InTime** (p, m) is derived from the corresponding obstacle; it requires participants to be kept informed during a period starting at least *N* days before the meeting date:

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

Once this more liberal goal is obtained, the predicates that were transformed to weaken the goal are propagated in the goal tree to replace their older version everywhere; this generally results in strengthened goals at other places. The result of the change propagation in the tree shown in Fig. 2 will produce a strengthened goal in the middle of the tree, namely,

Intended(p, m) $\Rightarrow \Diamond$ [Informed**InTime**(p, m) $\wedge$ Convenient(p, m)]

The deidealization procedure thus has two steps:

(1) *Weaken* the goal formulation to obtain a more liberal version that covers the obstacle. Syntactic generalization operators [11, 15] can be used here such as adding a disjunct, removing a conjunct, or adding a conjunct in some antecedent.

(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.

For the starvation obstacle derived at the end of Section 4.2, the first step applied to the goal formalized there might produce the deidealized goal

Requesting(u, r) $\wedge$ [Available(r) $\mathcal{W}$ Gets(u, r)] $\Rightarrow \Diamond$ Gets(u, r)

The new version Available(r) $\mathcal{W}$ Gets(u, r) of the predicate Available(r) states that the resource must be kept permanently available unless the requesting user gets it; this new version has to be propagated into the goal AND-tree. The goals that refer to this new predicate as target condition might be operationalized by a reservation procedure.

For an ambulance dispatching system, a deidealization of the goal

$\forall$ am: Ambulance, ac: Accident
Alerted (am, ac) $\Rightarrow \Diamond$ Intervention(am, ac)

from the following obstacle (obtained by a starvation pattern):

$\Diamond$ [Alerted(am,ac) $\wedge \neg$ Intervention(am, ac) $\mathcal{U}$ Breakdown(am)]

will produce the transformed goal

Alerted(am,ac) $\wedge \neg$ Breakdown(am) $\mathcal{W}$ Intervention(am,ac)
$\Rightarrow \Diamond$ Intervention(am, ac)

The propagation will result in strengthened goals like

Reported (ac, s) $\Rightarrow \Diamond \exists$am: Alerted(am,ac)
$\wedge \neg$ Breakdown(am) $\mathcal{W}$ Intervention(am,ac)

to be refined and deidealized in turn.

5.2 Adding New Goals

Beside transformations of obstructed goals, new goals may need to be introduced in the goal graph in order to resolve the obstacles. Three sub-strategies can be applied.

1. *Obstacle prevention:* A new goal is introduced which has the *Avoid* pattern P $\Rightarrow \Box \neg$ Q, where *Q* is intanstiated to the assertion capturing the obstacle. AND/OR refinement and obstacle analysis are then applied to the new goal in turn.

Back to our meeting scheduler example, an obstacle like WrongDateCommunicated obstructing the goal Achieve [InformedParticipantsAttendance] mentioned earlier would produce the new goal Avoid [WrongDateCommunicated].

2. *Goal restoration:* It is often the case that obstacles that result from unexpected behaviours of agents in the environment cannot be avoided. A first possibility in such cases is to restore the obstructed goal from states in which the obstacle occurs. This leads to the introduction of a new goal taking the form $O \Rightarrow \Diamond GA$ where *O* and *GA* denote the obstacle and goal assertions, respectively. This strategy could be followed for the obstacle ParticipantConfusesDates that obstructs the goal Achieve[InformedParticipantsAttendance] as well.

3. *Obstacle mitigation:* Another possibility in cases where the obstacle cannot be avoided is to seek effective ways of mitigating its consequences. The idea here is to attenuate the effects of obstacle occurrences by introduction of appropriate new goals. For the first obstacle derived in Section 4.1, named LastMinuteImpediment (see Fig. 2), one would introduce the new goal

Achieve [ImpedimentNotified]

as a subgoal of the deidealized goal

Intended (p, m) $\wedge$ Informed (p, m) $\wedge$ Convenient (p, m)
$\Rightarrow \Diamond$ [Participates(p, m) $\vee$ *Excused (p, m)*]

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

5.3 Reconsidering Agent Assignments

Last but not least, one may overcome the obstacle by considering alternative designs with different agent assignments so that the obstacle no longer exists. This may result in different system proposals, in which more or less functionality is automated and in which the interaction between the software and its environment may be quite different.

Back to our meeting scheduler example, one might overcome the obstacle ParticipantNotResponsive to the goal ParticipantsConstraintsProvided by introducing ElectronicAgenda objects to be made accessible to the Scheduler agent. Other designs would correspond to alternative goals of participant's constraints being requested to the participant's secretary instead --by email, or by phone call (to overcome the EmailNotCheckedRegularly, ParticipantTooBusy obstacles); etc.

In the electronic reviewing example, one could introduce a software agent for checking that no occurrences of the reviewer's name are found in the review (to overcome the obstacle NonAnonymousReview); a software agent for checking destination tables (to overcome the obstacle MessageSentToWrongPerson); and so on.

6. CONCLUSION

In order to get high-quality software, it is of upmost imprtance to reason about agent behaviour 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 [28] and has motivated previous work in the field. For example, planning techniques have been proposed for exhibiting scenarios that violate given requirements [2], [10]; model-checking techniques have been used to generate counterexamples to claims [13]; informal fault-tree analysis has proven popular in the safety-critical systems community [17]; HAZOP-based techniques have been proposed for forward propagation of perturbations from input variables to output variables in operational specifications [25].

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 *goal* level. The earlier such analysis is started, the more freedom is left to resolve the obstacles. Moreover, goals provide a precise entry point for starting analysis in a more focussed way like, e.g., fault-tree construction 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-driven elaboration of requirements and systematic obstacle analysis proceed hand-in-hand.

Our work is based on the notion of obstacles to goals first introduced by Potts [23]. We have presented various formal and heuristic techniques for obstacle identification and refinement from goal specifications and domain properties, and for resolving obstacles through goal deidealization, introduction of new goals to avoid or mitigate the obstacles, and investigation of alternative agent assignments. Domain knowledge was seen to play an important role in some of these techniques; however, as we pointed out, such knowledge can be elicited gradually during obstacle analysis.

Our obstacle identification techniques allowed us to formally derive the 17 obstacles informally identified in [23] from our meeting scheduler case study [16], 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 be retained.

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 identifying and resolving subtle obstacles. Our plan is to integrate these techniques in the KAOS/GRAIL environment [6] in the near future so that large-scale experimentation on industrial projects from our tech transfer institute can take place.

ACKNOWLEDGEMENT

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). Warmest thanks to Colin Potts for insightful discussions and suggestions on a preliminary draft of this paper. David Till made useful comments on an earlier version. Thanks are also due to two ICSE reviewers for helpful feedback.

REFERENCES

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

[2] J.S. Anderson and S. Fickas, "A Proposed Perspective Shift: Viewing Specification Design as a Planning Problem", *Proc. IWSSD-5 - 5th Intl. Workshop on Software Specification and Design*, IEEE, 1989, 177-184.

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

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

[5] R. Darimont and A. van Lamsweerde, "Formal Refinement Patterns for Goal-Driven Requirements Elaboration", *Proc. FSE'4 - Fourth ACM SIGSOFT Symp. on the Foundations of Software Engineering*, San Francisco, October 1996, 179-190.

[6] R. Darimont, E. Delor, P. Massonet, and A. van Lamsweerde, "GRAIL/KAOS: An Environment for Goal-Driven Requirements Engineering", *Proc. ICSE'97 - 19th Intl. Conf. on Software Engineering*, Boston, May 1997, 612-613.

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

[8] M. Feather, "Language Support for the Specification and Development of Composite Systems", *ACM Trans. on Programming Languages and Systems* 9(2), Apr. 87, 198-234.

[9] M. Feather, "Towards a Derivational Style of Distributed System Design", *Automated Software Engineering* 1(1), 31-60.

[10] S. Fickas and R. Helm, "Knowledge Representation and Reasoning in the Design of Composite Systems", *IEEE Trans. on Software Engineering*, June 1992, 470-482.

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

[12] M. Jackson and P. Zave, "Domain Descriptions", *Proc. RE'93 - 1st Intl. IEEE Symp. on Requirements Engineering*, Jan. 1993, 56-64.

[13] D. Jackson and C.A. Damon, "Elements of Style: Analyzing a Software Design Feature with a Counterexample Detector", *Proc. ISTA '96 - Intl. Symp. on Software Testing and Analysis*, ACM Softw. Eng. Notes Vol. 21 No. 3, 1996, 239-249.

[14] R. Koymans, *Specifying message passing and time-critical systems with temporal logic,* LNCS 651, Springer-Verlag, 1992.

[15] A. van Lamsweerde, "Learning Machine Learning", in: *Introducing a Logic Based Approach to Artificial Intelligence*, A. Thayse (Ed.), Vol. 3, Wiley, 1991, 263-356.

[16] A. van Lamsweerde, R. Darimont and P. Massonet, "Goal-Directed Elaboration of Requirements for a Meeting Scheduler: Problems and Lessons Learned", *Proc. RE'95 - 2nd Int. Symp. on Requirements Engineering*, York, IEEE, 1995.

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

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

[19] Z. Manna and the STep Group, "STeP: Deductive-Algorithmic Verification of Reactive and Real-Time Systems", *Proc. CAV'96 - 8th Intl. Conf. on Computer-Aided Verification*, LNCS 1102, Springer-Verlag, July 1996, 415-418.

[20] P. Massonet and A. van Lamsweerde, "Analogical Reuse of Requirements Frameworks", Proc. RE-97 - *3rd Int. Symp. on Requirements Engineering*, Annapolis, 1997, 26-37.

[21] J. Mylopoulos, L. Chung and B. Nixon, "Representing and Using Nonfunctional Requirements: A Process-Oriented Approach", *IEEE Trans. on Sofware. Engineering*, Vol. 18 No. 6, June 1992, pp. 483-497.

[22] D.E. Perry, "The Inscape Environment", *Proc. ICSE-11, 11th Intl. Conf. on Software Engineering*, 1989, pp. 2-12.

[23] C. Potts, "Using Schematic Scenarios to Understand User Needs", *Proc. DIS'95 - ACM Symposium on Designing interactive Systems: Processes, Practices and Techniques*, University of Michigan, August 1995.

[24] B. Potter, J. Sinclair and D. Till, *An Introduction to Formal Specification and Z.* Second edition, Prentice Hall, 1996.

[25] J.D. Reese and N. Leveson, "Software Deviation Analysis", *Proc. ICSE'97 - 19th Intl. Conference on Software Engineering*, Boston, May 1997, 250-260.

[26] W. N. Robinson, "Integrating Multiple Specifications Using Domain Goals", *Proc. IWSSD-5 - 5th Intl. Workshop on Software Specification and Design*, IEEE, 1989, 219-225.

[27] D.S. Rosenblum, "Towards a Method of Programming with Assertions", *Proc. ICSE-14, 14th Intl. Conf. on Software Engineering*, 1992, pp. 92-104.

[28] K. Ryan and S. Greenspan, "Requirements Engineering Group Report", Succeedings of *IWSSD8 - 8th Intl. Workshop on Software Specification and Design*, *ACM Software Engineering Notes, Sept. 1996, 22-25.*

[29] R. Waldinger, "Achieving Several Goals Simultaneously", in *Machine Intelligence*, Vol. 8, E. Elcock and D. Michie (Eds.), Ellis Horwood, 1977.

[30] K. Yue, "What Does It Mean to Say that a Specification is Complete?", *Proc. IWSSD-4, Fourth International Workshop on Software Specification and Design*, Monterey, 1987.

[31] P. Zave, "Classification of Research Efforts in Requirements Engineering", *Proc. RE'95 - 2nd IEEE Int. Symposium on Requirements Engineering*, March 1995, 214-216.

[32] P. Zave, "Secrets of Call Forwarding: A Specification Case Study", *Proc. 8th IFIP Conf. on Formal Description Techniques for Distributed Systems and Communication Protocols*, Chapman & Hall, 1996, 153-168.