Risk-driven Engineering of Requirements for Dependable Systems

Axel van Lamsweerde
ICTEAM Institute, Dept. Computing Science
Université catholique de Louvain
B-1348 Louvain-la-Neuve (Belgium)
avl@info.ucl.ac.be

Abstract. Engineering the right software requirements under the right environment assumptions is a critical precondition for developing the right software. Requirements completeness, in particular, is known to be among the most critical and difficult software engineering challenges. Missing requirements often result from poor risk analysis at requirements engineering time. A natural inclination to conceive over-ideal systems prevents adverse conditions from being properly identified and, when likely and critical, resolved through adequate countermeasure requirements.
The paper overviews a model-based approach for integrating risk analysis in requirements engineering. The approach is aimed at anticipating exceptional conditions in which the target system should behave adequately. In a goal-oriented modeling framework, obstacles are introduced as as preconditions for the non-satisfaction of system goals. Following the *identify*-assess-*control* cycle of risk analysis, the paper reviews a variety of formal techniques available for generating obstacles, for assessing their likelihood and the severity of their consequences, and for resolving them through countermeasures whose integration in the system model results in increased requirements completeness.

Keywords. Goal-oriented requirements engineering, risk analysis, formal multi-view models, hazard analysis, model checking, inductive learning, refinement patterns.

1. Introduction

Requirements engineering (RE) refers to the analysis of the problems experienced with an existing system (the *systems-as-is*) and the identification and evaluation of the opportunities, objectives, and options for a new system (the *system-to-be*). The outcome of this process is a requirements document in which the objectives of the system-to-be, the distribution of responsibilities, and the software functionalities, qualities, constraints, and assumptions are organized and specified precisely for subsequent development and maintenance.

The RE task is intrinsically difficult. We need to produce a complete, adequate, consistent, and well-structured set of measurable software requirements and environmental assumptions from incomplete, imprecise, and sparse material originating from multiple, often conflicting sources [Lam09]. The target *system* comprises the software-to-be together with environment components such as people, devices, and pre-existing software.

The RE task is critical as well. Requirements-related errors are recognized to be the most numerous, persistent, expensive, and dangerous types of software errors [Lam09]. They result in cost overruns, delivery delays, failure to meet expectations or even degradations in the environment controlled by the software.

Requirements completeness in particular is among the most critical and difficult challenges. Missing requirements and assumptions are known to be the major cause of software failure [Lam09]. They often result from a lack of anticipation of unexpected conditions under which the software should behave adequately. A natural inclination to conceive over-ideal systems prevents adverse conditions from being properly handled.

Risk analysis should therefore be at the heart of the requirements engineering process [Lam00, Fea03, Lam09, As11, Lun11]. A *risk* is commonly defined as an uncertain factor whose occurrence may result in some loss of satisfaction of a corresponding objective. A risk has a *likelihood of occurrence* and one or several undesirable *consequences* associated with it. Each consequence is uncertain as well; it has a likelihood of occurrence if the risk occurs. A consequence has a *severity* in terms of degree of loss of satisfaction of the corresponding objective. The likelihood of a risk should not be confused with the likelihood of a consequence of the risk; for example, the likelihood of the risk of the GPS device not working properly inside a mobilized ambulance is not the same as the likelihood of the consequence that the ambulance might get lost if the GPS is not working.

Depending on the category of objective being obstructed, risks may correspond to safety hazards [Lut93, Lev95, Lev02], security threats [Amo94, Lam94a], inaccuracies between software variables and the environment quantities they should reflect [Lam00], and so forth.

The paper overviews a systematic approach for integrating risk analysis in model-based RE. The models we consider inter-relate the goals to be achieved by the system, the conceptual objects involved in their specification, the agents responsible for the goals, the operations needed to ensure the goals, and the agent behaviors required to meet them. To enable formal reasoning about goals when and where needed, a real-time linear temporal logic is used to specify them.

In such models, obstacles are introduced as a natural abstraction for risk analysis [Lam00]. An *obstacle* to a goal is a precondition for the non-satisfaction of this goal. Obstacle analysis iterates on three steps:

- *Identification:* as many obstacles as possible to every leaf goal in the goal refinement graph should be identified;
- *Assessment:* the likelihood and severity of each obstacle should be assessed;
- *Control:* likely and critical obstacles should be resolved by integration of appropriate countermeasures in the goal model.

Obstacle analysis has been applied to a variety of mission-critical systems –see, e.g., [Dar07, Lut07].

The paper reviews and illustrates the techniques available to date for supporting *the identify-assess-control* cycles of obstacle analysis.

- For obstacle *identification*, we may use a formal regression calculus to derive obstacles from goals and domain properties [Lam00], or instantiate goal obstruction patterns to shortcut such derivations [Lam00], or combine model checking and inductive learning to generate a domain-complete set of obstacles [Alr12].
- For obstacle *assessment*, we need to enrich our models with a probabilistic layer allowing us to estimate the likelihood of fine-grained obstacles and

propagate these through the obstacle and goal models in order to determine the severity and likelihood of obstacle consequences [Cai12].

- For obstacle *resolution*, we may explore alternative countermeasures by use of systematic model transformations that encode risk-reduction tactics such as *reduce risk likelihood*, *avoid risk*, *reduce consequence likelihood*, *avoid risk consequence*, or *mitigate risk consequence*. Most appropriate countermeasures may then be selected based on the likelihood of obstacles and the severity of their consequences.

The paper is organized as follows. Section 2 introduces some necessary background on goal-oriented system modeling. The basic concepts of goal, domain property and agent are defined and interrelated; the mechanisms of refinement and operationalization are introduced as a basis for elaborating a goal-oriented model; the specification formalism for analyzing goal models is outlined. Section 3 introduces obstacles and obstacle analysis as a goal-anchored form of risk analysis to complete goal models. Section 4 presents various techniques for obstacle identification, namely, the formal regression of goal negations through available domain properties, the use of formal obstruction patterns, and the combined use of model checking and inductive learning for obstacle generation. Section 5 introduces a probabilistic framework for assessing the likelihood and severity of obstacles. Section 6 then reviews a number of obstacle resolution operators available for integrating countermeasures to likely and severe obstacles in the goal model.

2. Goal-Oriented Model Building for Requirements Engineering

In order to capture the multiple system facets relevant to the RE process, a model should integrate complementary views [Lam09].

- The *intentional* view captures the system objectives as functional and non-functional goals together with their mutual contribution links.
- The *structural* view captures the conceptual objects referred to in the other views, their structure and their inter-relationships.
- The *responsibility* view captures the agents forming the system, their responsibilities with respect to system goals, and their interfaces with each other in terms of the state variables they monitor and control.
- The *functional* view captures the services the system should provide in order to operationalize its goals.
- The *behavioral* view captures the behaviors required for the system to satisfy its goals. Interaction scenarios illustrate expected interactions among specific agent instances whereas state machines prescribe classes of behaviors of any agent instance on the state variables it controls.

Section 2.1 introduces these various model views and their integration mechanism. Section 2.2 introduces the real-time logic used for specifying goals to enable their formal analysis. Sections 2.3 and 2.4 introduce some basic techniques for verifying the correctness of goal refinements and goal operationalizations, respectively.

2.1. Goal-Oriented Modeling

The intentional view. The target system is intended to meet a number of objectives. These are to be highlighted as first-class citizens and interrelated. A *goal* is a prescriptive statement of intent the system should satisfy through cooperation of its agents [Dar93, Lam01]. An *agent* is an active system component playing some role in goal satisfaction through adequate control of state variables.

The finer-grained a goal is, the fewer agents are involved in its satisfaction. A *requirement* is a goal under responsibility of a single agent of the software-to-be. An *expectation* is a goal under responsibility of a single agent in the environment of the software-to-be. Expectations form one kind of assumption we need to make for the system to satisfy its goals.

When reasoning about goal satisfaction in the RE process, we often need to use *domain properties*. These are descriptive statements about the problem world, unlike goals which are prescriptive. They are expected to hold invariably regardless of how the system will behave. The distinction between descriptive and prescriptive statements is important. Goals may need to be negotiated with stakeholders, prioritized, weakened in case of conflict, or strengthened in case of unacceptable exposure to risks. Unlike prescriptive statements, domain properties are not subject to such decisions.

A goal is either a behavioral goal or a soft goal. A *behavioral* goal prescribes intended system behaviors declaratively. It implicitly defines a maximal set of admissible behaviors. Behavioral goals can be *Achieve* or *Maintain*/*Avoid* goals. An *Achieve goal* prescribes a *TargetCondition* to be established sooner or later when some *CurrentCondition* holds. A *Maintain goal* prescribes a *GoodCondition* to be maintained (similarly, an *Avoid goal* prescribes some *BadCondition* to be avoided).

Unlike behavioral goals, a *soft goal* cannot be established in a clear-cut sense. It prescribes preferences among alternative system behaviors, being more satisfied along some alternatives and less satisfied along others.

Behavioral goals are therefore used for deriving system operations to satisfy them [Dar93, Let02b] whereas soft goals are used for comparing alternative options to select preferred ones [Ch00, Let04].

Those goal types should not be confused with a categorization into functional goals, underlying system services, and non-functional goals, prescribing qualiy of service. For example, a confidentiality goal *Avoid [SensitiveInformationDisclosed]* is traditionally considered as non-functional; it is a behavioral goal though.

A *goal model* is basically an annotated AND/OR graph showing how higher-level goals are satisfied by lower-level ones (goal *refinement*) and, conversely, how lower-level goals contribute to the satisfaction of higher-level ones (goal *abstraction*) [Lam01]. The top goals are the highest-level ones still in the system scope whereas the leaf goals are assignable to single agents as requirements or expectations. In a goal model, an AND-refinement link relates a goal to a set of subgoals called *refinement*; this means that the parent goal can be satisfied by satisfying all subgoals in the refinement. A goal node can be OR-refined into multiple AND-refinements; each of these is called *alternative* for achieving the parent goal. The meaning of multiple alternative refinements is that the parent goal can be satisfied by satisfying the conjoined subgoals in any of the alternative refinements.

Fig. 1 shows a goal model fragment for an ambulance dispatching system that will be used as a running example throughout the paper. An AND-refinement is denoted by an arrow joining subgoals to the parent goal; multiple incoming arrows indicate an OR-

refinement. The figure also shows a leaf goal assignment to the AmbulanceStaff agent. Home-shaped nodes represent domain properties required for refinement correctness.

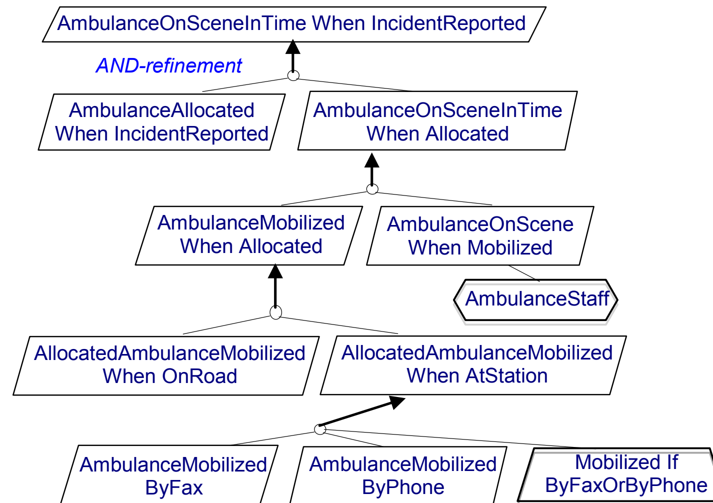

Figure 1. Portion of a goal graph for an ambulance dispatching system

The nodes in a goal model are decorated by annotations to characterize the corresponding goal – such as its precise definition, an optional formal specification of the goal (see Section 2.2), the goal's priority level, etc.

The structural view. Conceptual objects capture the concepts referred to by the system goals and domain properties. They are interrelated and defined precisely in an *object model* as entities, associations, agents, or events. This model provides the concept definitions and domain properties used in the other models. In particular, the object attributes and associations define the system's *state variables* in terms of which goals, agents, operations, and behaviors are specified in the other system views. An object model is represented by an annotated UML class diagram, where annotations capture individual object features such as a precise definition of the object, its attributes, relevant domain properties associated with it, initial values when an object instance appears in the system, etc. [Lam09].

The responsibility view. Agents were already introduced as active system components that are responsible for the leaf goals in a goal model. The *agent model* captures the distribution of responsibilities within the system together with the *capabilities* of every agent. The latter are defined in terms of the state variables from the object model that the agent can monitor or control. An agent model thus shows the system scope and the boundary between the software-to-be and its environment.

The functional view. An *operation model* captures the system operations in terms of their individual features and their links to the goal, object, agent, and behavior models. This model specifies, for each operation, its signature, the descriptive *domain pre-and postconditions* that intrinsically characterize the state transitions produced by the operation in the problem world, and the prescriptive *required precondition*, *trigger condition*, and *required post condition*s that must further constrain any application of the operation for each underlying goal to be satisfied [Dar93, Let02b].

The explicit linking of operational specifications to the underlying system goals provides a rich basis for satisfaction arguments, traceability management, and evolution support [Lam09].

The behavioral view. A *behavior model* captures desired system behaviors. Global behaviors are obtained by parallel composition of agent behaviors. The latter are made explicit through scenarios and state machines. A scenario shows sequences of interactions among specific agent instances. It is represented by a UML sequence diagram. A state machine shows sequences of state transitions for the variables controlled by any agent instance within some class. Such transitions are caused by operation applications or by external events. A state machine is represented by a UML state diagram or by a labelled transition system (LTS) depending on the type of analysis we want to perform on it [Lam09].

View integration. The complementary views of the target system are integrated through inter-model links constrained by rules for structural consistency and completeness of the overall model. For example, *responsibility* links connect leaf goals in the goal model and agents in the agent model; *concern* links connect goals in the goal model and the conceptual objects in the object model these goals refer to; *operationalization* links connect leaf goals in the goal model and the operations ensuring them in the operation model; scenarios or state machines in the behavior model are connected to behavioral goals by *coverage* links. The rules constraining inter-model links allow us to check the structural completeness and consistency of the overall model, e.g., *"every conceptual item referenced by a goal specification in the goal model must appear as an attribute or object in the object model, and vice versa"; "an agent responsible for a goal must have the capability of controlling the variables constrained by the goal specification and of monitoring the variables to be evaluated in it", "every operation in the operation model must operationalize at least one leaf goal from the goal model"*, etc.

2.2. Specifying model items

The approach outlined here is a "two-button" one where the formal analysis button is pressed only when and where needed. The button pressed by default is the semi-formal one, where the modeler is using the graphical notations and supporting tools to elaborate her models, perform static semantics checks on them through queries on the model database, generate HTML files for model browsing, generate UML use cases and other derived diagrams, and generate the requirements document [Obj04].

Formal analysis of critical aspects in the models require a formal specification language for the goals, domain properties attached to objects, and pre- and postconditions on the operations. A linear real-time temporal logic (RT-LTL) is used for the goals, domain properties, and required trigger conditions (for the latter conditions, with past operators only). A simple state-based, Z-like language is used for the domain and required pre- and postconditions.

The following logical connectives are used: $\land$ (*and*), $\lor$ (*or*), $\neg$ (*not*), $\rightarrow$ (*implies*), $\leftrightarrow$ (*equivalent*). The main temporal operators are fairly standard:

$\circ P$ (*P* holds in the next state)

$\Box P$ (*P* holds in every future state)

$P \; \mathbf{W} \; N$ (*P* holds in every future state *unless* *N* holds)

$\Diamond P$ (*P* holds in some future state)

$\square_{\leq d}\, P$ (P holds in every future state up to d time units)

$\lozenge_{\leq d}\, P$ (P holds within d time units)

$P \Rightarrow Q$ for $\square\, (P \rightarrow Q)$

The counterpart over past states is provided by past, "blackened" operators, e.g.,

$\bullet\, P$ (P holds in the previous state)

$@\, P$ $\bullet\, (\neg P) \wedge P$

These formulas are interpreted as usual over historical sequences H of states, e.g.,

$(H, i) \models \square\, P$ **iff** $(H, j) \models P$ **for** all $j \geq i$

$(H, i) \models \lozenge_{\leq d}\, P$ **iff** $(H, j) \models P$ **for** some $j \geq i$ with $\text{distance}(i,j) \leq d$

The o/• operators refer to the next/previous state within the smallest time unit. They are often used for expressing immediate obligations.

To make the presentation simpler, we will often provide propositional goal formalizations as illustrations. The techniques in this paper are however applicable to first-order goal formalizations as well.

For example, the top goal in Fig. 1 may be specified as follows:

Goal *Achieve* [AmbulanceOnSceneInTime When IncidentReported]
FormalSpec IncidentReported $\Rightarrow \lozenge_{\leq 14\ min}$ AmbulanceOnScene

Another goal might prescribe any mobilized ambulance to remain mobilized until intervention:

Goal *Maintain* [AmbulanceMobilized]
FormalSpec AmbulanceMobilized $\Rightarrow$ AmbulanceMobilized W AmbulanceOnScene

In goal specifications, the keywords prefixing goal names are used to indicate temporal specification patterns [Dar93] – e.g., *Achieve [P]* indicates a pattern $\lozenge_{\leq d}\, P$ on a target predicate P; *Avoid [P]* indicates a pattern $\square \neg P$; and so forth. Such patterns help writing the specification from informal prescriptive statements. They prove convenient for non-expert specifiers to use elementary temporal logic without knowing it.

The system's semantic picture is as follows. The global state of the system at some time position is the aggregation of the local states of all its agents at that time position. The local state of an agent at some time position is the aggregation of the states, at that time position, of all the state variables the agent controls. The state of a variable at some time position is a mapping from its name to its value at that time position. The system evolves synchronously from system state to system state, where the time distance between successive states is the smallest time unit defined in the RT-LTL language. (This time unit may be chosen arbitrarily small.) A system's state transition is caused by the application, by some agents, of applicable operations they may or must perform on the state variables they control. Operations are atomic; an operation applied in the current state maps the corresponding agent's state to the next state one smallest time unit later. As multiple trigger conditions may become true in the same state, the corresponding operations *must* fire simultaneously. We thus have true concurrency here; a system's state transition is composed of parallel transitions on local states. An interleaving semantics is not possible in view of the obligations expressed by trigger conditions.

The system's non-determinism arises from the non-deterministic behavior of its agents. While an agent *must* perform an operation when one of the operation's trigger conditions becomes true, the agent has the freedom to perform an operation or not

when its required preconditions are all true. Such non-determinism, while suitable at a more abstract level for declarative reasoning, must in general be removed when the specification is translated into a more operational language (e.g., for specification animation or other checks on the operational version) [Tra04]. A choice must then be made between an eager or lazy behavior scheme for each operation performed by the agent. In the *eager* behavior scheme, the agent performs the operation as soon as it can, that is, as soon as *all* required *preconditions* are true. This corresponds to a maximal progress property. In the *lazy* behavior scheme, the agent performs the operation when it is really obliged to do so, that is, when *one* of its required *trigger conditions* becomes true.

A system's behavior is then defined by a temporal sequence of system state transitions. The system *satisfies* a behavioral goal if the set of all its possible behaviors is included in the set of behaviors prescribed by the RT-LTL specification of the goal.

2.3. Goal refinement

As seen in Section 2.1, a goal AND-refinement means that the parent goal can be satisfied by satisfying all subgoals in the refinement. A first kind of model verification consists in checking that AND-refinements of behavioral goals in the goal model are "correct", that is, complete, consistent and minimal.

A set of goals $\{SG_1, ..., SG_n\}$ correctly refines a goal G in a domain theory Dom made of known domain properties iff

$\{SG_1, ..., SG_n, \text{Dom}\} \models G$ *completeness*

$\{SG_1, ..., SG_n, \text{Dom}\} \not\models \textbf{\textit{false}}$ *consistency*

$\{\wedge_{j \neq i} SG_j, \text{Dom}\} \not\models G$ for each $i \in [1..n]$ *minimality*

Such correctness checking is important; incomplete refinements result in missing requirements whereas non-minimal refinements produce unnecessary requirements. Several approaches can be followed to verify the correctness of a goal refinement.

Theorem proving. We might use a temporal logic theorem prover to prove that the subgoals conjoined with domain properties entail the parent goal. This is obviously a heavyweight approach requiring the assistance of an expert user. Moreover we get no real clue in case the verification fails.

Formal refinement patterns. A more lightweight and constructive approach consists in using formal patterns to check, complete, or explore refinements [Dar96, Let02a]. The idea is to build a catalogue of common refinement patterns that encode refinement tactics. The patterns in the catalogue are proved formally correct once for all, e.g., using a LTL theorem prover. They are then reused in matching situations through instantiation of their meta-variables. Fig. 2 and Fig. 3 show three frequent refinement patterns. The left pattern in Fig. 2 encodes the tactics of introducing an intermediate milestone goal; the right one encodes a decomposition-by-guard pattern. The pattern in Fig. 3 captures goal refinement through decomposition by cases.

The two top refinements in Fig. 1 were obtained using the milestone-driven pattern with *AmbulanceAllocated* and *AmbulanceMobilized* as milestone conditions, respectively. The next lower-level refinement instantiates the decomposition-by-cases pattern with *AmbulanceOnRoad* and *AmbulanceAtStation* as case conditions.

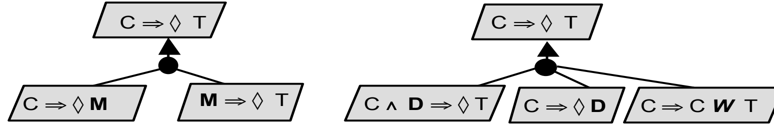

Figure 2. Formal refinement patterns: milestone-driven and guard introduction

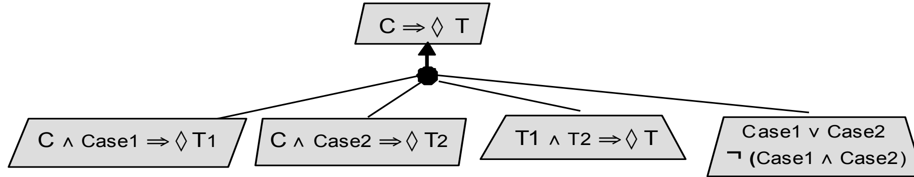

Figure 3. Formal refinement patterns: decomposition by cases

Refinement patterns support a constructive approach to refinement correctness. They may be used for guiding the refinement process. When a goal is partially refined, we may retrieve all matching patterns from the catalogue and thereby explore alternative ways of completing the partial refinement with missing subgoals [Dar96]. Similar patterns are used for obstacle refinement in obstacle trees, see Section 3.

Using a bounded SAT solver. To verify the correctness of a goal refinement, we may also make a roundtrip use of a bounded SAT solver. In view of the above definition of refinement completeness, we would like to know whether the temporal logic formula

$$SG_1 \land \ldots \land SG_n \land Dom \land \neg G$$

is satisfiable and, if so, find a historical sequence of states satisfying it.

To achieve this, a tool front-end can apply the following steps [Pon07]:

- ask the user to instantiate the above formula to selected object instances in order to obtain a propositional formula,
- translate the result into the input format required by the SAT solver,
- ask the user to determine a maximal length to bound counterexample traces,
- run the SAT solver, and
- translate the output back to the level of abstraction of the input model.

Such use of a bounded SAT solver allows partial goal models to be checked and debugged incrementally as the model is being built. The major payoff resides in the counterexample traces that may suggest missing subgoals.

2.4. Goal operationalization

A set of operations *operationalizes* a leaf goal in the goal model if the specification of their required precondition, trigger condition and required postcondition for that goal ensures that the goal is satisfied.

Verifying the correctness of goal operationalizations is important too; we must make sure that the operational specifications meet the intentional ones. For this we need a temporal logic semantics for operations [Let02b]. Let *op* denote an operation from the operation model, specified by a domain precondition *DomPre(op)*, a domain postcondition *DomPost(op)*, a set *ReqPre(op)* of required preconditions, a set

ReqTrig(op) of trigger conditions and a set *ReqPost(op)* of required postconditions, all constraining the operation so as to meet the goals operationalized by it. Let:

$$[|\ \mathrm{op}\ (\mathrm{in}, \mathrm{out})\ |] =_{\mathrm{def}} \mathrm{DomPre}\ (op) \wedge \circ\ \mathrm{DomPost}\ (op)$$

The semantics of required pre-, trigger-, and postconditions is then:

If $R \in \mathrm{ReqPre}\ (op)$ **then**

$$[|\ R\ |] =_{\mathrm{def}} (\forall *) (\ [|\ \mathrm{op}\ |] \Rightarrow R\ )$$

If $R \in \mathrm{ReqTrig}\ (op)$ **then**

$$[|\ R\ |] =_{\mathrm{def}} (\forall *) (\ R \wedge \mathrm{DomPre}\ (op) \Rightarrow [|\ \mathrm{op}\ |]\ )$$

If $R \in \mathrm{ReqPost}\ (op)$ **then**

$$[|\ R\ |] =_{\mathrm{def}} (\forall *) (\ [|\ \mathrm{op}\ |] \Rightarrow \circ\ R\ )$$

A set of required conditions R_1, ..., R_n on operations from the operation model correctly operationalizes a goal *G* [Let02b] *iff*

$[\lvert R_1 \rvert] \wedge ... \wedge [\lvert R_n \rvert] \models G$	*completeness*
$[\lvert R_1 \rvert] \wedge ... \wedge [\lvert R_n \rvert] \not\models$ ***false***	*consistency*
$G \models [\lvert R_1 \rvert] \wedge ... \wedge [\lvert R_n \rvert]$	*minimality*

Every operationalization defines a proof obligation. The correctness of a goal operationalization can be verified in different ways.

Bounded SAT solver. Similarly to goal refinement checking, we may make a roundtrip use of a bounded SAT solver. The temporal logic formula to be checked for satisfiability is here:

$$[|\ R_1\ |] \wedge ... \wedge [|\ R_n\ |] \wedge Dom \wedge \neg\ G$$

The FAUST toolset proceeds similarly to check bounded operationalizations and generate counterexample traces [Pon07].

Formal operationalization patterns. The principle is similar to goal refinement patterns. A catalogue of operationalization patterns is built and formally proved correct once for all [Let02b]. The patterns cover common goal specification patterns, e.g., *Achieve* goals of form $C \Rightarrow \Diamond_{\leq d}\ T$ or $C \Rightarrow \circ\ T$ and *Maintain* goals of form $C \Rightarrow T$ or $C \Rightarrow TWN$. The patterns are then reused in matching situations through instantiation of their meta-variables. Fig. 4 shows a pattern for operationalizing *Immediate Achieve* goals.

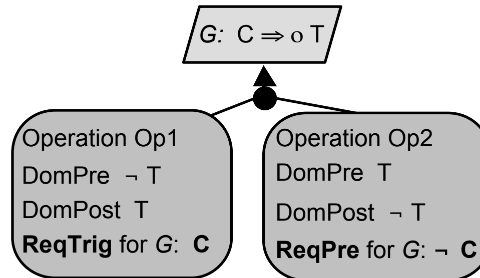

Figure 4. Operationalization pattern for *Immediate Achieve* goals

3. Obstacle Analysis for Risk-Driven Requirements Engineering

As introduced before, the goals identified in the early stages of the RE process are often too ideal. They are likely to be violated due to unintentional or malicious agent behaviors. For system robustness and requirements completeness, it appears essential to integrate risk analysis in the goal elaboration process in order to anticipate potential problems at RE time.

Risk analysis may be anchored on goal models through obstacle identification, assessment and resolution.

An *obstacle O* to goal *G* is a precondition for goal violation that satisfies the following conditions [Lam00]:

$\{O, Dom\} \models \neg G$ *goal obstruction*

$\{O, Dom\} \not\models \textit{\textbf{false}}$ *obstacle satisfiability in domain*

For behavioral goals, obstacles are existential properties that capture inadmissible behaviors called *negative scenarios*. Such behaviors should be feasible in view of known domain properties.

Obstacles may be classified in categories corresponding to the category of goals they are obstructing. Hazards are obstacles obstructing safety goals; threats are obstacles obstructing security goals (with subcategories such as disclosure, corruption or denial-of-service obstacles); inaccuracy obstacles obstruct accuracy goals constraining software variables with respect to the environment quantities they should reflect; misinformation obstacles obstruct information goals; and so forth.

For risk analysis to be reliable, obstacle completeness is highly desirable –at least for mission-critical goals. A set of obstacles O_1, ..., O_n to some goal *G* is *domain-complete* iff:

$\{\neg O_1, ..., \neg O_n, Dom\} \models G,$ *obstacle completeness*

that is, the goal is guaranteed to be satisfied if none of the obstacle conditions is satisfied. Note that the notion of obstacle completeness is relative to what is known about the domain. As the next section will show, obstacle analysis can be used to elicit relevant domain properties as well.

An *obstacle model* is a set of *goal-anchored* fault trees [Amo94, Lev95] where each fault tree is an AND/OR refinement tree showing how the corresponding goal can be violated. The root of the tree is the goal negation; the leaves are elementary obstruction conditions whose satisfiability and likelihood can be easily assessed.

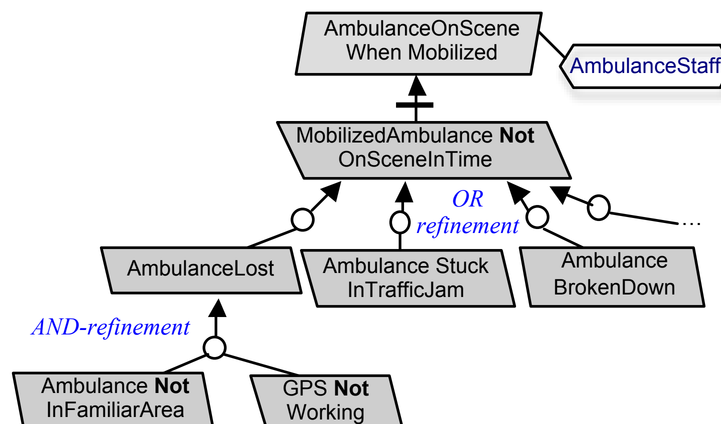

Figure 5. Portion of an obstacle tree [Lam09]

Fig. 5 shows a portion of an obstacle tree anchored on the leaf goal *AmbulanceOnSceneWhenMobilized* appearing in Fig. 1.

For risk analysis to be "correct", the obstacle refinements in obstacle trees must be correct.

- An AND-refinement of an obstacle O into sub-obstacles $SO_1, ..., SO_n$ should be a correct AND-refinement in view of known domain properties:

  $\{SO_1, ..., SO_n, Dom\} \models O$ *refinement completeness*

  $\{SO_1, ..., SO_n, \mathrm{Dom}\} \not\models \textbf{\textit{false}}$ *domain consistency*

  $\{\wedge_{j\text{-}I}\, SO_j, \mathrm{Dom}\} \not\models O$ for each $i \in [1..n]$ *minimality*

- An OR-refinement of an obstacle O into sub-obstacles $SO_1, ..., SO_n$ should meet the following conditions:

  $\{SO_i, Dom\} \models O$ *entailment*

  $\{SO_i, Dom\} \not\models \textbf{\textit{false}}$ *satisfiability*

  $\{SO_i, SO_j, Dom\} \models \textbf{\textit{false}} \quad (i \neq j)$ *disjointness*

  $\{\neg SO_1, \neg SO_2, ..., \neg SO_n, Dom\} \models \neg O$ *domain-completeness*

As an obvious consequence of these correctness conditions, if an obstacle SO_i OR-refines an obstacle O and the latter obstructs a goal G, then SO_i obstructs G as well.

Obstacle analysis aims at anticipating as many likely and critical obstacles as possible, where *critical* means that the consequences of obstacle satisfaction are likely and severe. The expected outcome is a set of countermeasures to such obstacles whose integration in the goal model will result in a more complete and realistic set of software requirements and environment assumptions.

An overall procedure for obstacle analysis looks like this:

For each selected leaf goal in the goal refinement graph (requirement or expectation):
(a) identify as many obstacles to it as possible;
(b) assess their satisfiability, likelihood, and criticality;
(c) resolve the likely and critical ones according to their degree of likelihood and severity.

Obstacle analysis should primarily be focussed on mission-critical goals as the consequences of their obstruction are expected to be more severe. The obstacle trees should be anchored on leaf goals; it appears easier to find ways of breaking finer-grained goals, and leaf goal obstructions get up-propagated in the goal model anyway (see Section 5).

As Fig. 6 shows, goal model elaboration and obstacle analysis are iterative and intertwined processes. The goal-obstacle analysis loop terminates when the remaining obstacles can be tolerated, that is, when they are too unlikely or their consequences are acceptable.

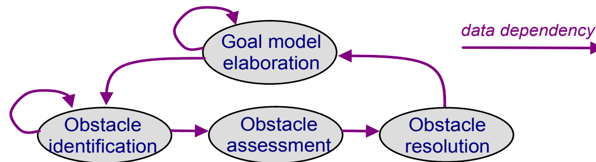

Figure 6. Goal elaboration and risk analysis are iterative and intertwined

The techniques available for supporting steps (a), (b) and (c) will be discussed in Sections 4, 5 and 6, respectively.

4. Obstacle identification

Given a goal G, we are looking for obstruction preconditions O such that :

$$\{O, Dom\} \mid\!- \neg G, \quad Dom \mid\!\not- \neg O$$

To build an obstacle tree anchored on G, we may proceed as follows :

- Negate G;
- Find as many AND/OR refinements of ¬ G as possible in view of properties in *Dom*,
- until reaching obstruction preconditions whose satisfiability, likelihood and criticality can easily be assessed.

As discussed before, the obstacle tree should produce correct refinements of the goal negation; its leaves should, ideally, form a domain-complete set of obstacles.

Various techniques are available for building obstacle trees in a systematic way. Section 4.1 introduces a fairly straightforward principle of tautology-based refinement. Section 4.2 presents a formal regression technique for generating obstacles from domain properties. Section 4.3 shows how formal derivations can be shortcut by use of formal obstruction patterns. Section 4.4 finally overviews a recent technique for generating a domain-complete set of obstacles through a suitable combination of model checking and inductive learning.

4.1. Tautology-based refinement

In view of the negation prefixing root obstacles, propositional tautologies may drive some of the refinements, e.g.,

$\neg (A \wedge B)$ *is equivalent to* $\neg A \vee \neg B$,
$\neg (A \vee B)$ *is equivalent to* $\neg A \wedge \neg B$,
$\neg (A \rightarrow B)$ *is equivalent to* $A \wedge \neg B$,
$\neg (A \leftrightarrow B)$ *is equivalent to* $(A \wedge \neg B) \vee (\neg A \wedge B)$

Fig. 7 illustrates the use of the latter tautology in the context of an automated car handbrake control system. One benefit of tautology-based decompositions is the *complete* OR-refinement they produce when an ∨-connective is introduced in the rewriting of the parent obstacle.

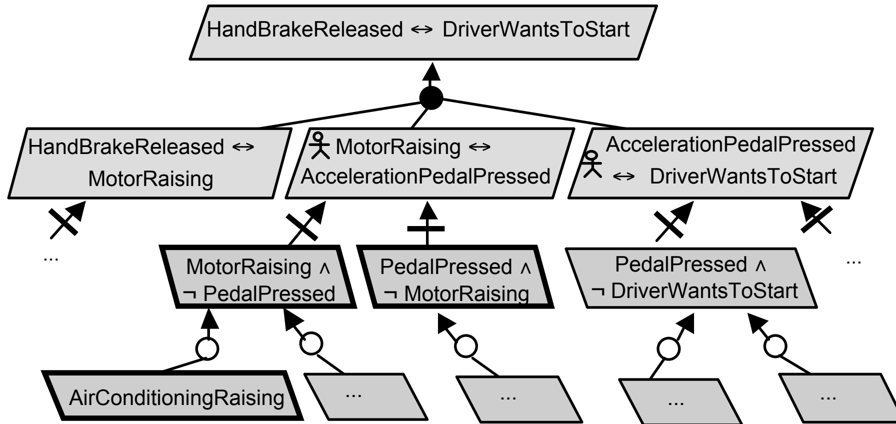

Figure 7. Tautology-based obstacle refinement

4.2. Regressing goal negations through domain properties

Obstacles may also be generated using a formal calculus somewhat corresponding, for declarative statements, to Dijkstra's precondition calculus [Dij76] or Waldinger's regression procedure used in AI planning [Wal77].

The general idea is to regress the goal negation $\neg G$ backwards through matching domain properties in *Dom* and then apply the regression recursively to the results obtained. Assuming domain properties take the general form $A \Rightarrow C$, the procedure is as follows [Lam00].

Initial step:
  take $O := \neg G$
Inductive step:
  let $A \Rightarrow B$ be the domain property selected with B matching some litteral L in O
    whose occurrences are all positive in O [Man92]
  then $\mu := \text{mgu}(L, B)$ (mgu: most generat unifier)
    $O := O\,[L\,/\,A.\,\mu]$

Every iteration of the inductive step produces finer sub-obstacles. A very strong heuristic may be used for guiding the search for matching domain properties. Assume G is an *Achieve* goal of form $C \Rightarrow \Diamond T$. Its negation is thus $\Diamond\,(C \wedge \Box \neg T)$. We may then look for domain properties stating *necessary conditions for the target condition* T, that is, properties of form $T \Rightarrow N$; their equivalent contraposed form $\neg N \Rightarrow \neg T$ provides a match. The heuristic is similar for *Maintain* goals. If such domain properties are not available, we might elicit them.

Let us illustrate the regression procedure on our running example of an ambulance dispatching system. Consider the leaf goal:

Goal *Achieve* [IncidentResolved When AmbulanceOnScene]
  FormalSpec AmbulanceOnScene $\Rightarrow \Diamond$ IncidentResolved

Applying the regression procedure we first negate the goal which yields the root obstacle:

Obstacle IncidentNotResolvedByAmbulance
  FormalSpec $\Diamond$ (AmbulanceOnScene $\wedge\ \Box\ \neg$ IncidentResolved)

We regress this root obstacle through domain properties providing necessary conditions for incident resolution, e.g.,

IncidentResolved $\Rightarrow$ (PersonInjured $\rightarrow$ (NeedsCare $\rightarrow$ CareProvided),
IncidentResolved $\Rightarrow$ (PersonInjured $\rightarrow$ AdmittedAtHospital)

Regressing the root obstacle through these domain properties yields two alternative sub-obstacles:

Obstacle CriticalCareNotGiven
  FormalSpec $\Diamond$ (AmbulanceOnScene
    $\wedge\ \Box$ (PersonInjured $\wedge$ NeedsCare $\wedge\ \neg$ CareProvided))
Obstacle PersonNotAdmittedToHospital
  FormalSpec $\Diamond$ (AmbulanceOnScene $\wedge\ \Box$ (PersonInjured $\wedge\ \neg$ AdmittedAtHospital))

Regressing the first subobstacle *CriticalCareNotGiven* through the domain property

AmbulanceOnScene $\wedge$ PersonInjured $\wedge$ CareProvided $\Rightarrow$ ResourceAvailableInAmbulance

yields the following finer-grained obstacle:

Obstacle ResourceUnavailableInAmbulance
FormalSpec $\Diamond$ (AmbulanceOnScene $\wedge$ PersonInjured $\wedge$ NeedsCare
$\wedge$ $\neg$ ResourceAvailableInAmbulance)

The generation of finer-grained sub-obstacles producing the obstacle tree may go on this way until leaf obstacles that can easily be assessed are reached.

4.3. Using obstruction patterns

Similarly to goal refinement patterns, a catalogue of common goal obstruction patterns may be built and proved correct once for all. The patterns are then reused for obstacle identification by instantiation in matching situations [Lam00]. Fig. 8 shows two such patterns. The pattern on the left encodes the common heuristic used before; the obstacle is obtained by a single regression step through a domain property stating a necessary condition for the goal's target condition. As mentioned before, this pattern can be used for eliciting relevant domain properties as well ("*what are necessary conditions for the goal's target condition?*").

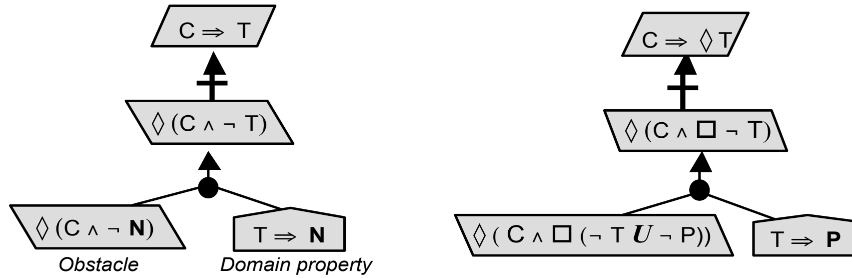

Figure 8. Goal obstruction patterns [Lam09]

The pattern on the right in Fig. 8 encodes a starvation pattern. To understand it, let us consider its instantiation to a generic resource allocation system [Lam09]. A main functional goal in this system is the following:

$\forall$ u: User, r: Resource
Requesting (u, r) $\Rightarrow$ $\Diamond$ Allocated (r, u)

The following known domain property is matching the right leaf node of the starvation pattern:

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

This suggests the following meta-variable instantiations:

C: Requesting (u, r) *T:* Allocated (r, u) *P*: $\neg \exists$ u' $\neq$ u: Allocated (r, u')

The following starvation obstacle is derived by instantiation of the left leaf node:

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

This obstacle captures a feasible coalition among other users that prevents the requesting user from ever getting the resource.

4.4. Combining model checking and inductive learning for obstacle generation

The objective of this technique is more ambitious. Here we want to generate a domain-complete set of obstacles to a given leaf goal whose satisfiability in the domain is guaranteed by construction. The generated obstacle set must thus meet the *obstacle completeness* condition in Section 3. To achieve this, the technique combines model checking for generating goal example and counterexamples traces and then inductive learning for generalizing those traces into obstacle conditions [Alr12].

- For *model checking,* the LTSA tool is applied to goals and domain properties formalized in a LTL variant known as FLTL where atomic formulas are fluents [Gia03, Mag06]. A *fluent* is defined by two sets of events: the intitiating events making it true and the terminating events making it false. Fluents provide a simple, effective means for integrating event-based and state-based specifications. In a pre-processing phase, the tool generates FLTL specifications of goals and domain promerties from LTL ones [Let08].
- For *inductive learning* from example traces, an inductive logic programming (ILP) engine is used to construct generalized properties from examples [Mug94, Cor10]. Given a knowledge base K, a set of positive examples E^+, a set of negative examples E^-, and a set IC of integrity constraints, the learner generates a generalization H such that, for every $E^+$and E^-:

  $\{K, H\} \models E^+$ (*example coverage*)
  $\{K, H\} \not\models E^-$ (*counterexample exclusion*)
  $\{K, H, IC\} \not\models \textbf{\textit{false}}$ (*consistent generalization)*

  In our framework, K will be the domain theory *Dom* whereas the set IC will include constraints on the search for generalizations. The ILP engine is guaranteed to be sound and complete; it scales up for finite domains.

The obstacle generation technique iterates on the following steps until a domain-complete set of obstacles is obtained (see Fig. 9).

1. The working set *BP* of background properties initially includes the set *Dom* of known domain properties. The labelled transition system (LTS) containing all satisfying traces of *BP* is synthesized using LTSA. Let's call it *L(BP).*
2. LTSA is used to model-check the target goal against the synthesized LTS, that is, to check that

   $$L(BP) \models (C \Rightarrow \Theta T)$$

   where C and T are the goal's current and target conditions, respectively, and Θ represents one of the LTL operators introduced in Section 2.2. If the goal is not satisfied, the check produces a counterexample trace E^-.
3. LTSA is used again to model-check the goal's anti-target against the synthesized LTS, that is, it checks that

   $$L(BP) \models (C \Rightarrow \neg \Theta T)$$

   If the anti-target is not satisfied, the check produces a witness trace E^+.
4. The polarity of the example and counterexample is inverted; the goal counterexample trace E^- becomes a witness trace for the obstacle condition we want to obtain whereas the goal witness trace E^+ becomes a counterexample trace for the obstacle condition. The domain properties, goal, counterexample

trace and witness trace are then automatically translated to the logic programming formalism for input to the ILP engine.

5. The ILP engine generates a set of generalizations O_i that covers the obstacle trace E^- while excluding the counterexample trace E^+.
6. The user is asked to select one condition O among the generated O_i. New domain properties *Dom* may also be elicited during this interaction using the aforementioned heuristic of considering domain properties that provide necessary conditions for the considered current target condition.
7. The current set *BP* of background properties is extended as follows:

$$BP := BP \cup \neg O \cup Dom'$$

8. The process is repeated by considering this new *BP* at step 1 until a set of obstacles satisfying the *obstacle completeness* condition

$$\{\neg O_1, ..., \neg O_n, Dom\} \models G$$

is obtained.

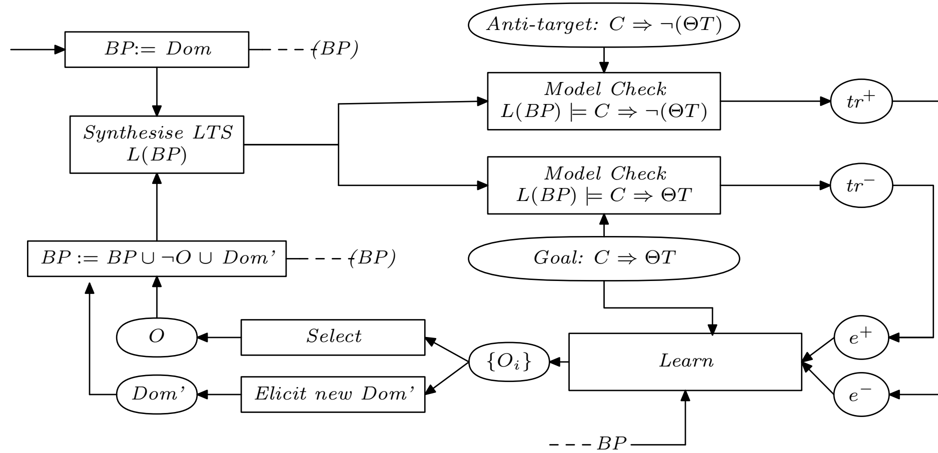

Figure 9. Using model checking and inductive learning for generating a domain-complete set of obstacles [Alr12]

Let us have a closer look at this technique using our running example. Consider the goal

Goal *Achieve* [AllocatedAmbulanceMobilizedByPhone]
FormalSpec Allocated ⇒ o MobilizedByPhone

Step 1. Initially, the set *BP* of background properties contains domain properties such as:

MobilizedByPhone ⇒ CrewResponsive
MobilizedByPhone ⇒ AmbulanceReady

The corresponding fluent definitions are also domain properties in *BP*:

Allocated = <*allocate*, *deallocate*, ***false***>
MobilizedByPhone = <*mobilizePh*, *demobilizePh*, **false**>
CrewResponsive = <crew-responds, crew-ignores, ***true***>
AmbulanceReady = <*setUp*, *leaveStation*, ***true***>

The first fluent definition, for example, states that the fluent *Allocated* is made true by *allocate* events and made false by *deallocate* events (where *allocate* events correspond to

applications of the *Allocate* operation from the operation model); this fluent is initially *false*.

Step 2. The LTL *L(BP)* synthesized from this set of domain properties is model-checked against the goal

$$\text{Allocated} \Rightarrow \circ\ \text{MobilizedByPhone}.$$

This produces the following counterexample trace to the goal:

$$E^- = \ \textit{allocate};\ \textit{crew-ignores}$$

Step 3. The same LTL *L(BP)* is now model-checked against the anti-target

$$\text{Allocated} \Rightarrow \neg \circ\ \text{MobilizedByPhone}.$$

This produces the following witness trace for the goal:

$$E^+ = \ \textit{allocate};\ \textit{mobilizePh}$$

Step 4. After inversion of the polarity of the example and counterexample, the domain properties, goal and example/counterexample are translated to the logic programming formalism.

4.a. A fluent definition such as:

CrewResponsive = <crew-responds, crew-ignores, ***true***>

adds the following facts to the knowledge base *K*:

```
initiates(crew-responds, crewResponsive).
terminates(crew-ignores, crewResponsive).
initially(crewResponsive).
```

4.b. A domain property such as:

MobilizedByPhone ⇒ CrewResponsive

adds the following rule to the integrity constraints *IC*:

```
:- holdsAt(mobilizedByPhone,T,S), not holdsAt(crewResponsive,T,S).
```

4.c. The goal Allocated ⇒ $\circ$ MobilizedByPhone adds the following facts to the knowledge base *K*:

```
holdsAt(mobilizedByPhone,T2,S) :-
   holdsAt(allocated,T1,S), next (T2, T1),
   not obstructed_next (mobilizedByPhone,T1,S)).
```

Note the last predicate added in this translation; it states that the target `mobilizedByPhone` will hold at time *T2* if `allocated` holds at previous time *T1* *provided this target is not obstructed at T1*, that is, provided there is no obstacle that would prevent the ambulance from being mobilized by phone.

4.d. Counterexamples such as *allocate*; *crew-ignores* add the following facts to the knowledge base:

```
happens(allocate, 0, cx).
happens(crew-ignores, 1, cx).
```

while adding the following fact to the set of *positive* obstacle examples:

```
not holdsAt(mobilizedByPhone,2,cx).
```

The latter fact calls for a generalization to be inferred in order to explain why the goal's target is obstructed in this example.

4.e. Witnesses such as *allocate*; *mobilizePh* add the following facts to the knowledge base:

```
happens(allocate, 0, wx).
happens(mobilizePh, 1, wx).
```

while adding the following fact to the set of *negative* obstacle examples:

```
holdsAt(mobilizedByPhone,2,wx).
```

The latter fact requires the inferred generalization to be consistent with the goal's target being not obstructed in this negative example.

Step 5-7. The ILP engine then returns the following learnt rule:

```
obstructed_next (mobilizedByPhone,T,S)) :-
    holdsAt(allocated,T,S),
    not holdsAt(crewResponsive,T,S).
```

which is translated back to the FLTL obstacle specification:

$\Diamond$ (Allocated $\wedge$ $\neg$ CrewResponsive)

whose negation is added to the current set *BP* of background properties.

Step 8. A second iteration on Steps 1-7 produces the following new obstacle:

$\Diamond$ (Allocated $\wedge$ $\neg$ AmbulanceReady)

A next iteration using the domain property

CrewResponsive $\Rightarrow$ PhoneWorking,

either available or acquired at a previous Step 6, produces the following subobstacle:

$\Diamond$ (Allocated $\wedge$ $\neg$ PhoneWorking)

and the process goes on until a domain-complete set of obstacles is obtained.

Note that no user intervention is required for example provision in this incremental obstacle generation process. Moreover, the inferred obstacles are satisfiable in the domain in view of the soundess of the ILP generalization process.

5. Obstacle Assessment

The second phase of obstacle analysis consists of assessing the likelihood and criticality of the generated obstacles (see Section 3), where *criticality* is measured in terms of likelihood and severity of the consequences of obstacle satisfaction. As a consequence of this phase, the next phase of obstacle resolution can be more focussed on likely and critical obstacles.

Obstacle assessment calls for a probabilistic framework in which we can reason about partial goal satisfaction; the degree of goal satisfaction may then be related to the probability of goal obstruction by obstacles.

In passing, we would like to integrate goals that are sometimes stated probabilistically. For example, the ORCON standards for ambulance dispatching systems states that "ambulances shall be on the incident within 14 minutes *in at least 95% of the cases*" [Rep93].

A generalized setting should enable us to capture the severity of obstacle consequences in terms of the difference between the *required* degree of satisfaction of a goal and its *estimated* degree of satisfaction due to likely obstacle obstructions.

This section outlines a probabilistic framework for doing this [Cai12]. Section 5.1 introduces probabilistic goals, their refinements and their possible obstructions by probabilistic obstacles. Section 5.2 shows how the likelihood of obstacles can be evaluated in this generalized setting using the structure provided by obstacle trees. Section 5.3 then shows how the severity of obstacle consequences can be evaluated using the structure provided by the goal model. The prioritization of obstacles by levels of criticality for subsequent resolution is briefly discussed in Section 5.4.

5.1. Probabilistic goals and obstacles

As introduced in Section 2.1, a behavioral goal implicitly defines a maximal set of admissible behaviors. Restricting ourselves to finite-behavior systems, the probability of goal satisfaction is defined in terms of the probability of observing one of those behaviors.

Let us again consider the general form $C \Rightarrow \Theta T$ for a behavioral goal where C and T are the goal's current and target conditions, respectively, and Θ represents one of the LTL operators introduced in Section 2.2. We are obviously interested by the non-vacuous satisfaction of such a goal, leaving aside those behaviors where the goal is trivially satisfied by making C false.

The *probability of non-vacuous satisfaction of a goal* is defined as the ratio of the number of possible system behaviors satisfying its antecedent C and consequent ΘT over the number of possible system behaviors satisfying the condition C. For example, the probability of satisfaction of the goal $Allocated \Rightarrow o\ Mobilized$ in Fig. 1 is the ratio of the number of system behaviors where an allocated ambulance was immediately mobilized over the number of system behaviors where an ambulance was allocated when an incident was reported.

We may sometimes need to check whether some goals are independent or not. Two goals are said to be *dependent* if the set of behaviors non-vacuously satisfying one of them non-vacuously satisfies or denies the other. In terms of conditional probabilities, the independence of goals G_1 and G_2 is characterized by the following conditions:

$$P\ (G_1 \mid G_2) = P\ (G_1 \mid \neg\ G_2) = P\ (G_1), \qquad P\ (G_2 \mid G_1) = P\ (G_2 \mid \neg\ G_1) = P\ (G_2),$$

where $P\,(G)$ denotes the probability of satisfaction of G and $P(G \mid H)$ denotes the probability of satisfaction of G over all behaviors satisfying property H.

In view of the completeness, consistency and minimality conditions on goal refinements in Section 2.3, one can show that (a) in a goal model whose AND-refinements are complete, two goals are dependent if they are connected through a refinement path or a conflict link; and (b) in a minimal, complete and consistent goal refinement, the subgoals are independent.

Two different degrees of goal satisfaction are defined. The *required* degree of satisfaction (RDS) of a goal is the minimal probability of satisfaction admissible for this goal. It is imposed by elicited requirements. Annotating a behavioral goal $C \Rightarrow \Theta T$ in a goal model with its *RDS* amounts to specifying it in a probabilistic temporal logic [Kwi02], e.g., through an assertion such as:

$$C \Rightarrow Pr_{\geq RDS}\ [\Theta T]$$

Note that the standard case of non-probabilistic goals *G* introduced in Section 2 corresponds to $RDS\,(G) = 1$.

On the other hand, the *estimated* probability of satisfaction (EPS) for a goal is the probability of satisfaction of this goal in view of its possible obstructions by obstacles. This probability, denoted by $P\,(G)$ for a goal *G*, will be computed from the goal/obstacle models (see Sections 5.2 and 5.3 hereafter).

The *severity* of violation of a goal *G* is defined by the difference between those two degrees of satisfaction:

$$SV\,(G) = RDS\,(G) - P(G).$$

In this setting the desirable conditions on goal refinements introduced in Section 2.3 may be generalized accordingly [Cai12]:

$P(G \mid Dom) > 0$ (*domain-consistency*)

$P(G \mid SG_1, \ldots, SG_n, Dom) > 0$ (*complete refinement*)

$P(SG_1, \ldots, SG_n \mid Dom) > 0$ (*consistent refinement*)

$P(G \mid SG_1, \ldots, SG_{i-1}, SG_{i+1}, \ldots, SG_n, Dom) < P(G \mid SG_1, \ldots, SG_n, Dom)$ (*minimal refinement*)

A goal is partially satisfied because of obstacles that might obstruct it. The *probability of an obstacle* is the probability of satisfaction of the obstacle condition, that is, the ratio of the number of possible system behaviors satisfying the obstacle condition over the number of possible system behaviors. For example, the probability of the obstacle $\Diamond$ *(Allocated* $\wedge$ $\neg$ *CrewResponsive)* is the ratio of the number of possible behaviors where the crew of an allocated ambulance is not responsive over the number of possible system behaviors.

The conditions on obstacle trees introduced in Section 3 are now generalized to probabilistic obstacles, e.g.,

$P(\neg G \mid O, Dom) > 0$ (*potential obstruction*)

$P(O \mid Dom) > 0$ (*satisfiability in domain*)

$P(O \mid SO_i) > 0$ for all SO_i (*entailment*)

$P(O \mid \neg SO_1, \ldots, \neg SO_n, Dom) = 0$ (*domain completeness*)

The *disjointness* condition on sub-obstacles in Section 2.2 is generalized into an *independence* condition, namely,

$$P(SO_i \mid SO_j) = P(SO_i \mid \neg SO_j) = P(SO_i)$$
$$P(SO_j \mid SO_i) = P(SO_j \mid \neg SO_i) = P(SO_j)$$

Note that two dependent obstacles can be captured through three independent ones each having a specific probability: one where the first obstacle condition holds but not the second, one where the second obstacle condition holds but not the first, and one where both hold.

5.2. Assessing the likelihood of obstacles

To assess our generated obstacles, we first need to evaluate how likely these obstacles are. To achieve this in our probabilistic framework, we need to:

- estimate the probabilities of the leaf obstacles in the obstacle refinement trees we have built,
- up-propagate these probabilities through the trees until the probability of the root obstacles are obtained.

To get probability estimates for the leaf obstacles, we need to rely on domain expertise or use statistical data about past system behaviors (cf. the definition of a probabilistic obstacle). For example, consider the leaf obstacle:

$$\Diamond\,(\text{AmbulanceMobilized} \wedge \Box\,\neg\,\text{GPSWorking})$$

in Fig. 5. Statistical data might tell us that the situation of a non-working GPS inside a mobilized ambulance occurs in 10% of the cases.

To up-propagate probabilities in an obstacle refinement tree, we may use propagation equations such as the following [Cai12]:

- for an AND-refinement:

$$P(O) = P(SO_1) \times P(SO_2) \times P(O \mid SO_1, SO_2)$$

- for an OR-refinement:

$$P(O) = 1 - (1 - P(SO_1) \times P(O \mid SO_1)) \times (1 - P(SO_2) \times P(O \mid SO_2))$$

The preceding equations are recursively applied bottom-up through the refinement tree until reaching the probability of the root obstacle, that is, the goal negation.

For example, let us assume that the following leaf estimates were obtained from statistical data (see Fig. 5):

P (Ambulance**Not**InFamiliarArea) = 0.2, P (GPS**Not**Working) = 0.1,

P (AmbulanceLost | Ambulance**Not**InFamiliarArea, GPS**Not**Working) = 0.95;

we get by the AND-propagation equation:

P (AmbulanceLost) = 0.019.

Assuming now the following leaf estimates available from statistical data (see Fig. 5):

P (AmbulanceStuckInTrafficJam) = 0.02, P (AmbulanceBrokenDown) = 0.005,

P (Ambulance**Not**OnSceneInTime | AmbulanceLost) = 0.99,

P (Ambulance**Not**OnSceneInTime | AmbulanceStuckInTrafficJam) = 0.98,

P (Ambulance**Not**OnSceneInTime | AmbulanceBrokenDown) = 1,

we obtain by the OR-propagation equation the following probability for the root obstacle in Fig. 5:

P (MobilizedAmbulance**Not**OnSceneInTime) = 0.0429

which means that a mobilized ambulance will not arrive on the incident scene in time in 4.3% of the cases.

5.3. Assessing the severity of obstacle consequences

Once we have evaluated how likely our generated obstacles are, we need to evaluate how likely and severe their consequences are. In risk analysis, the consequence of a risk is expressed in terms of the degree of loss of satisfaction of the associated objective. This is translated in our framework by defining the consequences of an obstacle as the lower degree of satisfaction of the obstructed leaf goal and, recursively, of its parent and ancestor goals in the goal model.

The propagation of the propability of a root obstacle *RO* to the obstructed leaf goal *LG* is fairly straightforward; we just need to use the following equation:

$$1 - P(LG) = P(RO) \times P(\neg LG \mid RO)$$

In our preceding example, we will thereby obtain for the obstructed leaf goal in Fig. 5:

P (AmbulanceOnSceneWhenMobilized) = 0.957

The decreased degree of satisfaction of the obstructed leaf goal must then be up-propagated in the goal refinement graph in order to determine all obstacle consequences [Cai12]. In case of a single system with no alternative OR-refinements and with complete AND-refinements, the general up-propagation equation reduces to:

$$P(G) = P(SG_1, SG_2) + P(SG_1, \neg SG_2) \times P(G \mid SG_1, \neg SG_2) + P(SG_2, \neg SG_1) \times P(G \mid SG_2, \neg SG_1) \quad (\textit{AND}\text{-propagation})$$

This equation may be further simplified for common refinement patterns such as those introduced in Section 2.3 (see Figs. 2-3); their completeness, consistency and minimality make the subgoals independent. For example,

$$P(G) = P(SG_1) \times P(SG_2) \quad (\textit{milestone-driven}\text{ refinement})$$

$$P(G) = P(CS) \times P(SG_1) + (1 - P(CS)) \times P(SG_2) \quad (\textit{case-driven} \text{ refinement with condition } CS)$$

Two kinds of impact analysis may be performed to evaluate the consequences of obstacles [Cai12].

- *Global impact analysis*: the computed probabilities for all obstructed leaf goals are together up-propagated in the goal graph to see how much the resulting *EPS* of higher-level goals deviates from their required *RDS*.
- *Local impact analysis*: the consequence of a single leaf goal obstruction is evaluated by up-propagation of the computed probability for this leaf goal, all other leaf goals being assigned a probability of 1.

Let us briefly illustrate a global impact analysis for the model in Fig. 1. The EPS computed in Section 5.2 for the leaf goal *AmbulanceOnSceneWhenMobilized* was 0.956. Replaying similar up-propagations through the obstacle trees anchored on the other leaf goals in Fig. 1 yields the following EPS for these leaf goals:

P (AmbulanceMobilizedByFax) = 0.90, P (AmbulanceMobilizedByPhone) = 0.95,
P (AllocatedAmbulanceMobilizedWhenOnRoad) = 0.98,
P (AmbulanceAllocatedWhenIncidentReported) = 0.98.

Using the *AND*-propagation equation above for the bottom AND-refinement, the simplified case-driven one for the next higher-level refinement, and the simplified milestone-driven one for the next two higher-level refinements, we obtain for the root goal in Fig. 1:

P (AmbulanceOnSceneInTime WhenIncidentReported) = 0.928

This means that the system as modelled is not able to satisfy the ORCON standard; the latter requires this root goal to be satisfied in at least 95% of the cases. The violation severity *SV* for this goal is 2.2%.

5.4. Prioritizing obstacles

In view of the possibly large number of potential obstacles to a goal model, we want to focus the resolution process on the most problematic leaf obstacles.

This is a multi-criteria optimization problem as we need to find minimal sets of leaf obstacles that maximize the severity *SV* of goal violations.

A brute-force approach includes the following steps:

- generate all leaf obstacle combinations,
- compute the *SV(G)* for each obstructed goal *G*, possibly weighted by the priority level annotating *G* in the goal model (see Section 2.1),
- sort the leaf obstacle combinations by severity.

Alternatively, we may use optimized techniques that are available for generating Pareto fronts [Kun75].

In our running example, the brute force approach highlights 4 obstacle combinations resulting in

SV (AmbulanceOnSceneInTime WhenIncidentReported) > 0

(out of 8 feasible combinations). No single obstacle is sufficient to get below the *RDS* of 95%. Two combined obstacles are however sufficient. In particular, the possibility of an ambulance being lost and stuck in traffic jam is sufficient for severe obstruction of the goal (with $SV = 1.8$); this is the pair to resolve first.

6. Obstacle Resolution for a More Complete Goal Model

The final phase in *identify-assess-control* cycles of obstacle analysis consists in resolving critical obstacles, that is, those sufficiently likely and with sufficiently severe consequences to fall below *RDS* thresholds. Obstacle resolution consists in identifying suitable countermeasures and deploying them.

Two strategies may be followed for this.

- *RE-time resolution*: We may (a) explore alternative countermeasures by application of model transformation operators encoding obstacle resolution tactics [Lam00]; (b) select a "best" resolution based on obstacle severity, on estimates of risk-reduction leverage [Lam09] or on contribution to soft goals from the goal model [Chu00]; and (c) integrate the selected countermeasure in the goal model. This may call for further refinement of the new or transformed goals thereby introduced.
- *Run-time resolution*: We may tolerate obstacles that are not too problematic and defer their resolution at runtime. This requires mechanisms for run-time monitoring of obstacle occurrences and dynamic system reconfiguration when repeated obstacle occurrences become intolerable [Fea78].

This section overviews a variety of obstacle resolution tactics and their corresponding model transformation operators [Lam00, Lam09]. Fig. 10 shows some of them.

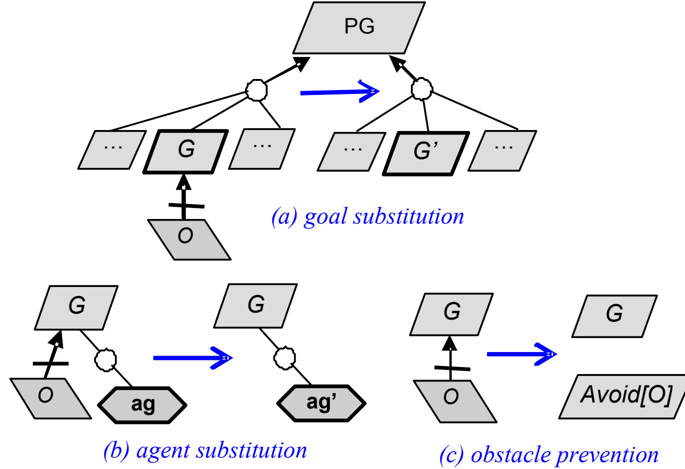

Figure 10. Model transformation operators for obstacle resolution [Lam09]

Goal substitution*:* This tactic consists of considering an alternative refinement of the parent goal to avoid the obstruction of one of the child goals. The new alternative should be less exposed to obstruction risks.

As an illustration, consider the obstacle *AmbulanceCrewNotInFamiliarArea* that obstructs the leaf goal *AmbulanceOnSceneWhenMobilized* in Fig. 5 up to the root goal *AmbulanceOnSceneInTimeWhenIncidentReported* in Fig. 1. We might then replace the subgoal *AmbulanceAllocatedWhenIncidentReported* in Fig. 1 by the alternative subgoal *DivisionalAmbulanceAllocatedWhenIncidentReported* to ensure that ambulances are allocated to incidents within the same geographic division only.

Agent substitution: This tactic consists of considering an alternative responsibility assignment for the obstructed goal so as to make the obstacle unsatisfiable in the domain.

For example, the obstacle $\lozenge$ *(Allocated* $\wedge\neg$ *AmbulanceReady)* generated in Section 4.4 might be resolved by assigning the leaf goal *AmbulanceAllocatedWhenIncidentReported* in Fig. 1 to ambulance crew instead of assigning it to the software-to-be.

Obstacle prevention: A new goal *Avoid [obstacle]* is added to the goal model in order to prevent the obstacle from occurring. This goal must in general be further refined.

For example, consider the obstacle *MobilizationOrderTakenByOtherAmbulance* that obstructs the goal the AmbulanceMobilizedByFax. To resolve it, we might introduce a new goal *Avoid [*AmbulanceMobilizedWithoutOrder*]* .

Goal weakening: This tactic consists of weakening the obstructed goal so that the weaker version is no longer obstructed. For a goal $C \Rightarrow \Theta T$, the weakening may be achieved by adding a conjunct in C or a disjunct in T.

This tactic is among the most frequently used ones. For example, the overideal goal *AmbulanceOnSceneWhenMobilized* might be replaced by the weaker version *AmbulanceOnSceneWhenMobilized**And**NotBrokenDown.*

Obstacle reduction: This tactic consists of reducing the probability of the obstacle by some ad hoc countermeasure.

For example, we might try to reduce ambulance crew practice leading to obstacle satisfaction by some dedicated reward/dissuasion system.

Goal restoration: Here we tolerate the obstacle but enforce the target condition of the obstructed goal when the obstacle occurs. This corresponds to adding a new goal *Obstacle* $\Rightarrow \lozenge$ *TargetCondition* in the goal model.

For example, the leaf goal *AmbulanceOnSceneWhenMobilized* in Fig. 5 might be restored by mobilization of another ambulance nearer to the incident scene if the mobilized one gets lost.

Obstacle mitigation: Here again we tolerate the obstacle but introduce a new goal to mitigate the consequences of the obstacle. In a *weak mitigation*, the new goal ensures a weaker version of the goal when the latter is obstructed. In a *strong mitigation*, the new goal ensures a parent of the goal when the latter is obstructed.

For example, the preceding obstacle *MobilizationOrderTakenByOtherAmbulance* might be mitigated by introduction of a new goal *MobilizationByOtherAmbulanceKnown*.

Domain transformation: This tactic consists of transforming the domain and its descriptive properties so that the obstruction disappears. The new or modified domain properties make the obstacle unsatisfiable or not obstructing the goal anymore.

As an illustration of the former case, consider the goal *AmbulanceMobilizedByFax*. One obstacle to it 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 be eliminated by transforming the mobilization order so that it no longer mentions the incident location, the latter being provided by a mobile data terminal inside the ambulance.

7. Conclusion

It is important to verify that software applications implement their specifications correctly. Do these specifications meet the software requirements? Do these requirements meet the system goals under realistic assumptions? Are these goals, requirements, and assumptions complete, adequate, and consistent? These are critical questions with many challenging issues for formal methods.

Rich models are essential to support the requirements engineering (RE) process. Such models must address multiple perspectives including the intentional, structural, responsibility, operational, and behavioral facets of the system. They must cover the entire system, comprising both the software and its environment – made of humans, devices, pre-existing software, mother Nature, attackers, etc. Rich RE models should make alternative options explicit – such as alternative goal refinements, alternative agent assignments, or alternative countermeasures to critical risks. They should support a seamless transition from high-level concerns to operational requirements.

Building such models is hard and critical. We should therefore be guided by methods that are systematic, incremental, and support the analysis of partial models.

Goal-based reasoning is pivotal for model building and requirements elaboration, for exploration and evaluation of alternatives, and for anticipation of incidental or malicious behaviors.

Requirements completeness is a key issue. It can be achieved through multiple means such as refinement checking, to find out missing subgoals, and obstacle analysis to identify, assess, and control risks through countermeasure goals that were missing.

Declarative specifications play an important role in the RE process – in particular, for communicating with stakeholders and for early reasoning about partial models.

In order to engineer highly reliable and secure systems, it is essential to start thinking methodically about these aspects as early as possible, that is, at requirements engineering time. We must be pessimistic from the beginning about the software and about its environment, and anticipate all kinds of risks including hazards, threats and other unexpected agent behaviors.

The use of the various risk analysis techniques presented in the paper results in increased requirements completeness under unexpected but likely conditions. The techniques illustrated the benefits of a "multi-button" framework where semi-formal techniques are used for modeling, navigation, and traceability whereas formal techniques are used, when and where needed, for precise, incremental reasoning on mission-critical model portions. As suggested in this overview paper, goal-oriented models offer a lot of opportunities for formal methods.

Acknowledgement. Many of the ideas presented in this paper were developed jointly with Emmanuel Letier, Antoine Cailliau, Robert Darimont, Christophe Damas, Bernard Lambeau, Philippe Massonet, Christophe Ponsard, André Rifaut, Hung Tran Van, and with Dalal Alrajeh, Jeff Kramer, Alessandra Russo and Sebastian Uchitel at Imperial College. Warmest thanks to them all!

8. References

[Alr12] D. Alrajeh, J. Kramer, A. van Lamsweerde, A. Russo and S. Uchitel, "Generating Obstacle Conditions for Requirements Completeness", *Proc. ICSE'2012: 34th Intl. Conf. on Software Engineering*, Zürich, May 2012.

[Amo94] E.J. Amoroso, *Fundamentals of Computer Security*. Prentice Hall, 1994.

[As11] Y. Asnar, P. Giorgini and John Mylopoulos, "Goal-driven Risk Assessment in Requirements Engineering", *Req. Eng. Journal* 16(2), June 2011, 101-116.

[Cai12] A. Cailliau and A. van Lamsweerde, "A Probabilistic Framework for Goal-Oriented Risk Analysis", *Proc. RE'2012: IEEE Intl. Conf. on Requirements Engineering*, Chicago, Sept. 2012.

[Chu00] L. Chung, B. Nixon, E. Yu and J. Mylopoulos, Non-functional requirements in software engineering. Kluwer Academic, Boston, 2000.

[Cor10] D. Corapi, A. Russo, and E. Lupu, "Inductive logic programming as abductive search", *Tech. Comm. of 26th Intl. Conf. on Logic Programming*, Vol. 7 of LIPIcs, 2010, 54–63.

[Dar93] A. Dardenne, A. van Lamsweerde and S. Fickas, "Goal-Directed Requirements Acquisition", Science of Computer Programming, Vol. 20, 1993, 3-50.

[Dar96] R. Darimont and A. van Lamsweerde, Formal Refinement Patterns for Goal-Driven Requirements Elaboration. Proceedings FSE-4 - Fourth ACM Conference on the Foundations of Software Engineering, San Francisco, October 1996, 179-190.

[Dar07] R. Darimont and M. Lemoine, "Security Requirements for Civil Aviation with UML and Goal Orientation", *Proc. REFSQ'07 – International Working Conference on Foundations for Software Quality*, Trondheim (Norway), LNCS 4542, Springer-Verlag, 2007.

[Dij76] E.W. Dijkstra, *A Discipline of Programming*. Prentice-Hall, 1976.

[Fea98] M. Feather, S. Fickas, A. van Lamsweerde, and C. Ponsard, "Reconciling System Requirements and Runtime Behaviour", *Proc. IWSSD'98 - 9th International Workshop on Software Specification and Design*, Isobe, IEEE CS Press, April 1998.

[Fea03] M.S. Feather and S.L. Cornford, "Quantitative Risk-Based Requirements Reasoning", *Requirements Engineering Journal* Vol. 8 No.4, 2003, 248-265.

[Gia03] D. Giannakopoulou and J. Magee, "Fluent Model Checking for Event-Based Systems", Proc. ESEC/FSE 2003, 10th European Software Engineering Conference, Helsinki, 2003.

[Kun75] H. T. Kung, F. Luccio and F. P. Preparata, "On Finding the Maxima of a Set of Vectors", *J. ACM* Vol. 22 No. 4, Oct. 1975, 469-476.

[Kwi02] M. Kwiatkowska, G. Norman and D. Parker, "Probabilistic Symbolic Model Checking with PRISM: A Hybrid Approach", *Proc. TACAS'02*, LNCS 2280, Springer-Verlag, April 2002, 52-66.

[Lam00] A. van Lamsweerde and Emmanuel Letier, "Handling Obstacles in Goal-Oriented Requirements Engineering", *IEEE Transactions on Software Engineering* 26(10), October 2000, 978-1005.

[Lam01] A. van Lamsweerde , "Goal-Oriented Requirements Engineering: A Guided Tour", *Proc. RE'01 - 5th IEEE Intl. Symp. Requirements Engineering*, Toronto, Aug. 2001, 249-263.

[Lam04a] A. van Lamsweerde, "Elaborating Security Requirements by Construction of Intentional Anti-Models", *Proc. ICSE'04, 26th Intl. Conf. on Software Engineering*, ACM-IEEE, May 2004, 148-157.

[Lam04b] A. van Lamsweerde, "Goal-Oriented Requirements Engineering: A Roundtrip from Research to Practice", Invited Keynote Paper, *Proc. RE'04, 12th IEEE Joint International Requirements Engineering Conference*, Kyoto, Sept. 2004, 4-8.

[Lam09] A. van Lamsweerde, *Requirements Engineering: From System Goals to UML Models to Software Specifications*, Wiley, January 2009.

[Let02a] E. Letier and A. van Lamsweerde, “Agent-Based Tactics for Goal-Oriented Requirements Elaboration”, *Proceedings ICSE'2002 - 24th International Conference on Software Engineering*, Orlando, May 2002, 83-93.

[Let02b] E. Letier and A. van Lamsweerde, “Deriving Operational Software Specifications from System Goals”, *Proc. FSE’10: 10th ACM Symp. Foundations of Software Engineering,* Charleston, Nov. 2002.

[Let04] E. Letier and A. van Lamsweerde, “Reasoning about Partial Goal Satisfaction for Requirements and Design Engineering”, *Proc. FSE’04, 12th ACM International Symp. on the Foundations of Software Engineering*, Newport Beach (CA), Nov. 2004, 53-62.

[Let08] E. Letier, J. Kramer, J. Magee, and S. Uchitel, “Deriving event-based transitions systems from goal-oriented requirements models”, *J. Automated Software Engineering* Vol. 15 No. 2, 2008, 175–206.

[Lev95] N.G. Leveson, *Safeware: System Safety and Computers*. Addison-Wesley, 1995.

[Lev02] N.G. Leveson, “An Approach to Designing Safe Embedded Software”, *Proc. EMSOFT 2002 – Embedded Software: 2nd International Conference*, Grenoble, LNCS 2491, Springer-Verlag, Oct. 2002, 15-29

[Lun11] M.S. Lund, B. Solhaug and K. Stølen, *Model-Driven Risk Analysis: the CORAS approach*. Springer-Verlag, 2011.

[Lut93] R.R. Lutz, "Analyzing Software Requirements Errors in Safety-Critical, Embedded Systems", Proceedings RE'93 - First International Symposium on Requirements Engineering, San Diego, IEEE, 1993, 126-133.

[Lut07] R. Lutz, A. Patterson-Hine, S. Nelson, C.R. Frost, D. Tal and R. Harris, “Using Obstacle Analysis to Identify Contingency Requirements on an Unpiloted Aerial Vehicle”, *Requirements Engineering Journal* Vol. 12 No. 1, 2007, 41-54.

[Mag06] J. Magee and J Kramer, Concurrency – State Models & Java Programs. Second edition, Wiley, 2006.

[Man92] Z. Manna and A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems,* Springer-Verlag, 1992.

[Mug94] S.H. Muggleton and L. De Raedt, “Inductive Logic Programming: Theory and Methods”, *J. of Logic Programming*, Vol. 19/20 No.1, 1994, 629–679;

[Obj04] The Objectiver Toolset, http://www.objectiver.com.

[Pon07] Ch. Ponsard, P. Massonet, A. Rifaut, J.F. Molderez, A. van Lamsweerde, H. Tran Van, “Early Verification and Validation of Mission-Critical Systems”, Formal Methods in System Design Vol. 30 No. 3, 2007, 233-247.

[Rep93] *Report of the Inquiry Into the London Ambulance Service*. February 1993. The Communications Directorate, South West Thames Regional Authority.

[Tra04] H. Tran Van, A. van Lamsweerde, P. Massonet, Ch. Ponsard, “Goal-Oriented Requirements Animation”, *Proc. RE’04, 12th IEEE Joint International Requirements Engineering Conference*, Kyoto, Sept. 2004, 218-228.

[Wal77] R. Waldinger, “Achieving Several Goals Simultaneously”, in *Machine Intelligence*, Vol. 8, E. Elcock and D. Michie (Eds.), Ellis Horwood, 1977.