Integrating Exception Handling in Goal Models

Antoine Cailliau and Axel van Lamsweerde

ICTEAM – Institute for Information & Communication Technologies, Electronics and Applied Mathematics
Université catholique de Louvain
Louvain-la-Neuve, Belgium
{antoine.cailliau, axel.vanlamsweerde}@uclouvain.be

***Abstract*—Missing requirements are known to be among the major sources of software failure. Incompleteness often results from poor anticipation of what could go wrong with an over-ideal system. Obstacle analysis is a model-based, goal-anchored form of risk analysis aimed at identifying, assessing and resolving exceptional conditions that may obstruct the behavioral goals of the target system. The obstacle resolution step is obviously crucial as it should result in more adequate and more complete requirements. In contrast with obstacle identification and assessment, however, this step has little support beyond a palette of resolution operators encoding tactics for producing isolated countermeasures to single risks. In particular, there is no single clue to date as to where and how such countermeasures should be integrated within a more robust goal model.**

To address this problem, the paper describes a systematic technique for integrating obstacle resolutions as countermeasure goals into goal models. The technique is shown to guarantee progress towards a complete goal model; it preserves the correctness of refinements in the overall model; and keeps the original, ideal model visible to avoid cluttering the latter with a combinatorial blow-up of exceptional cases. To allow for this, the goal specification language is slightly extended in order to capture exceptions to goals seperately and distinguish normal situations from exceptional ones. The proposed technique is evaluated on a non-trivial ambulance dispatching system.

***Index Terms*—Obstacle analysis, goal modeling, probabilistic goals, risk control, requirements completeness, exception handling, goal-oriented requirements engineering, quantitative reasoning.**

I. INTRODUCTION

Requirements-related errors are commonly recognized to be the most frequent, persistent, expensive and dangerous types of software errors [13]. Among these, missing requirements tend to be the worst. They often arise from a natural inclination to believe that the software and its environment will always behave as expected; no requirements are engineered for cases where this optimistic assumption does not hold. Requirements completeness therefore calls for putting risk analysis at the heart of the RE process [2, 3, 5, 8, 13, 14, 20].

A *risk* is an uncertain factor whose occurrence may result in the loss of satisfaction of some high-level objective [4, 8, 13]. A risk has a probability of occurrence and one or multiple consequences. Each consequence has a severity in degree of loss of satisfaction of the corresponding objective [5, 8]. Risks may cover undesirable situations such as safety hazards [16, 18], security threats [12, 26] or data inaccuracies [14] dependent on the type of objective they negatively impact on.

At requirements engineering time, risks should be identified, assessed in terms of their likelihood and criticality, and controlled through effective countermeasures [13]. Obstacle analysis has been introduced and used as a model-based, goal-oriented form of risk analysis [2, 6, 14, 21]. An *obstacle* to a goal is a precondition for non-satisfaction of this goal. *Obstacle analysis* consists of (a) *identifying* obstacles from available goals, assumptions and domain properties; (b) *assessing* their likelihood and criticality in terms of severity of their consequences; and (c) *resolving* likely and critical obstacles through countermeasures to be incorporated into the goal model.

To support obstacle analysis, techniques are available for *identifying* obstacles systematically from goals and domain properties [1, 14]. For obstacle *assessment*, likelihoods and criticalities may be determined quantitatively by calculations over obstacle refinement trees and goal refinement trees, respectively; such calculations call for probabilistic extensions to cope with probabilistic goals and obstacles [5, 25]. For obstacle *resolution*, operators encoding risk control tactics were proposed to explore alternative resolutions –such as *avoid obstacle*, *reduce obstacle likelihood*, *mitigate obstacle*, *weaken goal*, *substitute goal*, *restore goal*, or *substitute agent* [14].

The obstacle resolution step is obviously crucial; it directly impacts the adequacy, completeness and robustness of the goal model. However, little support is currently available for this step beyond the above resolution operators for countermeasure exploration. In particular, it is totally unclear where and how selected countermeasures produced by such operators should be integrated in the goal model to increase its completeness and robustness.

To address this problem, the paper describes techniques for integrating obstacle resolutions systematically in the goal refinement graph while propagating the resulting changes wherever required in the model. These techniques guarantee that:

- the model is increasingly robust and complete as resolutions are being integrated;
- the normal system behaviors and those not affected by the obstacles are preserved;
- the correctness of goal refinements in the model is preserved.

A goal model integrating countermeasures to obstacles may need to be restructured so as to keep the goals refering to normal situations separate from the countermeasure goals refering to exceptional situations. There are multiple reasons for this.

- For higher readability and better visibility, the ideal model containing all functional and non-functional goals in normal situations should be kept visible. The

specification of these goals should not be cluttered with items refering to exceptional situations.

- The model structuring and specification should not exhibit any combinatorial blow-up of exceptional cases. Without any structuring mechanism, the integration of multiple countermeasures to multiple risks considered in combination might produce a large number of cases.
- Exceptional situations should be identifiable and integrated *incrementally*. The handling of each single situation should be isolated from the others.
- The traceability of exceptional cases should be supported from requirements to architecture. Keeping exceptions separate from each other and from the goals in normal situations enables traceability from the goal model and its operationalization on the one hand and exception handlers in the architecture on the other hand.
- In case of obstacle tolerance with no countermeasure integrated in the model, a one-to-one mapping should be maintained between (a) the obstacle and countermeasure identified at modeling time, and (b) the corresponding runtime monitor/adaptator mechanisms for dynamic reconfiguration when the obstacle is too frequent [9].

To support such separation between normal and exceptional situations, the paper extends the goal language with semantics-preserving constructs for specifying exceptions and their "handlers" –that is, the countermeasures associated with them. Model transformation operators are then provided for attaching and detaching exceptions to/from associated goals in the goal model.

The paper is organized as follows. Section II introduces some necessary background on modeling goals and their obstacles. Section III motivates our technique on a small example. Section IV more precisely specifies the problem to be solved by our integration approach. Section V describes the conditions and mechanisms for integrating countermeasures in a goal model. Section VI presents the constructs for specifying goals with exceptions together with their semantics. Section VII introduces model transformation operators for attaching and detaching exceptions to/from goals. Section VIII summarizes the evaluation of our proposals on an ambulance dispatching system. Section IX briefly discusses related work.

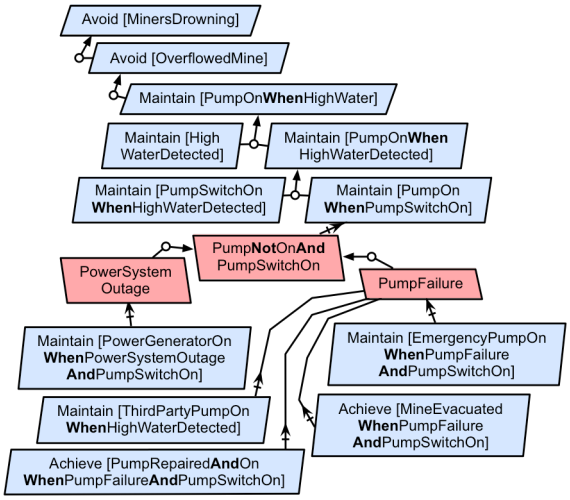


Fig. 1. Partial goal model for a mine pump system

II. BACKGROUND

This section recalls some basics on behavioral goal modeling, obstacle analysis and probabilistic goals while introducing our running example.

Goal-oriented system modeling. A *goal* is a prescriptive statement to be satisfied by cooperating agents forming the considered system. The latter may include devices such as sensors and actuators, people, preexisting software, and the software to be developed. *Domain properties* are descriptive statements about the problem space, e.g., physical laws.

Behavioral goals prescribe maximal sets of desired system behaviors; unlike soft goals they are satisfiable in a clear-cut sense [13]. A *behavior* is a sequence of system state transitions. Linear Temporal Logic (LTL) may be used to specify behavioral goals formally and enable formal analyses [13, 22]. The goals then have the general form:

$$C \Rightarrow \Theta T,$$

where Θ represents an LTL operator such as: $\bigcirc$ (in the next state), $\Diamond$ (sometimes in the future), $\Diamond_{\leq d}$ (sometimes in the future before deadline d), $\Box$ (always in the future), $\Box_{\leq d}$ (always in the future up to deadline d), W (always in the future unless), U (always in the future until), and where $P \Rightarrow Q$ denotes $\Box(P \rightarrow Q)$. The following standard logical connectives are used: $\wedge$ (and), $\vee$ (or), $\neg$ (not), $\rightarrow$ (implies), $\leftrightarrow$ (equivalent).

Among behavioral goals, *Achieve* goals follow the specification pattern "if C then sooner-or-later T", that is, $C \Rightarrow \Diamond T$, where C and T denote a current and target condition, respectively. *Maintain* goals follow the specification pattern "if C then always G", that is, $C \Rightarrow \Box G$ where G denotes a good condition. *Avoid* goals follow a similar pattern "if C then never B", that is, $C \Rightarrow \Box\neg B$ where B denotes a bad condition.

A *goal model* is an AND/OR graph showing how goals contribute positively or negatively to each other. Parent goals are obtained by abstraction, e.g., through *why* questions. Subgoals are obtained by refinement, e.g., through *how* questions. In a goal refinement graph, leaf goals are *requirements* or *assumptions* dependent on whether they are assigned to single software-to-be or environment agents, respectively.

Fig. 1 shows a partial goal model for a mine pump system [11, 13, 15]. Refinement patterns help building such a model through common goal decomposition tactics such as *Milestone-Driven*, *Case-Driven*, *Guard-Introduction*, *Divide-And-Conquer*, *Uncontrollability-Driven*, etc. [7, 13]. For example, consider the following goal in Fig. 1:

Goal Maintain [PumpOn**When**HighWater]
FormalSpec ∀ p:Pump, s:Sump
s.WaterLevel = "High" ∧ PumpInSump(p, s)
⇒ p.Motor = "On"

Applying the *Unmonitorability-driven* refinement pattern to this goal yields, after instantiation, the following refinement where the antecedent of the second subgoal becomes monitorable by the software pump controller:

Goal Maintain [HighWaterDetected]
FormalSpec ∀ p:Pump, s:Sump, c:PumpController
s.WaterLevel = "High" ∧ PumpInSump(p, s) ∧ CtrlPump(c, p)
⇒ c.HighWaterSignal = "On"
Goal Maintain [PumpOn**When**HighWaterDetected]
FormalSpec ∀ p:Pump, c:PumpController
c.HighWaterSignal = "On"∧ CtrlPump(c, p)
⇒ p.Motor = "On"

Refinement patterns produce goal refinements guaranteed to be complete, consistent and minimal. A refinement is *complete* when the subgoals SG_i, possibly with domain properties in *Dom*, are sufficient for satisfying the parent goal *PG*:

$\{SG_1, \ldots, SG_n, Dom\} \models PG$ (complete refinement)

A refinement is *consistent* if:

$\{SG_1, \ldots, SG_n, Dom\} \not\models false$ (consistent refinement)

A refinement is *minimal* if all subgoals are needed for satisfaction of the parent goal:

for all $1 < i < n$: $\{SG_1, \ldots, SG_{i-1}, SG_{i+1}, \ldots SG_n, Dom\} \not\models PG$

The two lower AND-refinements in the upper part of Fig. 1 are complete, consistent and minimal.

Obstacle analysis. An *obstacle* to a goal is a satisfiable precondition for not satisfying this goal [14]:

$\{O, Dom\} \models \neg G$ (obstruction)

$\{O, Dom\} \not\models false$ (domain consistency)

The obstacles to a goal may similarly be organized as an AND/OR refinement tree. The root obstacle is the negation of the obstructed goal. An AND-refinement captures a combination of subobstacles to satisfy the parent obstacle. An OR-refinement captures alternative ways of satisfying the parent. Each OR-refinement SO_i must entail the parent obstacle *PO*:

for all i : $\{SO_i, Dom\} \models PO$ (entailment)

OR-refinements should be domain-complete and disjoint:

$\{\neg SO_1, \ldots, \neg SO_n\} \models \neg PO$ (domain completeness)

for all $i \neq j$: $\{SO_i, SO_j, Dom\} \models false$ (disjointness)

Leaf obstacles are fine-grained obstacles whose satisfiability and likelihood can be more easily estimated by experts.

A variety of formal techniques, obstruction patterns and heuristic rules are available for systematic obstacle identification [1, 13, 14]. For example, consider the following right leaf goal in Fig. 1:

Goal Maintain [PumpOn**When**PumpSwitchOn]
FormalSpec $\forall$p: Pump
p.Switch = "On" $\Rightarrow$ p.Motor = "On"

Its negation yields the following root obstacle:

Obstacle Pump**Not**On**And**PumpSwitchOn
FormalSpec $\Diamond \exists$p: Pump
p.Switch = "On" $\wedge$ p.Motor $\neq$ "On"

Consider the following domain property capturing a necessary condition for the target p.Motor $\neq$ "On":

p.Failure = *true* $\Rightarrow$ p.Motor $\neq$ "On"

Regressing the root obstacle backwards through this domain property generates the following subobstacle:

Obstacle PumpFailure
FormalSpec $\Diamond \exists$p: Pump
p.Switch = "On" $\wedge$ p.Failure = *true*

A variety of tactics are available for exploring alternative ways of resolving obstacles to a goal –such as *avoid obstacle*, *reduce obstacle likelihood*, *mitigate obstacle*, *weaken goal*, *substitute goal*, *restore goal*, or *substitute agent* [14]. For example, the obstacle mitigation tactic applied to the preceding obstacle PumpFailure may produce the following countermeasure goal :

Goal Achieve[MineEvacuated**When**PumpFailure**And**PumpSwitchOn]
FormalSpec $\forall$m: Miner, p: Pump
p.Switch = "On" $\wedge$ p.Failure = *true* $\Rightarrow$ $\Diamond_{\leq 10m}$ m.Position = "Out"

The lower part of Fig. 1 shows two OR-refinements for the obstacle Pump**Not**On**And**PumpSwitchOn.

Probabilistic goals and obstacles. Behavioral goals may be satisfied only partially [5]. The *probability of satisfaction* for a goal *G:* $C \Rightarrow \Theta T$ is defined as the ratio between (a) the number of possible behaviors satisfying both the goal antecedent *C* and consequent ΘT, and (b) the number of possible behaviors satisfying *C*. The *estimated probability of satisfaction* (EPS) of a goal is the probability of its satisfaction in view of its possible obstructions by obstacles. For goal *G*, it is denoted by *P(G).* The conditional probability *P(G|H)* denotes the probability of satisfaction of *G* over all behaviors satisfying *H*. The *required degree of satisfaction* (RDS) of a goal is the minimal probability of satisfaction admissible for this goal; it is prescribed by elicited requirements, existing regulations, standards, etc. For goal *G*, it is denoted by *RDS(G)*.

The *probability of an obstacle* is defined as the ratio between (a) the number of possible behaviors satisfying the obstacle, and (b) the number of possible behaviors. The probability of a root obstacle is computed by up-propagation though the obstacle refinement tree from estimates for leaf obstacles. The result is then up-propagated in turn through the AND/OR goal graph to determine the EPS of root goals. The severity of the consequences of obstacles to *G* is then assessed from the difference *P(G) - RDS(G)* [5].

III. Integrating Countermeasure Goals: Motivation

In addition to the PumpFailure obstacle in the previous section, the following other obstacle also results in severe loss of satisfaction of high-level goals:

Obstacle PowerSystemOutage
FormalSpec $\Diamond \exists$p: Pump
p.Switch = "On" $\wedge$ p.PowerLine = "Off"

Resolution tactics might produce, among others, the following countermeasure goals to resolve these obstacles (see Fig. 1):

Achieve [PumpRepaired**And**On**When**PumpFailure
AndPumpSwitchOn],
Maintain [EmergencyPumpOn**When**PumpFailure
AndPumpSwitchOn],
Maintain [ThirdPartyPumpOn**When**HighWaterDetected],
Achieve [PowerGeneratorOn**When**PowerSystemOutage
AndPumpSwitchOn].

The paper aims at providing precise and systematic answers to the following questions.

- Where should these countermeasure goals be integrated into the goal refinement graph? How?
- What parent goals should the countermeasure goals refine? With what other sibling subgoals? Should exceptional conditions such as p.Failure = *true* and p.PowerLine = "Off" pollute goal specifications throughout the entire goal model?

The answers to those questions should support the following objectives.

- *Separation of concerns.* The goals refering to normal situations should be distinguished from those handling obstacle occurrences. Such separation may significantly reduce model complexity and keep the ideal, normal model explicit.
- *Compositionality*. It should be possible to specify, structure, and analyze normal goals and countermeasures to their obstacles in a compositional way. A robust model

should not exhibit goal specifications or refinements intermixing normal cases and countermeasures. For example, consider a goal G obstructed by obstacles O_1 and O_2 with corresponding countermeasure CG_1 and CG_2, respectively. A brute-force integration might produce a cluttered specification for the robust version of this goal such as:

$$(\neg O_1 \wedge \neg O_2 \Rightarrow G) \wedge (O_1 \Rightarrow CG_1) \wedge (O_2 \Rightarrow CG_2)$$

The refinement of such a goal specification and, recursively, of its subgoals would thus intermix normal cases and combinations of exceptional cases which may lead, for a large number of obstacles, to a combinatorial blow-up of cases along refinements.

- *No premature decision and freedom of choice*. The modeler should be free to decide, when felt necessary, what are the goals refering to normal situations and what are those corresponding to exceptional situations. In our example, mine evacuation might or might not be part of the normal situation depending on its frequency, criticality, stakeholder wishes, and so forth.

IV. The Countermeasure Integration Problem

Integrating a countermeasure goal to an obstacle in a goal model means: (a) connecting this goal to a parent node in the model; (b) adding other sibling subgoals in the refinement if necessary; (c) propagating the corresponding changes along refinement trees in which the countermeasure goal is involved; and (d) refining this goal if necessary.

More precisely, an integration $Int_{CG,O}(M)$ of countermeasure CG to obstacle O in goal model M maps M to a new model M' so as to satisfy the following desired properties.

1. *Progress*. The application of the integration operator $Int_{CG,O}$ should increase the probability of satisfaction of some root goal G at least, that is, there must be at least one root goal G' in M' corresponding to G in M such that $P_{M'}(G') > P_M(G)$.

2. *Minimal change.* The application of $Int_{CG,O}$ should preserve prescribed behaviors in M that are not affected by O, that is, for any goal G in M such that $\{O, Dom\} \not\models \neg G$ and corresponding goal G' in M', we should have: $G' \models G$.

3. *Correctness preservation*. The correctness of goal refinements in M should be preserved in M', that is, if all refinements in M are complete, consistent, and minimal then those in M' are complete, consistent, and minimal as well.

V. Integrating Countermeasure Goals

This section explains how our integration operator ensures the three preceding properties.

A. Ensuring progress: valid countermeasures

For the *progress* constraint to be met, two conditions must hold on the countermeasure goal to be integrated.

1. Non-obstruction. The countermeasure goal CG may no longer be obstructed by the obstacle O it resolves:

$$\{O, Dom\} \not\models \neg CG \qquad \text{(non-obstruction)}$$

2. *Ancestor entailment.* The countermeasure goal CG must guarantee a deidealized version of an ancestor of the leaf goal obstructed by O. To make this further precise we need the following definitions.

A goal G' is a *deidealized version* of goal G if $G \models G'$.

A deidealized version G' of G is *acceptable* if, for every goal refinement with G as a child, there exists an acceptable deidealized version of its siblings and parents such that the corresponding refinement still meets the completeness, consistency and minimality conditions recalled in Section II.

The ancestor entailment condition can now be formulated as follows:

$$\{CG, G_1', \ldots, G_n', Dom'\} \models PG' \qquad \text{(ancestor-entailment)}$$
for some acceptable deidealized version PG' of ancestor PG,

where PG is an ancestor of the obstructed goal G and G_1', ..., G_n' are acceptable deidealized versions of descendants of PG.

A countermeasure goal CG against obstacle O is said to be *valid* if it satisfies the *non-obstruction* and *ancestor-entailment* conditions.

For example, the countermeasure goal Achieve [MineEvacuated**When**PumpFailure**And**PumpSwitchOn] is valid; the obstacle PumpFailure does not obstruct it, and this goal together with the deidealized goal Avoid [OverflowedMine**When** NoPumpFailure] guarantee the satisfaction of the parent goal Avoid [MinersDrowning]. In this example, the parent goal is not deidealized.

Obstacle resolution tactics such as *avoid obstacle*, *reduce obstacle likelihood*, *mitigate obstacle*, *weaken goal*, or *substitute goal* [14] can be shown to produce valid countermeasures modulo propagations of their effect through the refinement trees in which they are involved (see Section V.C).

Multiple candidate ancestors might be considered for the *ancestor-entailment* condition. In our example, Avoid [OverflowedMine] and Avoid [MinersDrowning] are potential candidates with respect to the goal Achieve [MineEvacuated**When** PumpFailure**And**PumpSwitchOn]. The nearest candidate to this goal appears preferable for more local model change –that is, the goal Avoid [OverflowedMine].

The *anchor* for a countermeasure goal CG is the lowest ancestor goal PG meeting the *ancestor-entailment* condition. It is the goal through which the countermeasure goal is integrated, as discussed in the next section.

Theorem (*Progress*). For any valid countermeasure goal CG, the probability of satisfaction of its anchor PG increases:

$$P(PG') > P(PG),$$

where PG' in M' corresponds to PG in M.

This can be proved *ab absurdo*. Assume there is no such increase: $P(PG') \leq P(PG)$. Introducing conditional probabilities, we have:

$$P(PG') = P(O) \times P(PG'|O) + P(\neg O) \times P(PG'|\neg O),$$
$$P(PG) = P(O) \times P(PG|O) + P(\neg O) \times P(PG|\neg O).$$

Given the obstruction of PG, we have: $P(PG|O) = 0$. Therefore,

$$P(O) \times P(PG'|O) + P(\neg O) \times P(PG'|\neg O) \leq P(\neg O) \times P(PG|\neg O)$$

Since PG' is a deidealized version of PG, we have:

$$P(PG'|\neg O) \geq P(PG|\neg O).$$

Therefore,

$$P(\neg O) \times P(PG'|\neg O) \geq P(\neg O) \times P(PG|\neg O).$$

As $P(O) \times P(PG'|O) \geq 0$, our initial assumption gets contradicted.

B. Ensuring minimal changes: integration schemas

A single valid countermeasure goal ensures progress towards a complete model; its integration in the model should also ensure that the *minimal change* property is met (see Section IV).

Two alternative integration schemas may be used for this, dependent on the obstacle resolution tactic being selected [14]. The first schema removes the obstructed goal; it should be applied when the *substitute goal* or *weaken goal* tactic is used for resolving the obstacle. The second integration schema keeps the obstructed goal in the model; it should be applied when the *avoid obstacle*, *reduce obstacle likelihood*, or *mitigate obstacle* tactic is used.

Removing the obstructed goal. Fig. 2 shows a first integration schema expressed as a model rewriting rule. In this first schema, the refinement of anchor goal *AG*, containing at least one obstructed goal, is replaced with a new refinement. The latter contains the countermeasure goal *CG* to leaf obstacle *LO* together with all non-obstructed children. (An anchor's *obstructed children* are those being directly or indirectly obstructed by *LO*.) The anchor *AG* may need to be deidealized; in this case, *AG'* replaces *AG* in the new goal model.

This first integration schema has a precondition for use, namely, the countermeasure goal *CG* and the non-obstructed children are sufficient for satisfying the anchor goal:

$$\{CG, Children(AG) \setminus ObstructedChildren\} \models AG'.$$

Otherwise, the new refinement would not be complete.

For example, the goal Maintain [ThirdPartyPumpOn**When**HighWaterDetected] is a countermeasure produced through the *goal substitution* tactic [14]. Its anchor goal is Maintain [PumpOn**When**HighWater]. The following refinement is complete, consistent, and minimal for this anchor goal:

Maintain [PumpOn**When**HighWater]
← Maintain [HighWaterDetected**When**HighWater]
← Maintain [ThirdPartyPumpOn**When**HighWaterDetected]

We may therefore replace the old refinement with this one.

It is easy to see that this first integration schema meets our *minimal change* property. Non-obstructed goals are composed from non-obstructed goals in the refinements of *AG* and its descendants, and in the siblings of *AG* and its ancestors. The former are kept as is whereas the latter might need to be deidealized through change propagation (see Section V.C). All these goals thus satisfy $G' \models G$.

Keeping the obstructed goal. Fig. 3 shows a second integration schema. In this schema, a new refinement is introduced; it includes a modified version of the obstructed anchor and the countermeasure goal. The obstructed anchor is deidealized for removing the obstruction. The negation of the leaf obstacle is added to form a decomposition by cases.

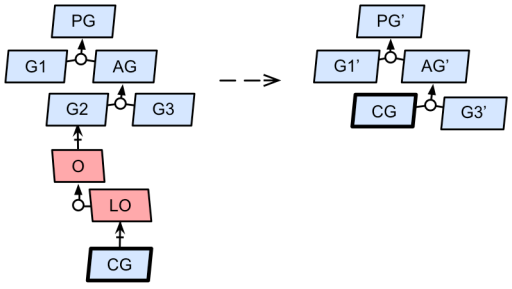


Fig. 2. Integration schema with obstructed goal being removed

This second integration schema has a precondition for use as well; the countermeasure goal must be sufficient for satisfying the anchor goal when the obstacle occurs:

$$\{LO, CG, Dom\} \models AG'$$

For example, the countermeasure goal Achieve [Mine EvacuatedWhenPumpFailureAndPumpSwitchOn] entails its anchor Avoid [MinersDrowning] when the obstacle occurs. This second rule then produces the following refinement:

Avoid [OverflowedMine]
← Avoid [OverflowedMine**When**NoPumpFailure]
← Achieve [MineEvacuated**When**PumpFailure**And**PumpSwitchOn]

Note that the anchor goal is not deidealized in this example. The new goal Avoid [OverflowedMine**When**NoPumpFailure] is a de-idealization of the anchor goal. The negation of the obstacle is added to the goal antecedent:

Goal Avoid [OverflowedMine]
FormalSpec ∀p: Pump
p.Failure = *false* ⇒ ¬OverflowedMine

This second integration schema can be seen to meet our *minimal change* property as well; the reasoning is similar to the first schema. Only siblings of the anchor goal and its ancestor may need to be deidealized. The other goals were obstructed by the resolved obstacle and are therefore not concerned with the *minimal change* property.

C. Preserving refinement correctness: change propagation

As introduced before, goal deidealizations and countermeasure integrations may require corresponding changes to be propagated along refinement trees in which the countermeasure goal is involved. Such propagations are intended to ensure our third property on integrations, that is, the resulting model must remain complete, consistent, and minimal. This section discusses how deidealizations and change propagations are performed.

Deidealization by strengthening the goal's antecedent. A first way of deidealizing a goal of form $C \Rightarrow \Theta T$ is to add an adequate conjunct to its antecedent:

$$AddConjunct(C \Rightarrow \Theta T, \Theta EC) = [C \wedge \Theta EC] \Rightarrow \Theta T.$$

For example, the goal Maintain [PumpOn**When**PumpSwitchOn] may be deidealized so as to exclude pump failure from pump actuation:

Goal Maintain [PumpOn**When**NoPumpFailure**And**PumpSwitchOn]
FormalSpec ∀p: Pump
[p.Switch = "On" ∧ p.Failure = *false*] ⇒ p.Motor = "On"

Deidealization by weakening the goal's consequent. A second way of deidealizing the goal $C \Rightarrow \Theta T$ is to add an adequate disjunct to its consequent:

$$AddDisjunct(C \Rightarrow \Theta T, \Theta ED) = C \Rightarrow [\Theta T \vee \Theta ED].$$

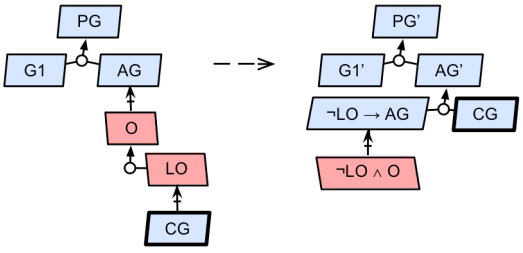


Fig. 3. Integration schema with obstructed goal being kept

For example, the goal Maintain[PumpOn**When**PumpSwitchOn] might be deidealized so as to require the pump to be actuated or the emergency pump to be activated:

Goal Maintain [PumpOn**Or**EmergencyPumpOn**When**PumpSwitchOn]
FormalSpec ∀p: Pump
p.Switch = "On" ⇒ [p.Motor = "On"
∨ ∃ ep: EmergencyPump · ep.Switch = "On"]

Change propagation. When a goal is deidealized, the change must be propagated along the refinement trees in which this goal is involved –both up and down such trees. The *AddConjunct* or *AddDisjunct* operators therefore have to be applied recursively to the other goals up and down refinement links.

For example, the integration of the countermeasure goal Achieve[MineEvacuated**When**PumpFailure**And**PumpSwitchOn] requires change propagation to the descendants of the goal Avoid [OverflowedMine]. First, this goal is modified as previously shown. Next, the change is down-propagated, leading to an application of *AddConjunct* to the goal Maintain [PumpOn**When** HighWater]:

Goal Maintain [PumpOn**When**NoPumpFailure**And**HighWater]
FormalSpec ∀p: Pump, s: Sump
s.WaterLevel = "High" ∧ PumpInSump(p, s) ∧ p.Failure = *false*
⇒ p.Motor = "On"

The next refinement instantiates the *unmonitorability-driven* refinement pattern. The *AddConjunct* operator is therefore applied to the goal Maintain[PumpOn**When**HighWaterDetected]. We obtain:

Goal Maintain [PumpOn**When**NoPumpFailure**And**HighWater Detected]
FormalSpec ∀p: Pump, c: PumpController
c.HighWaterSignal = "On" ∧ CtrlPump(c, p) ∧ p.Failure = *false*
⇒ p.Motor = "On"

The next refinement instantiates the *milestone-driven* refinement pattern. The *AddConjunct* operator is therefore applied to both children. We thereby obtain:

Goal Maintain [PumpSwitchOn**When**NoPumpFailure **And**HighWaterDetected]
FormalSpec ∀p: Pump, c: PumpController
c.HighWaterSignal = "On" ∧ CtrlPump(c, p) ∧ p.Failure = *false*
⇒ p.Switch = "On"

Goal Maintain [PumpOn**When**NoPumpFailure**And**PumpSwitchOn]
FormalSpec ∀p: Pump, c: PumpController
p.Switch = "On" ∧ p.Failure = *false* ⇒ p.Motor = "On"

Since these goals are leaf goals, the propagation ends.

Change propagation in the general case proceeds as follows. When *AddConjunct* or *AddDisjunct* is applied to a goal for deidealization, the pattern used for refining (resp. abstracting) it is identified. A *propagation pattern* associated with the refinement/abstraction pattern tells us what goals in the refinement (resp. abstraction) must be modified and how. The process is applied recursively to the subgoals (resp. parents) until leaf goals (resp. root goals) are reached.

For example, if the *case-driven* pattern is used for refining a goal through multiple disjoint cases [13], the application of *AddConjunct* or *AddDisjunct* to a child goal requires the application of the same operator to the parent goal (and vice-versa). If the *milestone-driven* pattern is used for refining a goal through milestone subgoals [13], an application of *AddConjunct* to the first milestone subgoal requires the application of the same operator to the parent goal –but not necessarily to the other subgoals; the corresponding extra condition is not necessarily relevant to the latter.

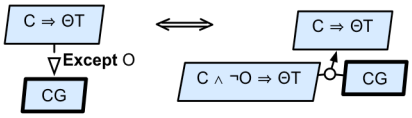


Fig. 4. Semantic equivalent of *Except*

For a given refinement pattern and application of *AddConjunct* or *AddDisjunct* to a goal, there might be alternative modifications of the refinement/abstraction structure. Moreover, other mechanisms are required for change propagation to goals not obtained through refinement patterns. Even though pattern-based propagation can be performed semi-automatically, the general problem of automatic change propagation through arbitrary refinement structures remains open.

VI. Obstacle Resolution as Exception Handling

The integration of countermeasure goals through new, explicit refinements in the original model raises several issues.

- The goal graph might undergo significant changes each time a new obstacle is identified.
- Normal situations would be mixed with exceptional ones; it might be hard to distinguish the former from the latter without domain expertise.
- Goal specifications become increasingly more complex.
- As new countermeasures are introduced, the ordered nesting of exceptional cases along refinements may lead to a combinatorial blow-up of special cases.

This section introduces a slight extension of the goal specification language that solves those issues. Dedicated constructs are provided for encapsulating the required modifications while documenting each exceptional case separately.

A. Extending the goal specification language

Except. A first construct links a countermeasure goal to its anchor goal:

Goal AG
FormalSpec C ⇒ ΘT
Except O **then** CG,

where *AG* denotes the anchor goal C ⇒ ΘT of countermeasure goal *CG* to obstacle *O*. Semantically, this implicit specification is fully equivalent to the refinement in Fig. 4.
This construct may be used under the following precondition:

{O, CG, Dom} ⊨ AG

For example, the goal Avoid [MinersDrowning] is satisfied in the ideal situation by avoiding mine overflow. Under the exceptional condition of a pump failure, the goal is guaranteed through miners evacuation. We may therefore write:

Goal Avoid [MinersDrowning]
Except PumpFailure
then Achieve [MineEvacuated**When**PumpFailure **And**PumpSwitchOn]

This specification is logically equivalent to the refinement illustrating the second integration schema in Section V.B.

Multiple *Except* annotations may be attached to a single goal to cope with different obstacles; the latter may therefore be introduced incrementally. Compared with the complexity of an equivalent explicit specification, the complexity of an implicit goal specification with multiple *Except* annotations

remains linear in the number of exceptions. The specification of the ideal goal remains unchanged. Moreover, multiple annotations sharing the same countermeasure goal may be factored out to simplify the model.

Provided. A second construct specifies an extra conjunct on the antecedent of an ideal goal *G* to produce a countermeasure:

Goal G
FormalSpec C ⇒ ΘT
Provided EC,

where *EC* denotes an extra conjunct to be added to *G*'s antecedent for resolving the considered obstacle. Semantically, this implicit specification is equivalent to:

Goal G
FormalSpec [C ∧ EC] ⇒ ΘT.

The *Provided* construct is typically used for deidealizing goals. For example, the deidealization of the goal Maintain [PumpOn**When**PumpSwitchOn] may be specified by highlighting the normal situation as follows:

Goal Maintain [PumpOn**When**PumpSwitchOn]
FormalSpec ∀p: Pump
p.Switch = "On" ⇒ p.Motor = "On"
Provided p.Failure ≠ f*alse*.

It often proves convenient to write **ProvidedNot** *EC* instead of **Provided** ¬*EC*.

RelaxedTo. Symmetrically to *Provided*, this construct specifies an extra disjunct on the consequent of an ideal goal *G* to produce a countermeasure:

Goal G
FormalSpec C ⇒ ΘT
RelaxedTo ED,

where *ED* denotes an extra disjunct to be added to *G*'s consequent for resolving the considered obstacle. Semantically, this goal is equivalent to:

Goal G
FormalSpec C ⇒ [ΘT ∨ ED].

This construct is useful for deidealizing goals as well. For example, another deidealization of the same goal Maintain [PumpOn**When**PumpSwitchOn] might be specified by highlighting the normal situation as follows:

Goal Maintain [PumpOn**When**PumpSwitchOn]
FormalSpec ∀p: Pump
p.Switch = "On" ⇒ p.Motor = "On"
RelaxedTo ∃ep: EmergencyPump · ep.Switch = "On".

Multiple *Provided* and *RelaxedTo* annotations may be attached to a single goal to introduce multiple countermeasures.

Replaces. This construct appears useful for tracing previous versions of a goal:

Goal G'
Replaces G

Such traceability helps readers understand the rationale behind the final goal, e.g.,

Goal Maintain [ThirdPartyPumpOn**When**PumpSwitchOn]
Replaces Maintain [PumpOn**When**PumpSwitchOn].

B. Exception diagrams

Textual goal specifications with *Except* and *Replaces* annotations may be graphically represented in an exception diagram. Fig. 5 shows a portion of such a diagram for the goal Maintain [ThirdPartyPumpOn**When**PumpSwitchOn]. This diagram captures that (a) when the obstacle PumpFailure occurs the countermeasure goal Achieve [PumpRepaired**And**On**When** PumpFailure**And**PumpSwitchOn] will guarantee this goal, and (b) this goal replaces Maintain [PumpOn**When**PumpSwitchOn]. Note that the *Except* annotation has been propagated to the replacing goal.

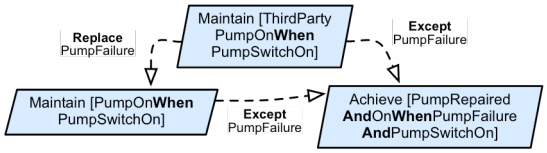


Fig. 5. Exception diagram

VII. Model Refactoring for Attaching or Detaching Goal Exceptions

In practice, the analyst should decide at some point whether a countermeasure goal refers to an exceptional situation or to a normal one to be considered in the ideal model. Such a decision might depend on various factors such as the frequency of the resolved obstacle, the criticality of the obstructed goal, domain-specific culture, stakeholders wishes, and so forth. To make the decision flexible and easily reversible, this section presents model refactoring operators for attaching or detaching the annotations introduced in Section VI to/from a goal model.

Three operators are available for transforming an annotated model portion into a standard one.

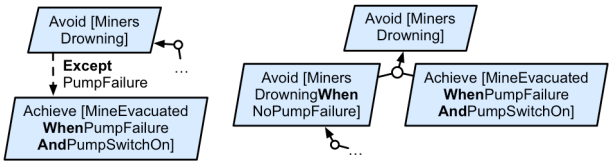
Detach-Except, applied to an annotated goal, produces a new model where the goal is no longer annotated with a specific *Except* clause. The operator introduces a new refinement with two children: the countermeasure goal and a deidealization of the original goal (see Fig. 4 from left to right). The children of the original goal are then children of the deidealized goal. Back to an earlier example, the operator takes the model fragment in Fig. 6a to produce the model fragment in Fig. 6b.

Detach-Provided, applied to an annotated goal, produces a new model where a specific *Provided* annotation is "compiled" into its equivalent formal specification. For example, after application of this operator the goal Maintain [PumpOn**When** PumpSwitchOn] is specified without its *Provided* annotation as follows:

Goal Maintain [PumpOn**When**PumpSwitchOn]
FormalSpec ∀p: Pump
[p.Switch = "On" ∧ p.Failure = *false*] ⇒ p.Motor = "On"

Detach-RelaxedTo, applied to an annotated goal, produces a new model where a specific *RelaxedTo* annotation is removed. Back to an earlier example, the application of this operator to the goal Maintain [PumpOn**When**PumpSwitchOn] yields the following goal specification:

Goal Maintain [PumpOn**When**PumpSwitchOn]
FormalSpec ∀p:Pump
p.Switch = "On" ⇒ [p.Motor = "On"
∨ (∃ep: EmergencyPump) ep.Switch = "On"]



(a) (b)

Fig. 6. Implicit and explicit countermeasure integration

Similarily, three operators are available for transforming a standard model portion into an annotated one –namely, *Attach-Except*, *Attach-Provided* and *Attach-RelaxedTo*. These operators are the reverse of the *Detach* ones.

VIII. EVALUATION

The techniques presented in this paper were applied[1] to a benchmark commonly used for evaluating obstacle analysis techniques [1, 14, 15]. The goal and obstacle models used for the London Ambulance System (LAS) are based on [14].

The goal model contains 42 goals, 19 refinements instantiating a variety of refinement patterns, and 8 agents. The obstacle model contains 71 obstacles and 30 countermeasure goals. The full models can be found in [14, 15]. Only portions of the goal model are considered here.

The top goal in this model is Achieve [IncidentResolved]. The *milestone-driven* refinement pattern produces three subgoals: Achieve [IncidentReported], Achieve [AmbulanceOnScene**When**IncidentReported] and Achieve [IncidentResolved**When**AmbulanceOnScene]. At a lower level of refinement, the goal Achieve [AmbulanceMobilized**When**Allocated] states that allocated ambulances shall be mobilized within 3 minutes. This goal is refined using a *case-driven* pattern into Achieve [AmbulanceMobilizedAtStation**When**Allocated] and Achieve [AmbulanceMobilizedOnRoad**When**Allocated]. These goals are in turn refined until they are assignable to single agents.

Obstacles to leaf goals were then generated and refined. Here is a sample of obstacle refinements in textual format:

MobilizationOrderPrinted**And**Ambulance**Not**Mobilized
- ← AmbulanceNoLongerAtStation
- ← AmbulanceNoLongerAvailable
- ← MobilizationOrderIgnored
- ← MobilizedToWrongDestination
- ← MobilizationOrderTakenByOtherAmbulance

Countermeasures goals were explored using available resolution tactics [14]. For example, here are countermeasure goals for two leaf obstacles:

MobilizationOrderTakenByOtherAmbulance
- ← Achieve [MobilizationByOtherAmbulanceKnown]
- ← Avoid [MobilizationWithoutOrder]

MDT-MobilizationOrderIgnored
- ← Achieve [SoundAlarmRaised**When**MDTMobOrderReceived]
- ← Achieve [FailedMobilizationRecovered]

The large number of obstacles and countermeasure goals called for our countermeasure integration and encapsulation techniques. As a result, the countermeasure goals appear to focus on a small number of important goals; e.g., the goal Achieve [IncidentResolvedByAmbulanceIntervention] has 15 exceptions. The overall integration produced 34 exceptions distributed over 7 goals only.

The techniques presented in this paper helped significantly for the following reasons.

Model simplification by separation of concerns. The goals refering to normal situations were systematically distinguished from those handling obstacle occurrences. Emerging assumptions were incrementally down-propagated to obstructed descendants of corresponding anchor goals; this required 7 propagations and produced 28 *Provided* annotations distributed over 6 goals. Without these annotations the formal specification of those 6 goals would have been cluttered with details related to exceptional cases. Table II quantifies our use of *Provided* annotations.

For example, the goal Achieve [AllocatedAmbulanceMobilization**When**MobilizationOrderPrinted] is defined as follows after integration in the model:

Goal Achieve [AllocatedAmbulanceMobilized**When**MobilizationOrderPrinted]
Provided AllocatedAmbulance**Not**LeavingBeforeMobilization
Provided AllocatedAmbulance**Not**UnavailableBeforeMob
Provided PrintedMobilizationOrder**Not**Ignored
Provided Mobilization**Not**TakenByOtherAmbulance.

The full equivalent specification of this goal without *Provided* annotations would completely hide the ideal case; it would then appear fairly hard to distinguish the part of the goal antecedent related to the ideal case from those related to exceptional cases.

The *Detach-Except* operator was applied to the *case-driven* refinement of the goal Achieve [AmbulanceMobilized**When**Allocated]. Allocating an ambulance when not at station was estimated fairly rare – 5% of cases according to typical figures in the domain. The parent goal of these two goals was therefore modified accordingly:

Goal Achieve [AmbulanceMobilized**When**Allocated]
Except AllocatedAmbulance**Not**AtStation
then Achieve [AllocatedAmbulanceMobilizedOnRoad]

Such refactoring reduces model complexity by hiding the part of the model handling the mobilization of an ambulance when on road. The resulting ideal goal model therefore contains fewer refinements and fewer goals, making it easier to understand and clearly separate ideal behaviors from exceptional ones.

Compositionality. Without our techniques, the integration of so many exceptions for only 7 goals would have resulted in large, complex refinements with a combinatorial blow-up of special cases. To illustrate this important point, consider the goal Achieve [AmbulanceMobilized**When**Allocated]. Its original, ideal specification is:

$$\forall amb: Ambulance, inc: Incident \; Allocated(amb, inc) \Rightarrow \Diamond_{\leq 3min} \exists amb: Ambulance \cdot Mobilized(amb, inc)$$

After obstacle analysis, this goal is guaranteed through 5 countermeasure goals (see Fig 7). The brute-force integration of only the three countermeasure goals depicted at the bottom of Fig. 6 would have resulted in the following formal specification for the final version of the goal Achieve [AmbulanceMobilized**When**Allocated]:

$$\forall c: UrgentCall, inc: Incident \; Allocated(amb, inc) \Rightarrow \Diamond_{\leq 3min} \exists \; amb: Ambulance \cdot Mobilized(amb, inc)$$
$$\vee [\, \Box_{>3min} \neg AmbAvailable(amb, inc) \rightarrow \Diamond_{\leq 6min} \exists amb': Ambulance \; amb \neq amb' \wedge Mobilized(amb', inc) \,]$$
$$\vee [\, \Box_{>3min} DisplayedMobilizationIgnored(amb, inc) \rightarrow \Diamond_{<6min} Mobilized(amb', inc) \,]$$
$$\vee [\, \Box_{>3min} PrintedMobilizationIgnored(amb, inc) \rightarrow \Diamond_{<6min} Mobilized(amb', inc) \,]$$

In addition to this complex specification, the goal refinement structure would have been heavily modified:

Achieve [AmbulanceMobilized**When**Allocated]
- ← Achieve [OtherAmbMob**When**AllocatedAmbUnavailable]
- ← Achieve [AllocAmbMobilized**When**AmbAvailableUntilMob]

[1]See http://www.info.ucl.ac.be/~acaillia/publications/las-system.html for full report.

← Achieve [LateMob**When**DisplayedMobOrderIgnored]
← Achieve[AllocAmbMobilized**When**AmbAvailableUntilMob **And**DisplayedMobOrder**Not**Ignored]
← Achieve [LateMob**When**PrintedMobOrderIgnored]
← Achieve [AllocAmbMobilized**When**AmbAvailUntilMob **And**DisplayedMobOrder**Not**Ignored **And**PrintedMobOrder**Not**Ignored]
← …

With such a brute-force integration, each countermeasure goal must be refined by taking other countermeasures into account. This would lead to a combinatorial blow-up of cases. Thanks to our technique, the original specification of this goal and its refinement structure are preserved. The *Except* and *Provided* constructs encapsulate the modifications for a more robust system. Table I provides some figures on goal exceptions for other goals.

No premature decision and freedom of choice. The specification and documentation of exceptional behaviors was separated from the normal ones; this allowed us delaying the decision of how and when the handling of exceptional cases should occur.

Other benefits. The *Replaces* annotation was felt useful for documenting the replacing countermeasure goals –e.g., Achieve [MobilizedAmbIntervention**Or**MobilizationCancelled] replacing Achieve [MobilizedAmbulanceIntervention] to resolve the obstacle MobilizationCancelled. Without this annotation we would have lost the previous version of the goal.

Exception diagrams significantly helped understand the model where all countermeasures are integrated; they document exceptions one single goal at a time (see Fig. 7). A total of 7 exception diagrams was produced for documenting exceptional cases and countermeasure goals.

Tool support. Our evaluation on the LAS case study was supported by a preliminary tool prototype. Given an obstacle resolution tactic and the corresponding anchor goal, the tool automatically generates the corresponding *Except* or *Replaces* anotations with corresponding countermeasure goal. The *Provided* and *RelaxedTo* constructs are supported as well. The tool also generates exception diagrams.

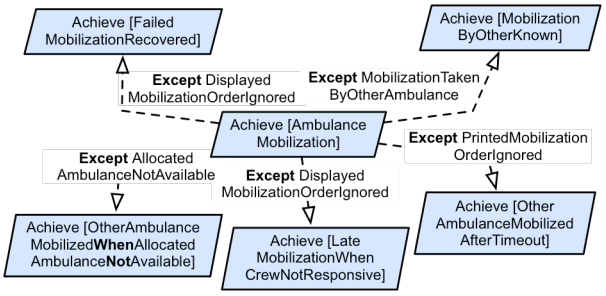


Fig. 7. Exception diagram for Achieve [AmbulanceMobilization]

IX. Related Work

In the *identify-assess-control* cycles of risk analysis at requirements engineering time [8, 13, 14, 17, 20], most of the work so far has been devoted to risk identification and assessment. For risk identification, scenario-based heuristics are available [2, 29] as well as goal-oriented formal techniques [1, 14]. For risk assessment, various kinds of quantitative techniques are available [3, 5, 8, 25]. For risk control, the only work on countermeasure exploration is [14] where the obstacle resolution tactics mentioned in this paper are described. We are not aware of any work on systematic integration of countermeasures in a requirement model with a clear, precise semantics.

TABLE I. USING GOAL EXCEPTIONS

Goals	Exceptions
Achieve [Incident Resolved By Ambulance Intervention]	15
Achieve [Ambulance Mobilization]	6
Achieve [Allocated Ambulance Mobilization **When** Mobilization Order Printed]	5
Achieve [Mobilized Ambulance Intervention]	3
Achieve [Mobilized Ambulance Intervention **Or** Mobilization Cancelled]	3
Achieve [Allocated Ambulance Mobilization **When** Mobilization Order Displayed]	2
Achieve [Allocated Ambulance Mobilization At Station]	1

TABLE II. USING PROVIDED-CLAUSES

Goals	Provided
Achieve [Allocated Ambulance Mobilization On Road]	3
Achieve [Allocated Ambulance Mobilization At Station Based On Location Info]	4
Achieve [Allocated Ambulance Mobilization **When** Mobilization Order Printed]	4
Achieve [Allocated Ambulance Mobilization On Road Based On Location Info]	3
Achieve [Allocated Ambulance Mobilization **When** Mobilization Order Displayed]	3
Achieve [Allocated Ambulance Mobilization At Station]	1

The relevance and importance of default-based reasoning has been recognized in the context of elaborating requirements or specifications. In [30], a formal framework is proposed for reasoning about evolving requirements. The framework is based on belief revision and default theory; operators for adding and retracting requirements are defined together with formal conditions for their valid application (similarly to our integration operators). The tracing of exceptional requirements is not discussed there. In [24], a specification is structured through axioms and *Overrides* relations. Such relations are derived from the structural decomposition of the system. Specific axioms predominate more general ones when a conflict occurs. This framework comes with formal foundations and well-defined procedures for identifying conflicts and predominance among axioms. It appears more oriented towards specification elaboration. In [27], default specifications are introduced together with exceptions in order to increase the completeness of algebraic specifications; the *But* relation there somewhat corresponds to our *Except* relation.

Our approach mainly differs from those previous efforts in the following directions.

- Our techniques operate at requirements level and benefit from the refinement structure of a goal model. This structure helps in building a model where exception handling is integrated and in propagating required changes throughout the model.
- New requirements for a more robust system are incrementally integrated through obstacle analysis. The model updates are traceable back to the identified obstructed goals and their obstacles.

At programming level, aspects may be used for separating exception handling from normal code [19]. At modelling level, [28] convincingly shows how aspects can be used for separating exceptional behaviors from normal ones. As an alternative to the approach advocated in this paper, robustness

aspects might be incorporated in a goal model by use of constructs similar to the ones sketched in, e.g., [10, 23]. Further work would however be required to define a declarative, logic-based semantics as well as an operational, trace-based semantics for such constructs –which seems unavailable to date. Suitable weaving mechanisms would then need to be defined in this semantic framework.

X. CONCLUSION

The paper presented systematic techniques for integrating countermeasures into ideal goal models. An integration operator was introduced as a model transformation ensuring *progress* towards a more complete model, *minimal change* of the original model, and *refinement correctness preservation*. *Anchor goals* were introduced to define where countermeasure goals should be integrated together with appropriate refinement schemas. Our goal-oriented RE framework was extended with constructs for structuring and documenting exceptional cases. Coming with these, model refactoring operators were proposed enabling analysts to attach and detach exceptions. The approach was evaluated on two case studies, a simple mine pump system and a much more complex ambulance despatching system.

As shown in these case studies, a more complete goal model is obtained while the ideal model is kept visible. The ideal specifications are preserved. The final refinement structure turns out to be nearly the same as the original one. Exceptions are documented aside; analysts and users of the model can dive into independent exceptions one by one. A large number of countermeasure goals can be integrated; the integration techniques reduce model complexity by keeping the combinatorial blow-up of exceptional cases implicit.

The current version of our tool is fairly basic. Among the planned extensions, the increased automation of change propagation deserves highest priority. The propagation procedure itself should be made less dependent on common refinement patterns.

Complementary techniques are needed for selecting "best" countermeasures according to soft goals from the goal model. The responsibilities of agents in exception handling should be integrated as well. Moreover, the use of our exception-related constructs for deriving exception handlers in the corresponding software architecture would be worth investigating. In parallel, their exploitation for runtime self-adaptation in changing contexts appears a promising direction for future work.

ACKNOWLEDGEMENT

This work was supported by the EU Fund for Regional Development & the Walloon Region (TIC-FEDER Grant CE-IQS). We wish to thank B. Lambeau, C. Damas and S. Busard for discussions on our approach, and the reviewers for useful comments.

REFERENCES

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

[2] A. Anton and C. Potts, "The Use of Goals to Surface Requirements for Evolving Systems", *Proc. ICSE'98: Intl. Conf. Softw. Eng.*, May 1998.

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

[4] T. Bedford and R. Cooke, *Probabilistic Risk Assessment-Foundations and Methods*. Cambridge University Press, 2001.

[5] A. Cailliau and A. van Lamsweerde, "Assessing requirements-related risks through probabilistic goals and obstacles", *Requirements Engineering Journal 18(2)*, Springer-Verlag, 2013, 129-146.

[6] R. Darimont and M. Lemoine, "Security Requirements for Civil Aviation with UML and Goal Orientation", *Proc. REFSQ'07: Intl. Conf. on Foundations for Softw. Quality*, LNCS 4542, Springer-Verlag, 2007.

[7] R. Darimont and A. van Lamsweerde, "Formal Refinement Patterns for Goal-Driven Requirements Elaboration", *Proc FSE'96: 4th ACM Symp. on the Foundations of Softw. Eng. (FSE'4)*, Oct.1996, 179-190.

[8] M.S. Feather and S.L. Cornford, "Quantitative Risk-Based Requirements Reasoning", *Req. Eng. Journal 8(4)*, Springer-Verlag, 2003, 248-265.

[9] M. Feather, S. Fickas, A. van Lamsweerde, and C. Ponsard, "Reconciling System Requirements and Runtime Behaviour", *Proc. IWSSD'98 - 9th Intl. Workshop on Soft. Spec. and Design*, Isobe, IEEE, April 1998.

[10] A. Gil, J. Araujo., "AspectKAOS: integrating early-aspects into KAOS", *Proc. 15th Workshop on Early Aspects*, ACM, 2009, 31-36.

[11] M. Joseph, *Real-Time Systems: Specification, Verification and Analysis*, Prentice Hall Intl., 1996.

[12] 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.

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

[14] A. van Lamsweerde and Emmanuel Letier, "Handling Obstacles in Goal-Oriented Requirements Engineering", *IEEE Trans. Softw. Eng. 26(10)*, October 2000, 978-1005.

[15] E. Letier, *Reasoning about Agents in Goal-Oriented Requirements Engineering*,. PhD Thesis, Univ. Cath. Louvain, May 2001.

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

[17] N. Leveson, "An Approach to Designing Safe Embedded Software", *Proc. EMSOFT 2002 – Embedded Software: 2nd Intl. Conference*, Grenoble, LNCS 2491, Springer-Verlag, October , 2002, 15-29.

[18] N.G. Leveson, *Engineering a Safer World.* MIT Press, 2011.

[19] M. Lippert, C. V. Lopes, "A study on exception detection and handling using aspect-oriented programming", *Proc. ICSE'2000: International Conference on Software Engineering*, IEEE, 2000, 418-427.

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

[21] 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", *Req. Eng. Journal 12(1)*, 2007, 41-54.

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

[23] G. Mussbacher, D. Amyot, J. Araújo, A. Moreira, M. Weiss, "Visualizing Aspect-Oriented Goal Models with AoGRL", *2nd Intl. Workshop on Requirements Engineering Visualization*, IEEE, 2007.

[24] M. Ryan, "Defaults in Specifications", *Proc. First IEEE International Symposium on Requirements Engineering*, 1993, 142-149.

[25] M. Sabetzadeh, D. Falessi, L. Briand, S. Di Alesio, D. McGeorge, V. Ahjem and J. Borg, "Combining Goal Models, Expert Elicitation, and Probabilistic Simulation for Qualification of New Technology, Proc". *IEEE 13th Intl. Symp. on High-Assurance Syst. Eng.*, Nov. 2011, 10-12.

[26] B. Schneier, *Secrets and Lies: Digital Security in a Networked World.* Wiley, 2004.

[27] P.-Y. Schobbens, "Exceptions for algebraic specifications: on the meaning of but", *Sci. Computer Programming 20(1-2)*, 1993, 73-111.

[28] A. Shaukat, L. Briand, H. Hemmati, "Modeling robustness behavior using aspect-oriented modeling to support robustness testing", *Software & Systems Modeling 11(4)*, 2012, 633-670.

[29] A. Sutcliffe, N.A. Maiden, S. Minocha, and D. Manuel, "Supporting Scenario-Based Requirements Engineering", *IEEE Trans. Software Eng. 24(12)*, Dec. 1998, 1072-1088.

[30] D. Zowghi and R. Offen, "A Logical Framework for Modeling and Reasoning About the Evolution of Requirements", *Proc. 3rd IEEE Intl. Symp. on Requirements Engineering*, 1997, 247-257.