Engineering Requirements for System Reliability and Security

Axel van LAMSWEERDE
Université catholique de Louvain
B-1348 Louvain-la-Neuve
avl@info.ucl.ac.be

Abstract. Requirements engineering (RE) is concerned with the elicitation of the objectives to be achieved by the system-to-be, the operationalization of such objectives into specifications of requirements and assumptions, the assignment of responsibilities for those specifications to agents such as humans, devices and software, and the evolution of such requirements over time and across system families. Getting high-quality requirements is difficult and critical. Poor requirements were recurrently recognized to be the major cause of system failures. The consequences of such failures may be especially harmful in mission-critical systems.
This paper overviews a systematic, goal-oriented approach to requirements engineering for high-assurance systems. The target of this approach is a complete, consistent, adequate, and structured set of software requirements and environment assumptions. The approach is model-based and partly relies on the use of formal methods *when and where needed* for RE-specific tasks, notably, goal refinement and operationalization, analysis of hazards and threats, conflict management, and synthesis of behavior models.

Keywords. Requirements engineering, goal refinement, hazard analysis, threat analysis, inconsistency management, model synthesis from scenarios, agent modeling.

1. Introduction

Requirements engineering (RE) embodies a wide range of concerns. The objectives to be achieved by the system-to-be must be elicited and analyzed within some organizational or physical context. Such objectives must be operationalized into specifications of services, constraints, and assumptions. The responsibilities for such specifications need to be assigned among the humans, devices, and software forming the system. Requirements emerge from this process as prescriptive assertions on the software-to-be, formulated in the vocabulary of the environment.

The requirements problem has been with us for a long time. Poor requirements were recurrently recognized to be the major cause of project cost overruns, delivery delays, failure to meet expectations, or severe degradations in the environment controlled by the software. In their early empirical study, Bell and Thayer observed that inadequate, inconsistent, incomplete, or ambiguous requirements are numerous and have a critical impact on the quality of the resulting software [Bel76]. Boehm estimated that the late correction of requirements errors could cost up to 200 times as much as correction during

requirements engineering [Boe81]. In his landmark paper on the essence and accidents of software engineering, Brooks stated that "*the hardest single part of building a sofware system is deciding precisely what to build (. . .) the most important function that the software builder performs for the client is the iterative extraction and refinement of the product requirements*" [Bro87]. In her study of software errors in NASA's Voyager and Galileo programs, Lutz reported that the primary cause of safety-related faults was errors in functional and interface requirements [Lut93]. More recent studies have confirmed the requirements problem on a much larger scale. A survey over 8000 projects undertaken by 350 US companies revealed that one third of the projects were never completed and one half succeeded only partially, that is, with partial functionalities, major cost overruns, and significant delays. When asked about the causes of such failure executive managers identifed poor requirements as the major source of problems [Sta95]. On the European side, a survey over 3800 organizations in 17 countries similarly concluded that most of the perceived software problems are in the area of requirements specification and requirements management [ESI96].

Requirements engineering is an intrinsically difficult task:

- it covers a wide spectrum of concerns ranging from high-level, strategic objectives to detailed, technical requirements;
- it involves two systems: the system-as-is and the system-to-be - both including software and environment components;
- it involves stakeholders having diverse, partial, and often conflicting concerns;
- it requires hazardous or malicious behaviors in the environment to be anticipated in order to guarantee requirements completeness and system robustness;
- it requires the evaluation of numerous alternative options: alternative refinements of objectives, alternative assignments of responsibilities, alternative resolutions of conflicts, alternative countermeasures to threats, etc.

The RE process must therefore be supported by systematic methods. To be effective a RE method should meet the following requirements.

- The method should be *goal-oriented* in order to ensure that the requirements meet the system's objectives -including security and safety objectives.
- It should be *incremental* and support early analysis of partial models - the later errors such as omissions, inadequacies, inconsistencies, and imprecisions are found, the more costly their repair is.
- The method should be *constructive* in order to provide analyst guidance and ensure high-quality requirements by construction.
- It should be *model-based* to support abstraction from details and specification structuring. The model should integrate the multiple system facets and support a variety of analyses.
- The method should mix declarative and operational styles of specification as needed.
- It should be formal when and where needed, and lightweight for usability in practical situations.

This paper overviews a RE method addressing these objectives. The method, known as *KAOS*, has been developed and refined for more than fifteen years of research, tool development, and experience in multiple industrial projects. (KAOS stands for "Keep All

Objectives Satisfied".) The details on the modeling notations, model building method, and model analysis techniques can be found in [Lam07].

Section 2 introduces a modeling framework that integrates multiple views of the system-to-be: goals and their refinements; hazards and threats to safety and security goals, respectively; conceptual objects which the goals refer to, together with their interrelationships; operations to ensure that the goals are satisfied; agents responsible for the goals, their behaviors, and interaction scenarios. Section 3 outlines how such a multi-view model can be constructed in a systematic way.

Critical model components should be formalized to enable formal reasoning about them. Section 4 briefly reviews some basics of real-time linear temporal logic for specifying goals, domain properties, hazards, and threats; goal-structured pre- and postconditions for specifying operations; and specification patterns for lightweight specification.

The next sections then discuss various formal reasoning techniques to support the following RE-specific tasks:

- refine goals and check the correctness of refinements (Section 5);
- operationalize fine-grained goals into operations and check the correctness of such operationalizations (Section 6);
- analyze safety hazards by generating obstacles to goal satisfaction and resolving them (Section 7);
- analyze security threats by generating malicious plans to break security goals, and countermeasures to address these (Section 8);
- analyze conflicts among stakeholder goals, and resolve them (Section 9);
- generate system behavior models inductively from interaction scenarios and goal specifications (Section 10).

2. A multi-view modeling framework for requirements engineering

The multiple facets of the target system are captured through complementary models:

- a goal model interrelates all intentional aspects;
- an object model defines the structural aspects;
- an agent model defines the system components, their interfaces and responsibilities;
- an operation model defines the functional services in relation with the system goals;
- a behavior model captures agent behaviors in terms of interaction scenarios and parallel state machines;
- obstacle and threat models capture unexpected ways of breaking system goals, including security goals, through incidental or malicious behaviors of environment agents.

We briefly review these models sucessively.

2.1. Modeling system goals

A *goal* is a prescriptive statement of intent [Dar93], [Lam00a]. It expresses some objective to be achieved by the system. The latter comprises the software *and* its environment.

For example, *"train doors shall be closed while the train is moving"* is a goal requiring some cooperation among the software train controller and train sensors and actuators.

Unlike goals, *domain properties* are descriptive statements about the environment, for example, *"a train is moving iff its physical speed is non-null"*.

Goals are defined at different levels of abstraction. Higher-level goals capture global, business-specific objectives, e.g., *"50% increase of transportation capacity"*. Lower-level goals capture local, technical objectives, e.g., *"train acceleration commanded every 3 secs"*.

There are different types of goals. *Functional goals* prescribe intended behaviors declaratively, e.g., *"passengers transported to their destination"*. They are used for building operational models such as use cases, state machines, and the like. *Quality goals* (sometimes called "non-functional goals") refer to non-functional concerns such as security, safety, accuracy, usability, performance, cost, or interoperability, in terms of application-specific concepts. Some of the quality goals are *softgoals*; they cannot be established in clear-cut sense. Softgoals capture preferred behaviors; they are used to compare alternative options [Myl92], [Chu00]. Non-soft goals prescribe sets of admissible behaviors.

Goal satisfaction requires agent cooperation. For example, the high-level goal *"safe train transportation"* requires the cooperation of agents such as the software train controller, the train tracking system, the train driver, passengers, etc. An *agent* is an active system component responsible for goal achievement. Agents refer to roles rather than individuals.

The finer-grained a goal is, the fewer agents are required for its satisfaction. A *requirement* is a goal assigned to a single agent in the software-to-be. For example, *"doorState = 'closed' while measured speed is non-zero"* is a requirement on the train controller. An expectation is a goal assigned to a single agent in the software environment. For example, *"passengers exit train when doors are open at their destination"* is an expectation. Expectations are sometimes called assumptions; unlike requirements they cannot be enforced by the software-to-be.

One important, often neglected part of the requirements engineer's job is to provide *satisfaction arguments* [Lam00a], [Ham01]. These take the form

$$R, E, D \vdash G,$$

meaning "in view of properties *D* of the domain, the requirements *R* satisfy goal *G* under expectations *E*".

Goals provide a criterion for requirements completeness and pertinence [Yue87]. Let REQ, EXPECT, and DOM denote a set of requirements, expectations, and domain properties, respectively.

A requirements set REQ is *complete* if for all identified goals G:

$\{REQ, EXPECT, Dom\} \vdash G$

A requirement r in REQ is *pertinent* if for some identified goal G:

r is used in a satisfaction argument $\{REQ, EXPECT, Dom\} \vdash G$

Note thus that requirements completeness and pertinence is relative to known domain properties and the identified goals and expectations.

A *goal model* shows contribution links among goals. It is represented by an AND/OR refinement graph whose nodes represent goals and edges represent AND/OR refinement links. In this graph, a goal *G* is *AND-refined* into subgoals *G1, G2, . . . , Gn* iff satisfying *G1, G2, . . . , Gn* contributes to satisfying *G*. (A more precise definition is

given in Section 5.) The set {G1, G2, ..., Gn} is called refinement of *G*. A goal *G* is *OR-refined* into refinements *R1, R2, ..., Rm* iff satisfying the subgoals of *Ri* is one alternative to satisfying G ($1 \leq i \leq m$). *Ri* is called an alternative for *G*. Fig. 1 shows a goal model fragment for our train control system.

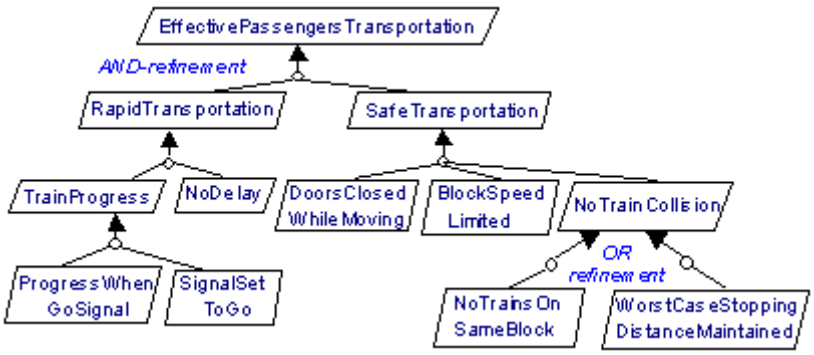

Figure 1. Portion of a goal graph in a train control system [Lam07]

Goal models are built using a variety of elicitation techniques. Preliminary goals are identified by analyzing the problems and deficiencies in the system-as-is, and by searching intentional and prescriptive keywords in available raw material and interview transcripts. More abstract, coarse-grained goals are then obtained bottom-up by asking *WHY* questions about available goals and operational material such as scenarios [Jar98]. In parallel, more concrete, fine-grained goals are obtained top-down by asking *HOW* questions about available goals. Goals are also derived by use of refinement patterns (see Section 5), by resolution of obstacles (see Section 7), and by exploration of countermeasures to security threats (see Section 8).

In this model elaboration process, goal refinement terminates when fine-grained subgoals are obtained that can be assigned as requirements or expectations to software or environment agents, respectively [Dar93]. Goal abstraction terminates when the system boundary is reached, that is, the more abstract supergoals cannot be satisfied under the sole responsibility of the agents forming the 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 in a real-time temporal logic (see Section 4), the goal's priority level, etc.

2.2. Modeling system objects

The object model provides a structural view of the target system. A conceptual object is a thing of interest in the system whose instances can be distinctly identified, share similar features, and have a specific behavior from state to state. An object is modeled as an *entity*, *association*, or *event* dependent on whether it is an autonomous, subordinate, or instantaneous object, respectively. The object model is represented by an operation-free, design-independent UML class diagram.

Such diagram can be systematically derived from the goal model [Lam00a]. Each goal formulation is analyzed to extract the entities, associations, and attributes the goal

refers to. For example, a goal *"avoid multiple trains on the same block"* gives rise to *"Train"* and *"Block"* entities and an *"On"* association. The goal *"train speed shall not exceed the speed limit of the block which the train is on"* gives rise to a *"speedLimit"* attribute of *"Block"*, etc.

Contrarily to what is is often confessed in the UML literature, no "hocus pocus" is required here to obtain a "good" object model; goal-directed construction guarantees a complete and pertinent object model.

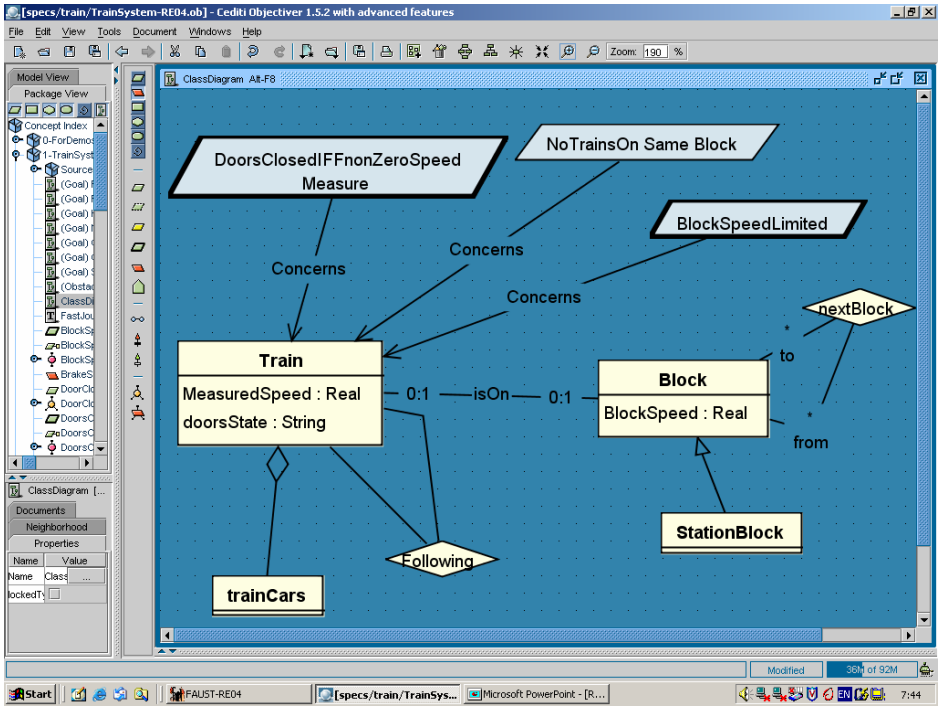

Figure 2. Modeling objects referred to by goals [Lam07]

Fig. 2 illustrates an object model fragment for our train control system. The nodes in an object model are decorated by annotations to characterize the corresponding object - such as its precise definition, domain properties associated with the object (that can be optionally specified in real-time temporal logic), etc.

2.3. Modeling system agents

The agent model defines the responsibilities and interfaces of the various agents. As introduced before, an agent is a software, device, or human component of the system that plays some specific role in goal satisfaction. It controls behaviors by performing operations (see Section 2.4). Agents run concurrently with each other.

An agent is modelled by responsibility links to goals and by monitoring/control links to object attributes and/or associations from the object model. Monitoring/control links capture the agent's interface through the state variables it monitors and controls in its

own environment [Par95]. A *state variable* is an attribute or association of some object. Each state variable is controlled by a single agent.

An agent responsible for some goal must restrict system behaviors [Fea87]. The goal must be realizable by the agent [Let02a]. A goal *G* is *realizable* by agent *ag* iff :

- (intuitively:) given ag's *monitoring & control* capabilities it is possible for *ag* alone to satisfy *G* without more restrictions than required by G;
- (more formally:) there exists a transition system $TS_{ag} = (Init, Next)$ on the state variables monitored and controlled by *ag* such that $RUN(TS_{ag}) = HISTORIES(G)$, that is, the set of agent runs equals the set of behaviors prescribed by the goal.

There can be multiple causes for goal unrealizability, namely, (a) lack of monitorability of variables to be evaluated in the goal formulation, (b) lack of controllability of the variables constrained by the goal, (c) need to evaluate variables in future states, (d) conditional goal unsatisfiability, or (e) reference to a target condition to be achieved in unbounded future. This taxonomy of unrealizability problems gives rise to goal refinement tactics for resolving unrealizability [Let02a]. The latter are encoded as refinement patterns (see Section 5).

In an agent model, OR-assignment links allow us to represent alternative assignments of the same goal to different agents. Alternative software-environment boundaries can thereby be captured and assessed with respect to softgoals [Chu00] so as to select a "best" responsibility assignment.

Responsibility assignments also provide a basis for simple forms of load analysis. Fig. 3 shows the responsibilities of an overloaded air traffic controller. This view was generated from a corresponding agent model using a query/visualization tool on the model database.

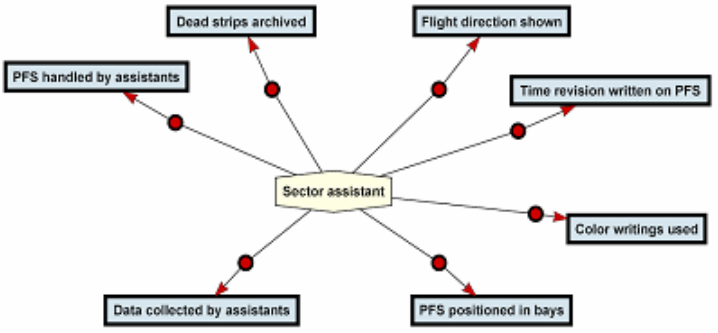

Figure 3. Load analysis [Lam07]

2.4. Modeling system operations

The operation model provides a functional view of the target system in terms of the services to be provided.

An operation *Op* is a relation: $Op \subseteq InputState \times OutputState$. It must operationalize some underlying goals from the goal model; this entails a proof obligation (see Section 6).

Operation applications yield state transitions and corresponding events. They are atomic; an input state is mapped to a state at next smallest time unit. (Operations with duration are represented through start/end events.) They can be concurrent with others.

In an operation model, operations are connected to goals via operationalization links, to objects via input/output links, and to agents via performance links. UML use case models can easily be generated from such models.

Each operation in an operation model is specified by a pair of conditions (*DomPre*, *DomPost*) where:

- *DomPre* is a *descriptive* condition that fully characterizes the class of input states of the operation in the domain,
- *DomPost* is a *descriptive* condition that fully characterizes the class of output states of the operation in the domain.

An operationalization of a goal *G* into operation *Op* is further specified by a triple of conditions (*ReqPre*, *ReqTrig*, *ReqPost*) where:

- *ReqPre* is a prescriptive necessary condition on *Op*'s input states to ensure *G*;
- *ReqTrig* is a prescriptive sufficient condition on *Op*'s input states to ensure *G*; it requires immediate application of *Op* provided *DomPre* holds;
- *ReqPost* is a prescriptive condition on *Op*'s output states to ensure *G*.

As an operation may contribute to multiple goals, it can have multiple required preconditions, trigger conditions, and/or postconditions. The global precondition for the operation to be applied is

$$Pre = DomPre \wedge \bigwedge_i ReqPre_i$$

The global postcondition when the operation is applied is

$$Post = DomPost \wedge \bigwedge_j ReqPost_j$$

The global trigger condition forcing the operation to be applied is

$$Trig = \bigvee_k ReqTrig_k$$

The specifier must always ensure the following consistency rule:

$$\bigvee_k ReqTrig_k \wedge DomPre \Rightarrow \bigwedge_i ReqPre_i$$

In our train control example, the operation for opening train doors might be specified as follows:

Operation OpenDoors
Def *Operation controlling the opening of all train doors*
Input Train, **Output** Train/DoorsState

DomPre *The train doors are closed*
DomPost *The train doors are open*
ReqPre For DoorsClosedWhileNonZeroSpeed
The train's measured speed is 0
ReqPre For SafeEntry&Exit
The train is at some platform
ReqTrig For NoDelayToPassengers
The train has just stopped

A corresponding formal version can optionally be specified as well (see Section 5). The distinction between domain and required conditions is important. Unlike in most specification languages, we are not confusing descriptions and prescriptions. Prescriptions may be assessed, negotiated, and replaced by alternatives; descriptions may not. Moreover, *traceability* between operations and their underlying goals is thereby supported.

2.5. Modeling obstacles to goals

The goals identified in the early stages of the RE process are often too ideal. They are likely to be violated due to unexpected or malicious agent behaviors. For system robustness and requirements completeness, it is essential to detect and resolve such "overoptimism" at RE time - especially in the case of mission-critical systems.

An obstacle *O* to goal *G* is a goal violation precondition satisfying the three following conditions:

1. O, Dom $\models \neg$ G — *obstruction*
2. Dom $\not\models \neg$ O — *domain consistency*
3. There exists a behavior *E* of the environment of the set of agents in charge of *G* such that $E \models O$ — *feasibility*

Hazards and threats are obstacles obstructing safety and security goals, respectively.

An *obstacle model* is a set of goal-anchored fault trees where each fault tree is an AND/OR refinement tree showing how the goal can be violated. The root of the tree is the goal negation; the leaves are elementary obstruction conditions that are consistent with the domain and satisfiable by the environment.

Obstacle resolution then consists in overcoming the sub-obstacles through various resolution tactics such as *goal weakening*, *goal substitution*, *agent substitution*, *obstacle mitigation*, and so forth [Lam00b]. Such resolution yields new or deidealized goals, resulting in a more complete set of requirements for a more robust system.

Fig. 4 shows an obstacle OR-refinement tree showing how the goal "*train stopped if signal set to 'stop'*" could be broken. The new goal of regularly sending out responsiveness checks to train drivers emerges there as a resolution to the sub-obstacle in the middle. The leaf obstacles in Fig. 4 occurred in various reported train accidents (see ACM's Risks forum.)

2.6. Modeling security threats

Threat models are augmented obstacle models where:

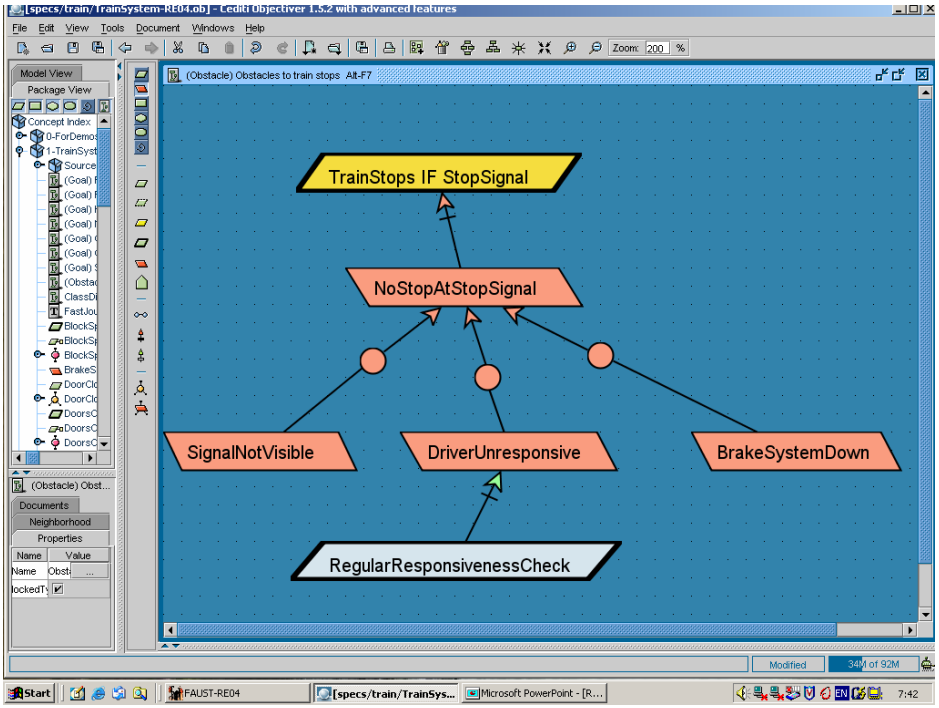

Figure 4. Portion of an obstacle model with new goal as resolution [Lam07]

- the root goal negation refers to a security goal,
- the obstacles are malicious obstacles (called *threats*),
- the refinement graph is extended with the attacker's anti-goals,
- the refinement terminates when leaf conditions are reached that can be monitored and controlled by the attacker.

Such models can be built systematically [Lam04a], and even automatically under certain restrictions [Jan06], see Section 8. They provide the basis for enriching the goal model with countermeasures to the identified threats.

2.7. Modeling agent behaviors

The agent behaviors are modelled by interaction scenarios at instance level and by parallel state machines at class level.

A *scenario* is a historical sequence of interaction events among agent instances. It illustrates some way of achieving a goal G; the scenario is a sub-history in the set of admissible behaviors prescribed by G. An interaction event corresponds to an application of some operation by a source agent, notified to a target agent.

Scenarios can be positive or negative. A *positive* scenario is an example of desired behavior. A *negative* scenario is a counterexample showing some undesired behavior. A scenario may be composed of sub-scenarios, called episodes, which may be common to multiple scenarios.

Scenarios are represented by simple message sequence charts (MSCs), see Fig. 5. Such diagrams capture a partial order on interaction events and, along each agent's timeline, a total order on events.

Scenarios and goals have complementary benefits. Scenarios provide a concrete, narrative way of eliciting requirements from examples and counterexamples. They also give us acceptance test data for free. On the downside, they are inherently partial and raise a coverage problem similar to test cases. They lead to a combinatorial explosion of traces for good coverage. They often entail premature choices such as unnecessary sequencing of events or decisions on the software-environment boundary. Last but not least, they keep the underlying requirements implicit. Scenarios are therefore useful for requirements elicitation and validation whereas goals are required for declarative reasoning (see sections below). Moreover, goal specifications can be inductively synthesized from scenario examples and counterexamples using learning algorithms [Lam98b].

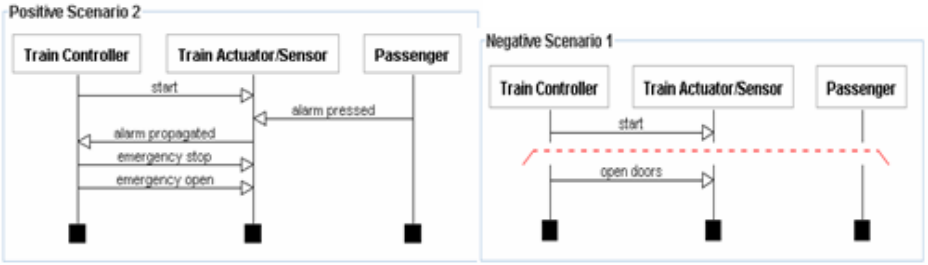

Figure 5. Positive and negative scenarios [Lam07]

At class level, the system's behavior is modelled as a parallel composition of agent behaviors. Each agent is behaviorally modeled by a labelled transition system (LTS), see [Mag06]. Agents thereby behave asynchronously but synchronize on shared events. Fig. 6 shows a LTS model that covers all possible event traces for train controllers in the system. Note that the event trace along the train controller's timeline in the positive scenario in Fig. 5 is covered by a path in the LTS model in Fig. 6.

LTS models have a simple, compositional semantics. They are executable for model animation [Mag00]. They support a variety of analyses including model checking [Gia03]. On the downside, they are hard to build and understand. Section 10 will outline a technique for synthesizing them inductively and interactively from scenarios and goals ([Dam05], [Dam06]).

3. Building the entire system model: a systematic method

In a model-based requirements engineering process, the various system views outlined in the previous section can be elaborated and integrated in a systematic way through the following general steps.

1. *Domain analysis*. Build a goal model for the current system-as-is through *WHY* and *HOW* questions on available material. (Section 5 outlines techniques to support this step.) In parallel, elicit scenarios of doing things in the system-as-is as illustrations of behaviors prescribed by goals.

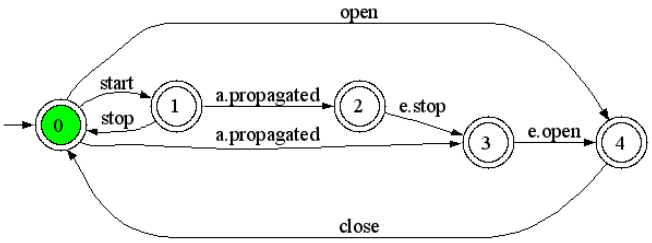

Figure 6. LTS model for train controllers [Lam07]

2. *Domain analysis*. Derive an object model for the system-*as-is* from this goal model.
3. *System-to-be analysis*. Replay Step 1 for the system-*to-be*: based on elicited material about the experienced problems with the system-as-is and emerging opportunities, update and expand the previous goal model. The upper levels of the goal model often remain unchanged as organization-wide business objectives tend to be fairly stable. New, specific features of the system-to-be are developed along OR-branches in the goal graph as alternative ways of meeting the same higher-level goals in view of the elicited problems and opportunities. (Section 5 outlines techniques to support this step.) In parallel, explore new scenarios of doing things in the system-to-be as illustrations of behaviors prescribed by new goals.
4. *System-to-be analysis*. Replay Step 2 for the system-*to-be*: update and expand the previous object model from the new version of the goal model. The basic concepts from the application domain often remain unchanged.
5. *Obstacle and threat analysis*. In parallel with Steps 1-4, build obstacle and threat models, and explore resolutions to enrich and update the goal model. (Sections 7 and 8 outline techniques to support this step.)
6. *Conflict analysis*. In parallel with Steps 1-5, detect conflicts among goals and explore resolutions to enrich and update the goal model. (Section 9 outlines techniques to support this step.)
7. *Responsibility* analysis. Explore alternative assignments of leaf goals to system agents, select best alternatives based on non-functional goals from the goal model [Chu00], and build an agent model.
8. *Goal operationalization*. Build an operation model ensuring that all leaf goals from the goal model are satisfied. (Section 6 outlines techniques to support this step.)
9. *Behavior analysis*. Build a behavior model for the system as parallel composition of behavior models for each component. Animate this model for adequacy checking and feeback from stakeholders [Tra04]. (Section 10 presents techniques to support this step.)

The above steps are ordered by data dependencies. They are often intertwined, with backtracking to previous steps. In particular, behavior models are sometimes built earlier in the process, for understanding the system-as-is and for earlier animation of portions of the system-to-be. See [Dar93], [Lam00a], [Lam07] for heuristics, justifications, and

illustrations of each step above. Industrial experience with this method is reported in [Lam04b].

4. Formal specification of goals, domain properties, and operations

The approach presented 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 main temporal operators used are the following standard ones:

- $\circ$ P: P holds in the next state
- $\Box$ P: P holds in every future state
- P ***W*** N: P holds in every future state unless N holds
- $\Diamond$ P: P holds in some future state
- $\Box_{\leq T}$ P: P holds in every future state up to T time units
- $\Diamond_{\leq T}$ P: P holds within T time units
- P $\Rightarrow$ Q for $\Box$ (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) $\models$ $\Box$ P iff (H, j) $\models$ P for all j $\geq$ i
- (H, i) $\models$ $\Diamond_{\leq T}$ P iff (H, j) $\models$ P for some j $\geq$ i with dist (i,j) $\leq$ T

The $\circ$ /$\bullet$ operators refer to the next/previous state within the smallest time unit. They are often used for expressing immediate obligations.

Here are some examples of formal specifications.

Goal *Maintain* [DoorsClosedWhileNonZeroSpeed]
  FormalSpec $\forall$ tr: Train
    tr.MeasuredSpeed $\neq$ 0 $\Rightarrow$ tr. DoorsState = 'closed'
Goal *Achieve* [FastJourney]
  FormalSpec $\forall$ tr: Train, bl: Block
    On (tr, bl) $\Rightarrow$ $\Diamond_{\leq T}$ On (tr, next(bl))

In goal specifications, the keywords prefixing goal names are used to indicate temporal specification patterns ([Dar93], [Dwy99]) - e.g., *Achieve [P]* indicates a pattern $\Diamond_{\leq T}$ *P* on a target predicate P; *Avoid [P]* indicates a pattern $\Box \neg$ *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 operation of controlling the opening of train doors is formally specified as follows:

Operation OpenDoors
Input tr: Train; **Output** tr: Train/DoorsState
DomPre tr. DoorsState = 'closed'
DomPost tr. DoorsState = 'open'
ReqPre for *DoorsClosedWhileNonZeroSpeed*: tr.MeasuredSpeed = 0
ReqPre for *SafeEntry&Exit*: (∃ pl: Platform) At (tr, pl)
ReqTrig For *NoDelayToPassengers*: @(tr.MeasuredSpeed = 0)

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. (Such variables are attributes and/or associations from the object model, see Section 2.3.) 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. As introduced in Section 2.4, 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) [Del03], [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 non-soft 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.

As opposed to generative semantics of operational languages such as Statecharts or VDM, where every state transition is forbidden except the ones explicitly required by the specification, we have a *pruning semantics* here: every state transition is allowed except the ones explicitly forbidden by the specification. With a generative semantics, operations are viewed as generating the set of admissible behaviors of the system; these

cover the only possible transitions. As a consequence, generative semantics have a built-in assumption that nothing changes except when an operation specification explicitly requires it; the specifier is relieved from explicitly specifying what does *not* change - in other words, a generative semantics avoids the *frame problem* [Bor93]. Such built-in frame assumption, however, makes it difficult to support incremental reasoning about partial models [Jac95]. With a pruning semantics (like in Z, LARCH, or other temporal logic-based formalisms), the specification prunes the set of admissible system behaviors. Incremental elaboration and reasoning through composition of partial models is then made possible. The price to pay is the need for handling the frame problem. In our case, we introduce two built-in axioms within our semantics to relieve the specifier from explicitly stating everything that does not change.

Frame axiom 1: Any state variable not declared in the output clause of the specification of an operation is left unchanged by any application of this operation.

This frame axiom is enforced by requiring the *DomPost* and *ReqPost* conditions of an operation to refer only to those state variables which are explicitly declared in the output clause of the operation (in a way similar to LARCH).

Frame axiom 2: Every state transition that satisfies the domain pre- and postconditions of an operation corresponds to an application of this operation:

for any operation *op*:
DomPre (op) $\wedge\circ$ DomPost (op) $\Rightarrow$ Performed (op)

5. Checking goal refinements

A first kind of RE-specific model verification consists in checking that the refinements of non-soft goals in the goal model are correct and complete. Such checking is important as missing subgoals result in incomplete requirements.

We first need a more precise definition of what it means for a goal refinement to be correct.
A set of goals {G1, ..., Gn} correctly refines a goal *G* in a domain theory *Dom* iff

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

Several approaches can be followed to verify the correctness of a goal refinement.

Approach 1: Theorem proving. We might use a temporal logic theorem prover - such as STeP, for example [Man96]. This is obviously a heavyweight approach requiring the assistance of an expert user. Moreover we get no real clue in case the verification fails.

Approach 2: 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 the STeP theorem prover. They are then reused in matching situations through instantiation of their meta-variables. Fig. 7 shows two frequent refinement pat-

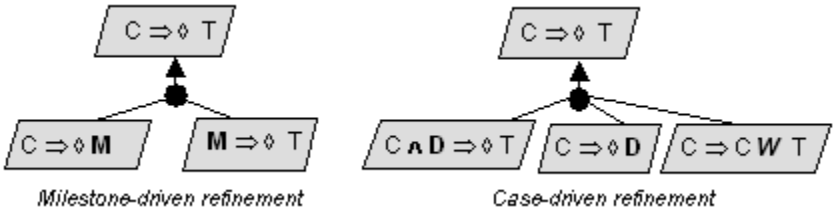

Figure 7. Formal refinement patterns

terns. The first pattern encodes the tactics of introducing an intermediate milestone goal whereas the second pattern encodes a standard case analysis pattern.

Fig. 8 illustrates the use of the case-driven pattern from Fig.7 in a situation where a refinement of the parent goal `Achieve[TrainProgress]` into the two left subgoals `Achieve[ProgressWhenGo]` and `Achieve[SignalSetToGo]` is being checked. An incomplete refinement is detected by pattern matching. The match reveals the missing subgoal, indicated by a dashed line in the instantiated refinement, namely, that the train must be waiting on its current block until it moves to the next block.

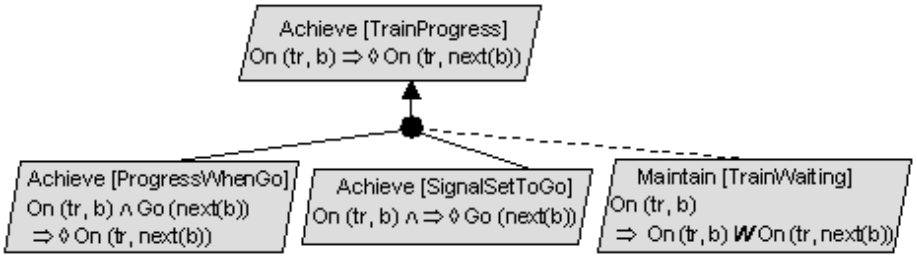

Figure 8. Pointing out missing subgoal through pattern instantiation [Lam07]

Refinement patterns support a constructive approach to refinement correctness. When a goal is being partially refined, we can retrieve all matching patterns from the catalogue and thereby explore alternative ways of completing the partial refinement [Dar96]. Fig. 9 illustrates that point. Three alternative subgoals appear as possible response to the refinement query on the left-hand side. Once instantiated the three returned alternatives should be assessed with respect to the application's non-functional goals to select the one that meets them best ([Chu00], [Let04]).

Another benefit of refinement patterns is the formal correctness proof of the instantiated refinement that we get for free. Each pattern in the catalogue is proved once

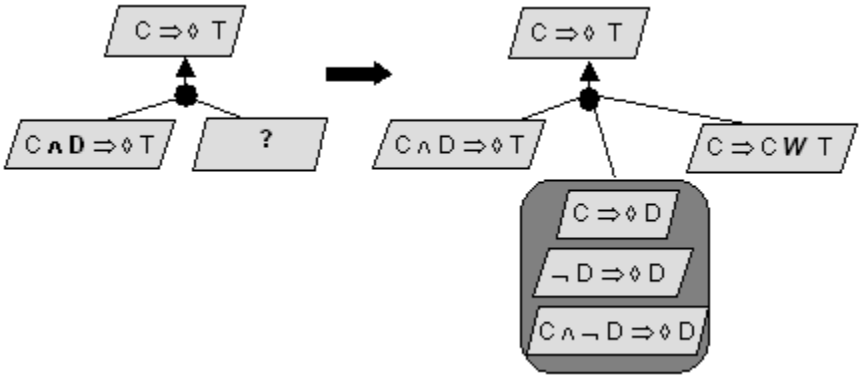

Figure 9. Generating alternative refinements [Lam07]

for all. For example, the proof of the case-driven pattern in Fig. 7 looks like this:

1. C ⇒ ◇ D	Hyp
2. C ∧ D ⇒ ◇ T	Hyp
3. C ⇒ C ***W*** T	Hyp
4. C ⇒ (C ***U*** T) ∨ □ C	3, def of Unless
5. C ⇒ ◇ T ∨ □ C	4, def of Until
6. C ⇒ ◇ D ∧ (◇ T ∨ □ C)	1, 5, strengthen consequent
7. C ⇒ (◇ D ∧ ◇ T) ∨ (◇ D ∧ □ C)	6, distribution
8. C ⇒ (◇ D ∧ ◇ T) ∨ ◇(D ∧ C)	7, trivial lemma
9. C ⇒ (◇ D ∧ ◇ T) ∨ ◇◇ T	8, 2, strengthen consequent
10. C ⇒ (◇ D ∧ ◇ T) ∨ ◇ T	9,◇-idempotence
11. C ⇒ ◇ T	10, absorption

Instead of having to redo such tedious proofs at every goal refinement when we build the goal model for the application, we get a proof when using a pattern just by instantiating the generic proof accordingly.

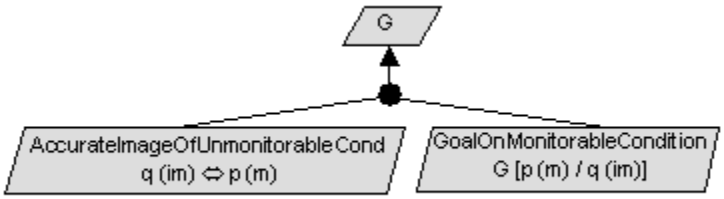

Figure 10. "Introduce accuracy subgoal" pattern

Some of the refinements in the pattern catalogue might be seen as high-level RT-LTL inference rules. Others are specifically aimed at refining goals towards subgoals that are realizable as defined in Section 2.3. They introduce finer-grained subgoals to resolve unrealizability problems [Let02a]. Fig. 10 shows one such pattern. The root goal *G* there

involves a condition *p(m)* on a variable *m* unmonitorable by the agent candidate for responsibility assignment. To resolve this unmonitorability, a monitorable "image" variable *im* and condition *q(im)* are introduced under the constraint that they must accurately reflect their unmonitorable counterpart.

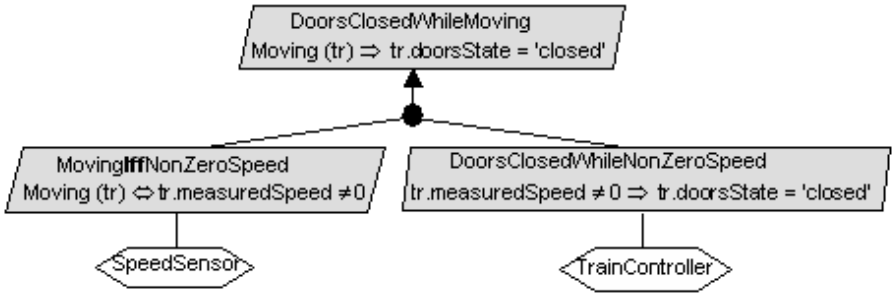

Figure 11. Using the "Introduce accuracy subgoal" pattern

Fig. 11 illustrates the use of this pattern on our running example. This example also suggests how such patterns are helpful for producing agent assignments as well.

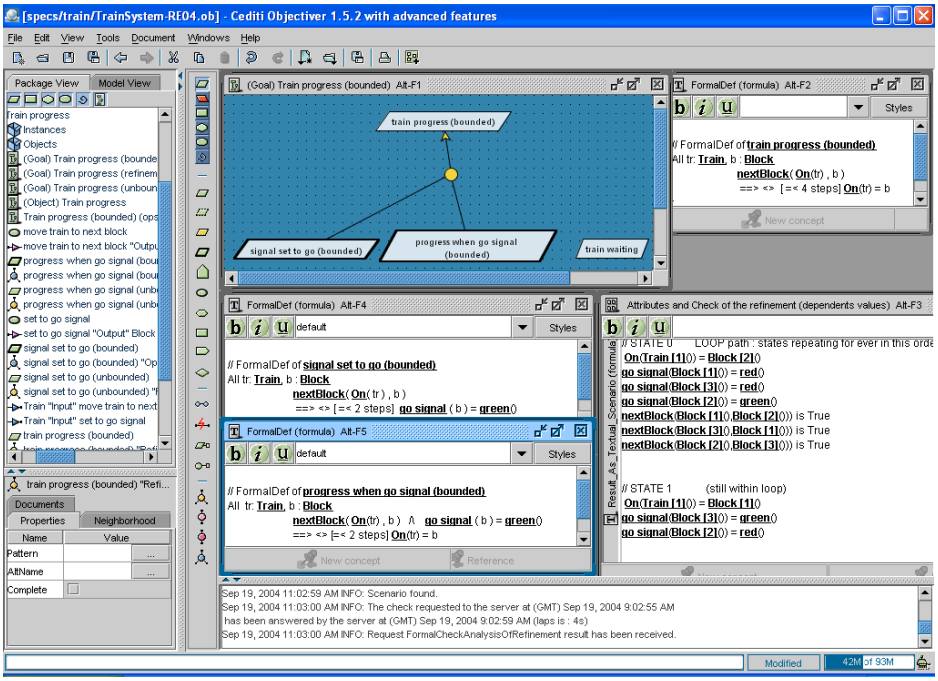

Figure 12. Roundtrip use of a bounded SAT solver for checking goal refinements

Approach 3: Bounded SAT solver. Beside using a theorem prover or a catalogue of formal refinement patterns, we can also make a roundtrip use of a bounded SAT solver. In view of the above definition of correct refinement of goal *G* into subgoals $G_1, \ldots, G_n$, we would like to know whether the temporal logic formula

$G_1 \wedge \ldots \wedge G_n \wedge Dom \wedge \neg G$

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

To achieve this, we can build a front-end that (a) asks the user to instantiate the above formula to selected object instances, in order to obtain a propositional formula, (b) translates the result into the input format required by the SAT solver, (c) asks the user to determine a maximal length to bound counterexample traces, (d) runs the SAT solver, and (e) translates the output back to the level of abstraction of the input model.

Fig. 12 shows the result produced by the FAUST tool [Pon04] on the incomplete refinement suggested in Fig. 8. The counterexample generated on the right lower window is a scenario showing the train getting back to the previous block thereby suggesting the missing subgoal of the train waiting on its current block until the signal is set to "go".

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. A bounded universe, however, makes it possible to show the presence of bugs in a goal model, not their absence.

6. Deriving goal operationalizations

Another kind of RE-specific model verification consists in checking the correctness of operationalizations of goals from the goal model into specifications of operations from the operation model. Such checking is important too; we must make sure that the operational specifications meet the intentional ones.

To perform such checks formally we first need a temporal logic semantics for operations [Let02b]. Such semantics is easily provided from the definition of pre-, post-, and trigger conditions in Section 2.4 and the semantic considerations in Section 4. Let *op* denote an operation from the operation model, and let

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

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

If $R \in \text{ReqPre}(op)$ **then**
$[\!|\ R\ |\!] =_{def} (\forall \star)\ ([\!|\ op\ |\!] \Rightarrow R)$
If $R \in \text{ReqTrig}\ (op)$ **then**
$[\!|\ R\ |\!] =_{def} (\forall \star)\ (R \wedge \text{DomPre}(op) \Rightarrow [\!|\ op\ |\!])$
If $R \in \text{ReqPost}(op)$ **then**
$[\!|\ R\ |\!] =_{def} (\forall \star)\ ([\!|\ op\ |\!] \Rightarrow\ \circ R)$

Next we need a more precise definition of what it means for a goal to be correctly operationalized into operational specifications.

A set of required conditions $R_1, \ldots, R_n$ on operations from the operation model correctly operationalizes a goal *G iff*

$[\!|\ R_1\ |\!] \wedge \ldots \wedge [\!|\ R_n\ |\!] \models \text{G}$ *completeness*
$[\!|\ R_1\ |\!] \wedge \ldots \wedge [\!|\ R_n\ |\!] \not\models \textbf{false}$ *consistency*
$\text{G} \models [\!|\ R_1\ |\!] \wedge \ldots \wedge [\!|\ R_n\ |\!]$ *minimality*

Every operationalization defines a proof obligation. Several approaches can be followed to verify the correctness of a goal operationalization.

Approach 1: Bounded SAT solver. Like for checking goal refinements, we can make a roundtrip use of a bounded SAT solver. We would now like to know whether the temporal logic formula

$$[|\ R_1\ |] \wedge \ldots \wedge [|\ R_n\ |] \wedge \text{Dom} \wedge \neg\ \text{G}$$

is satisfiable and, if so, find a historical sequence of states satisfying it. The FAUST toolset proceeds similarly to check bounded operationalizations and generate counterexample traces [Pon04].

Approach 2: Formal operationalization patterns [Let02b]. The principle is similar to goal refinement patterns. A catalogue of operationalization patterns is built and formally proved correct (e.g., using the STeP theorem prover). The patterns cover common goal specification patterns [Dwy99], e.g., *Achieve* goals of form $C \Rightarrow \diamond_{\leq d} T$ or $C \Rightarrow \circ T$, and *Maintain* goals of form $C \Rightarrow T$, $C \Rightarrow \ \Box\, T$, $C \Rightarrow T\ W\ N$, or $T \Rightarrow\ \bullet\, C$. The patterns are then reused in matching situations through instantiation of their meta-variables. Fig. 13 shows a pattern for operationalizing *Immediate Achieve* goals. If we apply it to the following safety goal on train signals:

$\forall$ b: Block

[($\exists$ tr: Train) On (tr, b)] $\Rightarrow\ \circ \neg$ GO (b)

we obtain two operations, SetSignalToStop(b) and SetSignalToGo(b), say, with trigger condition *($\exists$ tr: Train) On (tr, b)* on the operation SetSignalToStop(b) and a required precondition $\neg$ *($\exists$ tr: Train) On (tr, b)* on the operation SetSignalToGo(b).

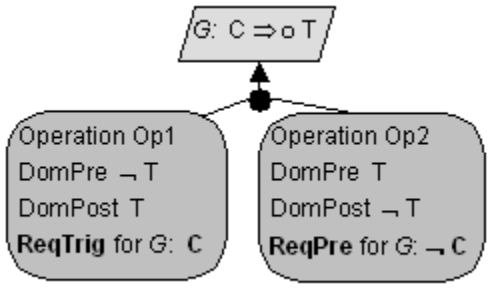

Figure 13. Operationalization pattern for *Immediate Achieve* goals

7. Obstacle analysis for mission-critical systems

Obstacle models were introduced in Section 2.5 as a means for anticipating what could go wrong in an overideal system. Goal completeness is increased through countermeasures to obstacles. This section overviews how obstacle analysis can be made further formal, in particular, through a calculus for generating obstacles from goals [Lam00b].

An overall procedure for obstacle analysis looks like this:

for every leaf goal in the goal refinement graph (requirement or expectation):

(a) identify as many obstacles to it as possible;
(b) assess their feasibility, likelihood, and severity;
(c) resolve the feasible ones according to their likelihood and severity.

Our focus here will be on steps (a) and (c). We discuss them successively.

7.1. Generating obstacles abductively from goal specifications

For a goal *G*, we are looking for feasible conditions *O* such that:

$\{O, Dom\} \vdash \neg G$

$Dom \not\vdash \neg O$

(see Section 2.5). We may proceed as follows :

- negate *G*;
- find as many AND/OR refinements of $\neg$ *G* as possible in view of properties in *Dom*,
- until obstruction preconditions are reached that are satisfiable by the environment of the set of agents assigned to *G*.

This amounts to constructing a *goal-anchored* fault-tree [Lev95] in a systematic way. To proceed more formally we need more precise definitions first.

A set of obstacles O1, ..., On is a *correct refinement* of some obstacle O in a domain theory Dom iff

$\{O_1, \ldots, O_n, Dom\} \models O$	refinement completeness
$\{O_1, \ldots, O_n, Dom\} \not\models$ **false**	domain consistency
$\{\bigwedge_{j \neq i} O_j, Dom\} \not\models O$ for each i $\in$ [1..n]	minimality

A set of obstacles $O_1, \ldots, O_n$ to some goal *G* is *domain-complete* iff

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

Note that the notion of obstacle completeness is relative to what we know about the domain.

The obstacle trees we want to build should produce correct refinements of the goal negation; the leaf goals must be satisfiable by the environment of the set of agents assigned to the goal and should, ideally, form a domain-complete set of obstacles.

To generate such obstacles abductively from the goal negation and the set of known domain properties, we may follow two approaches [Lam00b].

Approach 1 : Regression of the goal negation through the domain theory. This amounts to calculating preconditions for deriving $\neg G$ from *Dom*. Assuming domain properties to take the general form $A \Rightarrow C$, the procedure is as follows:

Initial step:
  take O := $\neg$ G
Inductive step:
  let A $\Rightarrow$ C be the domain property selected,
    with *C* matching some *L* in *O* whose occurrences are all positive in *O* [Man92]
  then μ := mgu (*L*, *C*) (mgu: most general unifier)
    O := O [L / A . μ]

Every iteration of the inductive step produces finer sub-obstacles. This technique is a counterpart, for declarative statements, of Dijkstra's precondition calculus [Dij76]. A variant of it has been used for long in AI planning [Wal77].

Let us illustrate this calculus on the generation of a well-known obstacle that caused a major accident during aircraft landing at Warsaw airport [Lad95].

We provide some context first. One goal for control of landing states (in simplified form):

MovingOnRunway $\Rightarrow$ $\circ$ ReverseThrustEnabled

As the autopilot software cannot monitor the variable MovingOnRunway, we apply the "*Introduce Accuracy Subgoal*" refinement pattern in Fig. 10 to produce the following subgoals:

MovingOnRunway $\Leftrightarrow$ WheelsState = 'turning'

WheelsState = 'turning' $\Rightarrow$ $\circ$ ReverseThrustEnabled

The second subgoal is a requirement on the autopilot software. The first subgoal is refined in the following assertions:

MovingOnRunway $\Leftrightarrow$ WheelsTurning

WheelsTurning $\Leftrightarrow$ WheelsState = 'turning'

The second assertion is an expectation on the wheel sensor. The first assertion states two assumptions made about the domain. The first says:

MovingOnRunway $\Rightarrow$ WheelsTurning

Let us try to break this assumption for obstacle analysis. We start by negating it:

$\Diamond$ MovingOnRunway $\wedge$ $\neg$ WheelsTurning

We refine this negation by regressing it through the domain. We look for, or elicit, domain properties that are *necessary conditions* for the target condition *WheelsTurning* in the assumption we want to obstruct. We might find, in particular,

WheelsTurning $\Rightarrow$ WheelsOut,

WheelsTurning $\Rightarrow$ $\neg$ WheelsBroken,

WheelsTurning $\Rightarrow$ $\neg$ Aquaplaning, etc.

Let us select the third domain property, equivalent to its contraposition:

Aquaplaning $\Rightarrow$ $\neg$ WheelsTurning

The consequent of this implication unifies with one of the conjuncts in the negated assumption above. We may therefore regress that negated assumption backwards through this domain property which yields the following subobstacle obstructing the target assumption:

$\Diamond$ MovingOnRunway $\wedge$ Aquaplaning

The generated obstacle is satisfiable by the environment (in this case, mother Nature). It was indeed satisfied during the Warsaw crash.

As the above derivation suggests, obstacle analysis may be used to elicit unknown domain properties as well.

Approach 2 : Formal obstruction patterns. We can again build a catalogue of common goal obstruction patterns, prove each of them, and reuse them by instantiation in matching situations [Lam00b]. Fig. 14 shows a very common obstruction pattern. The pattern encodes a single regression step. The above derivation can thus be seen as an application of this pattern as well.

Once generated, the obstacles need to be assessed for feasibility, likelihood and severity. For feasibility, a SAT solver might be used to check that the obstacle assertion is satisfiable by the environment. For likelihood and severity, standard risk management techniques should be used.

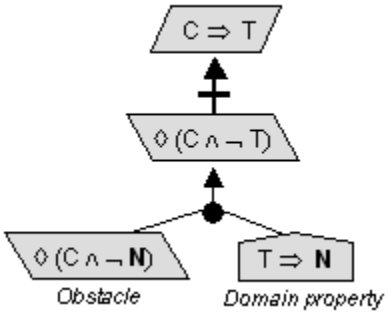

Figure 14. Goal obstruction pattern

7.2. Resolving obstacles

The generated obstacles, if feasible and likely, must be resolved through countermeasures. Resolution may be undertaken at requirements engineering time or deferred to system runtime through obstacle monitoring [Fea98]. For resolution at RE time, we may (a) explore alternative resolutions by application of model transformation operators [Lam00b], and then (b) select a "best" resolution based on the likelihood and severity of the obstacle, and on other non-functional goals from the goal model [Chu00].

Model transformation operators encode various resolution tactics such as the following.

- *Goal substitution*: Consider alternative refinements of the parent goal to avoid the obstruction of a child goal - e.g., replace the obstructed subgoal MotorReversedIffWheelsTurning by the goal MotorReversedIffPlaneWeightSensed.
- *Agent substitution*: Consider alternative responsibility assignments for the obstructed goal - e.g., replace the agent OnBoardTrainController, assigned to some obstructed safety-critical goal in the train control system, by the agent VitalStationComputer.
- *Goal weakening*: Weaken the goal formulation - e.g., weaken the goal SectorTrafficControllerOnDuty in an air traffic control system into the goal SectorTrafficControllerOnDutyOrWarningToNextSector.
- *Goal restoration*: Enforce the target condition in the obstructed goal when the obstacle occurs - e.g., generate an alarm to the pilot in case the obstacle WheelsNotOut occurs.
- *Obstacle prevention*: Introduce a new Avoid goal, to be refined in turn, in order to prevent the obstacle from occurring - e.g., introduce in the goal model a new goal Avoid[TrainAccelerationCommandCorrupted].
- *Obstacle mitigation*: Tolerate the obstacle but mitigate its effects - e.g., introduce a new goal Avoid[TrainCollisionWhenOutDatedTrainInfo].

8. Threat analysis for security-critical systems

As introduced in Section 2.6, threats are malicious obstacles obstructing security goals. Threat analysis consists in identifying threats to the system and resolving them through countermeasures. It involves an anti-model, that is, a dual model of threats to the system model. Such model shows how security goals can be obstructed by linking negated goals to the attacker's malicious goals, called *anti-goals*, and capabilities.

The attacker's *capabilities* are captured by two sets of conditions that the attacker can monitor and control, respectively. These capabilities define the interface between the attacker and its own environment, including the threatened software-to-be. The properties of the attacker's environment includes the properties of the software-to-be, including monitorable vulnerabilities to be exploited for anti-goal achievement.

The attacker is a system agent who knows (a) the application's goal model (b) all descriptive domain properties used to build it, and (c) the operation model. In the tradition of the *Most Powerful Attacker* model used in cryptographic protocol analysis ([Kem94], [Low96], [Cla00]), we assume a *Most Knowledgeable Attacker* (MKA) that knows everything about the application model being attacked. Worst-case analysis of threats is required to ensure the completeness of the set of countermeasures to them. The MKA assumption is trivially satisfied here as the attacker at RE time is the application modeller looking for missing countermeasures. Such MKA model also implies that the attacker has no need to dynamically increase its knowledge through observation of system behaviors in response to attacker's stimuli - he has that knowledge already.

An overall procedure for threat analysis looks like this:

1. Build threat graphs rooted on antigoals:
   (a) Get initial anti-goals as roots;
   (b) Identify classes of attackers wishing these, and their capabilities;
   (c) For each root anti-goal and attacker class:
   build an anti-goal refinement graph as a proof that the root anti-goal can be satisfied in view of the attacker's knowledge and capabilities; refinement terminates when leaf conditions are reached that meet the attacker's capabilities.
2. Derive new security goals as countermeasures to counter anti-goals from threat graphs.

We first review the types of security goals that might be threatened by an anti-model together with their specification patterns. Next we will overview and illustrate the various steps of the above procedure. Our main focus will be on formal support for steps (a) and (c).

8.1. Specification patterns for security goals

Security goals prescribe different types of protection of system assets. Numerous taxonomies of security properties are available from the literature - see, e.g., [Kem03]. We can formally specify property classes to define corresponding specification patterns on meta-variables. Instantiating these meta-variables to application-specific, sensitive objects provides candidate security goals for our system, to be refined in the goal model and to be obstructed in an anti-goal model [Lam04a].

For example, the specification pattern for *confidentiality* goals defines confidentiality in a generic way as follows:

Goal *Avoid* [SensitiveInfoKnownByUnauthorizedAgent]
FormalSpec $\forall$ ag: Agent, ob: Object
$\neg$ Authorized(ag, ob.Info) $\Rightarrow \neg$ KnowsV$_{ag}$ (ob.Info)

To specify security goals, our real-time linear temporal logic is augmented with epistemic constructs [Fag95]. In particular, the operator KnowsV$_{ag}$ is defined on state variables as follows:

KnowsV$_{ag}$(v) $\equiv \exists$ x : Knows$_{ag}$(x=v) ("knows value")
Knows$_{ag}$(P) $\equiv$ Belief$_{ag}$(P) $\wedge$ P ("knows property")

The operational semantics of the epistemic operator Belief$_{ag}(P)$ is: "*P is among the properties stored in the local memory of agent ag*". Domain-specific axioms must make it precise under which conditions property *P* does appear and disappear in the agent's memory. An agent thus *knows a property* if that property is found in its local memory and it is indeed the case that the property holds.

In the above pattern for confidentiality goals, the Authorized predicate is a generic predicate to be instantiated through a domain-specific definition. For example, for web banking services we would certainly consider the instantiation Object/Account while searching through the object model for sensitive information to be protected. We might then introduce the following instantiating definition:

$\forall$ ag: Agent, acc: Account
Authorized (ag, acc) $\equiv$ Owner (ag, acc) $\vee$ Proxy (ag, acc) $\vee$ Manager (ag, acc)

Sensitive information about accounts includes the objects Acc# and PIN. The latter are defined in the object model as entities composing the aggregated entity Account and linked through a *Matching* association.

Instantiating the *Confidentiality* specification pattern to this sensitive information yields the following confidentiality goal as candidate for inclusion in the goal model for web banking services:

Goal *Avoid* [AccountNumber&PinKnownByUnauthorized]
FormalSpec $\forall$ p: Person, acc: Account
$\neg$ (Owner (p, acc) $\vee$ Proxy (p, acc) $\vee$ Manager (p, acc))
$\Rightarrow \neg$ [KnowsV$_p$ (acc.Acc#) $\wedge$ KnowsV$_p$ (acc.PIN)]

Other patterns may be defined for specifying and eliciting application-specific instantiations of privacy, integrity, availability, authentication, accountability, or non-repudiation goals, e.g.,

Goal *Maintain* [PrivateInfoKnownOnlyIfAuthorizedByOwner]
FormalSpec $\forall$ ag, ag': Agent, ob: Object
KnowsV$_{ag}$ (ob.Info) $\wedge$ OwnedBy (ob.Info, ag') $\wedge$ ag $\neq$ ag'
$\Rightarrow$ AuthorizedBy (ag, ob.Info, ag')

Goal *Maintain* [ObjectInfoChangeOnlyIfCorrectAndAuthorized]
FormalSpec $\forall$ ag: Agent, ob: Object, v : Value
ob.Info = v $\wedge \circ$ (ob.Info $\neq$ v) $\wedge$ UnderControl (ob.Info, ag)
$\Rightarrow$ Authorized (ag, ob.Info) $\wedge \circ$ Integrity (ob.Info)

Goal *Achieve* [ObjectInfoUsableWhenNeededAndAuthorized]
FormalSpec $\forall$ ag: Agent, ob: Object, v : Value
[Needs (ag, ob.Info) $\wedge$ Authorized (ag, ob.Info)] $\Rightarrow$ $\Diamond_{\leq d}$ Using (ag, ob.Info)

Specifications of application-specific security goals are thus obtained from such patterns by (a) instantiating meta-classes such as Object, Agent, and generic attributes such as *Info*, to application-specific sensitive classes, attributes and associations in the object model; and (b) specializing predicates such as Authorized, UnderControl, Integrity, or Using through substitution by application-specific definitions.

The specification patterns can be diversified through variants capturing different security options. For confidentiality goals, for example, we may consider variants along two dimensions: (a) the degree of approximate knowledge to be kept confidential - exact value of a state variable, or lower/upper bound, or order of magnitude, or any property about the value; and (b) the timing according to which that knowledge should be kept confidential - confidential now, or confidential until some expiration date, or confidential unless/until condition, or confidential forever [Del05].

8.2. Identifying initial anti-goals and attackers

Preliminary anti-goals must be identified as root threats to be refined in threat graphs. One obvious option is to browse the goal model systematically in order to determine whether there are any goal negations that could be wished by malicious agents.

For example, while browsing the goal model for an online shopping system we might stop on the goal stating that every purchased item must have been paid within two days before being sent:

ItemSentToBuyer $\Rightarrow$ $\blacklozenge_{\leq 2d}$ ItemPaidToSeller

(The goal is specified propositionally for simplicity.) The goal negation is:

$\Diamond$ (ItemSentToBuyer $\wedge \neg \blacklozenge_{\leq 2d}$ ItemPaidToSeller)

This goal is obviously going to be wished by a number of malicious shoppers. We should therefore consider it among the root anti-goals for threat graph building.

We can also directly obtain root anti-goals by negating security goal patterns instantiated to application-specific sensitive objects. For example, the negation of the instantiated confidentiality goal *Avoid* [AccountNumber&PinKnownByUnauthorized] for web banking services yields another initial anti-goal:

AntiGoal *Achieve* [AccountNumber&PinKnownByUnauthorized]
FormalSpec $\Diamond \exists$ p: Person, acc: Account
$\neg$ [Owner (p, acc) $\vee$ Proxy (p, acc) $\vee$ Manager (p, acc)]
$\wedge$ KnowsV$_p$ (acc.Acc#) $\wedge$ KnowsV$_p$ (acc.PIN)

The identification of attacker classes is obviously intertwined with the identification of initial anti-goals; the negation of an application-specific goal raises the question of who might benefit from it. We may also use attacker taxonomies available from the literature to identify attackers.

For example, by asking who could benefit from the anti-goal *Achieve* [AccountNumber&PinKnownByUnauthorized] we could elicit agent classes such as Thief, Hacker, BankQualityAssuranceTeam, etc.

For each initial anti-goal and attacker class identified, we need to build an anti-goal refinement/abstraction graph as a basis for exploring countermeasures.

We can do this informally, like for any goal model, by asking *WHY* questions to identify parent anti-goals, and *HOW* questions to identify child anti-goals.

When the goals, domain properties, and anti-goals are specified formally we can use the regression technique presented in Section 7. The difference now is that the anti-goal regression should be applied not only to domain properties but also to requirements and expectations as the attacker should exploit these as well. We will thereby obtain anti-goal preconditions to be satisfied by the attacked software and its environment.

Whatever technique is used, the anti-goal refinement along a branch stops as soon as we obtain a precondition which is monitorable or controllable according to the attacker's capabilities.

Let us illustrate the construction of a threat graph for web banking services using a mix of informal and formal techniques.

We take a Thief agent, for example. Starting from the above anti-goal *Achieve* [AccountNumber&PinKnownByUnauthorized], we obtain through *WHY* questions a parent anti-goal *Achieve*[PaymentMediumKnownByThief] and a grand-parent anti-goal *Achieve* [MoneyStolenFromBankAccounts], see Fig. 15. The milestone refinement pattern in Fig. 7 produces two other subgoals of the parent anti-goal *Achieve* [PaymentMediumKnownByThief], namely, *Achieve* [ThiefKnowsWhichBank] and *Achieve* [ThiefKnowsAccountStructure].

Let us focus on the derivation of refinements for the anti-goal *Achieve* [AccountNumber&PinKnownByUnauthorized]. Looking at the formal specification of this anti-goal, obtained earlier as negation of an instantiated confidentiality goal pattern, we ask ourselves "*what are sufficient conditions in the domain for someone unauthorized to know both the number and PIN of an account simultaneously?*". We may also use the symmetry of the association Matching between account numbers and PINs in the object model and its multiplicity [1..1, 1..N]. As a result we find in *Dom*, or elicit, two symmetrical domain properties:

$\forall$ p: Person, acc: Account
$\neg$ [Owner (p, acc) $\vee$ Proxy (p, acc) $\vee$ Manager (p, acc)] $\wedge$ KnowsV$_p$ (acc.Acc#)
  $\wedge$ ($\exists$ x: PIN) (Found (p, x) $\wedge$ Matching (x, acc.Acc#))
  $\Rightarrow$ KnowsV$_p$ (acc.Acc#) $\wedge$ KnowsV$_p$ (acc.PIN)
$\neg$ [Owner (p, acc) $\vee$ Proxy (p, acc) $\vee$ Manager (p, acc)] $\wedge$ KnowsV$_p$ (acc.PIN)
  $\wedge$ ($\exists$ y: Acc#) (Found (p, y) $\wedge$ Matching (acc.PIN, y))
  $\Rightarrow$ KnowsV$_p$ (acc.Acc#) $\wedge$ KnowsV$_p$ (acc.PIN)

Now we may regress the anti-goal *Achieve*[AccountNumber&PinKnownByUnauthorized] through each of these domain properties to obtain two sub-goals as alternative preconditions for achieving this anti-goal. We thereby obtain an OR-refinement of that anti-goal into two alternative, symmetrical anti-subgoals, namely,

AntiGoal *Achieve* [AccountKnown&MatchingPinFound]
  FormalSpec $\Diamond \exists$ p: Person, acc: Account
    $\neg$ [Owner (p, acc) $\vee$ Proxy (p, acc) $\vee$ Manager (p, acc)]
    $\wedge$ KnowsV$_p$ (acc.Acc#)
    $\wedge$ ($\exists$ x: PIN) [Found (p, x) $\wedge$ Matching (x, acc.Acc#)]

AntiGoal *Achieve* [PinKnown&MatchingAccountFound]

FormalSpec $\Diamond \exists$ p: Person, acc: Account

$\neg$ [Owner (p, acc) $\vee$ Proxy (p, acc) $\vee$ Manager (p, acc)]

$\wedge$ KnowsV$_p$ (acc.PIN)

$\wedge$ ($\exists$ y: Acc#) [Found (p, y) $\wedge$ Matching (acc.PIN, y)]

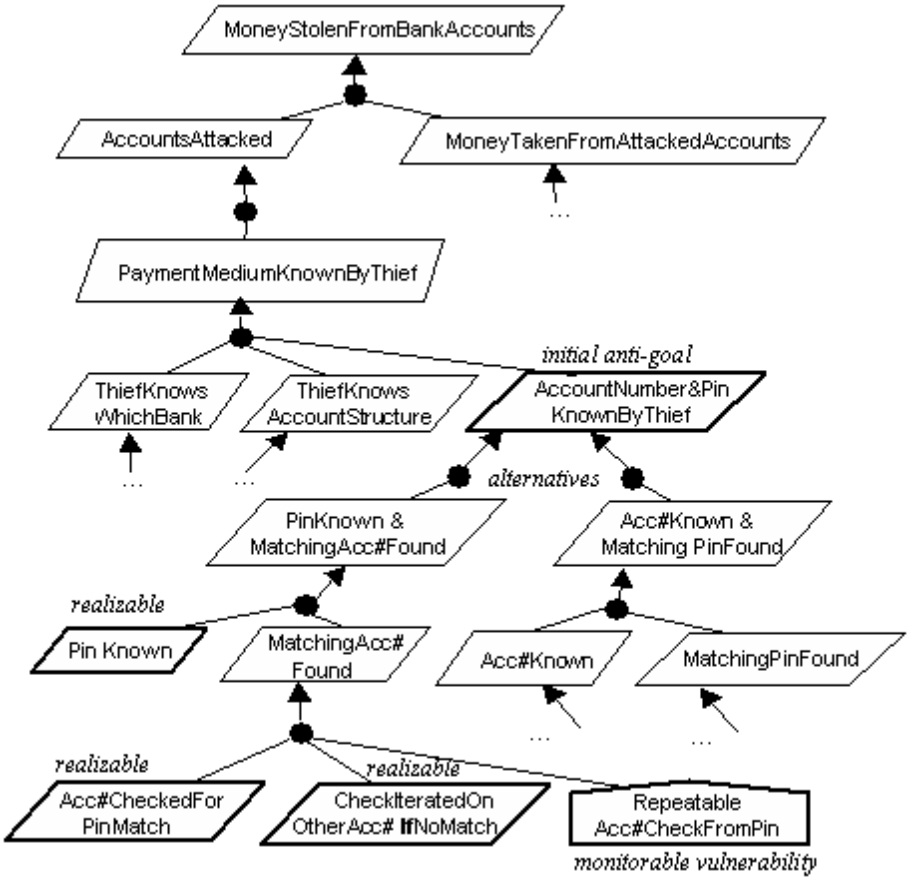

Figure 15. Threat graph fragment for web banking services [Lam04a]

The refinement process goes on until reaching terminal conditions that are either realizable anti-requirements in view of the attacker's capabilities or observable vulnerabilities of the attacker's environment (including the target software). Fig. 15 shows the threat graph obtained.

The derived anti-requirements on the Thief agent are, in the alternative refinement shown in Fig. 15,

AccountNumberCheckedForPinMatch,

CheckIteratedOnOtherAccountNumbers**If**NoMatch

These anti-requirements are realizable under the anti-domain property RepeatableAcc#-

CheckFromPin, stating that the attacker can iterate on account numbers to check whether they match some fixed 4-digit number.

The threat graph in Fig. 15 with this alternative branch corresponds to a real attack reported in [Dos00]. Along the other alternative, not fully elaborated in Fig. 15, we reach symmetrical leaf goals

PinCheckedForAccountNumberMatch,
CheckIteratedOnOtherPins**If**NoMatch,
RepeatablePinCheckFrom Acc#

The two first subgoals are realizable anti-requirements whereas the third condition is a vulnerability precluded by banking systems. This alternative is thus not realizable.

Recent efforts have been devoted to synthesizing threat graphs fully automatically and efficiently [Jan06]. Based on a BDD representation of the initial anti-goal, the technique consists in generating a proof showing that this anti-goal is realizable in view of the attacker's knowledge and capabilities. The proof amounts to a hierarchical plan for satisfying the anti-goal. The hierarchical levels in this plan are determined systematically by incrementally weakening powerful virtual macro-agents until the capabilities of the real attacker agent are reached. The weakening consists in removing macro-agent capabilities by following the anti-goal's BDD state-variable ordering.

8.4. Deriving countermeasures

Based on the threat graphs built for each initial anti-goal, we may obtain new security goals by application of the resolution operators reviewed in Section 7.

In security-critical systems the operator *Avoid[X]* is frequently used, where *X* is instantiated to an anti-goal or a vulnerability. For example, in our web banking system, we would certainly take the following goals as new goals to be refined:

Avoid [RepeatableAcc#CheckFromPin]
Avoid [RepeatablePinCheckFrom Acc#]

Resolution operators can be further specialized to malicious obstacles; in particular, the two following tactics should be considered for countermeasures:

- Make vulnerability condition unmonitorable by attackers.
- Make anti-requirement uncontrollable by attackers.

The alternative countermeasures obtained through such resolution operators must be refined in turn along alternative OR-branches of the updated goal model. Such alternatives must be assessed to keep some "best" one in view of the other non-functional goals [Chu00] and the conflicts they often introduce with other goals in the goal model (see next section). A new threat analysis cycle may need to be undertaken for these new goals.

The threat analysis method reported in this section was applied in the European SAFEE project to model and analyze on-board terrorist threats against civil aircrafts, and explore corresponding countermeasures. A goal model with derived countermeasures was used as a basis for elaborating the requirements for an on-board threat detection/reaction system.

9. Conflict analysis

Requirements engineers are faced with numerous conflicts while elaborating system goals, requirements, and expectations. Conflicts arise from multiple viewpoints among different stakeholders [Fin94], or from different categories of functional and non-functional goals that are potentially conflicting - for example, safety goals tend to be conflicting with performance goals. Security goal categories are especially involved in potential conflicts. For example, "maintain agent anonymity" is potentially conflicting with "achieve agent accountability"; "password-based authentication" is potentially conflicting with "application usability"; "encrypted transaction" is potentially conflicting with "efficient transaction"; and so forth.

Managing interactions among goals, requirements, and expectations is a core business in the RE process [Rob03]. Such interactions much more often amount to *potential* conflicts, rather than logical inconsistencies where one stakeholder says "I want P" whereas another says "I want $\neg$ *P*". The notion of potential conflict is captured through the following definition.

Goals $G_1, \ldots, G_n$ are *divergent* within a domain *Dom* iff there exists a *boundary condition B* such that the following conditions hold:

1.	{Dom, *B*, $\bigwedge_{1 \leq i \leq n} G_i$} $\models$ **false**	*potential conflict*
2.	For each *i*: {Dom, *B*, $\bigwedge_{j \neq i} G_j$} $\not\models$ **false**	*minimality*
3.	There exists a behavior E of the environment of the set of agents in charge of $G_1, \ldots, G_n$ such that $E \models$ O	*feasibility*

The boundary condition captures a particular combination of circumstances which makes the goals $G_1, \ldots, G_n$ conflicting if conjoined to it (see conditions (1) and (2)). Note that a conflict is a particular case of divergence in which *B* = **true**. Also note that the minimality condition precludes the trivial boundary condition *B* = **false**; it stipulates in particular that the boundary condition must be consistent with the domain theory *Dom*. The boundary condition must also be satisfiable by the environment of the agents involved in the satisfaction of the divergent goals.

Conflict management consists in detecting conflicts among goals, generating alternative resolutions of the detected conflicts, and selecting a best resolution ([Lam98a], [Rob03]). We briefly review these steps successively for the more general notion of divergence.

9.1. Detecting divergences

Similarly to obstacles, we may detect divergences among goals by regression or by use of conflict patterns [Lam98a].

Approach 1: Regression. The technique is based on the observation that the first condition for divergence is equivalent to:

$$\{\text{Dom}, B, \textstyle\bigwedge_{j \neq i} G_j\} \models \neg G_i$$

We may thus formally derive the boundary condition *B* as precondition for one of the negated goals $\neg G_i$, chaining backwards through an augmented theory {Dom, $\bigwedge_{j \neq i} G_j$}. The regession procedure is similar to the one given in Section 7.

Let us illustrate how a divergence can thereby be detected between two typical security goals, taken from a real situation [Lam98a]. Consider the electronic reviewing process for a scientific journal, with the following two security goals:

Goal *Maintain* [ReviewerAnonymity]
FormalSpec $\forall$ r: Reviewer, p: Paper, a: Author, rep: Report
Reviews (r, p, rep) $\wedge$ AuthorOf (a, p) $\Rightarrow$ $\Box \neg$ KnowsV$_a$ (Reviews[r,p,rep])

Goal *Maintain* [ReviewIntegrity]
FormalSpec $\forall$ r: Reviewer, p: Paper, a: Author, rep, rep': Report
AuthorOf (a, p) $\wedge$ Gets (a, rep, p, r) $\Rightarrow$ Reviews (r, p, rep') $\wedge$ rep' = rep

In this specification, the object Reviews[r,p,rep] designates a ternary association capturing a reviewer *r* having produced a referee report *rep* for paper *p*. The predicate *Reviews(r,p,rep)* expresses that an instance of this association exists in the current state. The predicate *Gets(a,rep,p,r)* expresses that author *a* has the report *rep* by reviewer *r* for his paper *p*. The *KnowsV* predicate is the epistemic construct introduced in Section 8.

The above goals are *not* logically inconsistent. However, let us see whether they are potentially conflicting. We take the goal *Maintain[ReviewerAnonymity]* for the initialization step of the regression procedure. Its negation is:

$\Diamond \exists$ r: Reviewer, p: Paper, a: Author, rep: Report (NG)
Reviews(r,p,rep) $\wedge$ AuthorOf(a,p) $\wedge \Diamond$ KnowsV$_a$(Reviews[r,p,rep])

Regressing (NG) through the *ReviewIntegrity* goal, whose consequent can be simplified to *Reviews(r,p,rep)* by term rewriting, yields:

$\Diamond \exists$ r: Reviewer, p: Paper, a: Author, rep: Report (NG1)
AuthorOf (a,p) $\wedge$ Gets (a, rep, p, r) $\wedge \Diamond$ KnowsV$_a$(Reviews[r,p,rep])

Let us assume that the domain theory contains the following sufficient conditions for identifiability of reviewers (the outer universal quantifiers are left implicit for simplicity):

Gets (a, rep, p, r) $\wedge$ Identifiable (r, rep) $\Rightarrow \Diamond$ KnowsV$_a$(Reviews[r,p,rep]) (D1)
Reviews (r, p, rep) $\wedge$ SignedBy (rep, r) $\Rightarrow$ Identifiable (r, rep) (D2)
Reviews (r, p, rep) $\wedge$ French (r)$\wedge \neg \exists$ r'$\neq$ r: (D3)
[Expert (r', p) $\wedge$ French (r')] $\Rightarrow$ Identifiable (r, rep)

In these property specifications, the predicate *Identifiable(r,rep)* means that the identity of reviewer r can be determined from the content of report rep. Properties (D2) and (D3) provide explicit sufficient conditions for this. The predicate *SignedBy(rep,r)* means that report rep contains the signature of reviewer r. The predicate *Expert(r,p)* means that reviewer r is a well-known expert in the domain of paper p. Property (D3) states that a French reviewer notably known as being the only French expert in the area of the paper is identifiable (as she makes typical French errors of English usage).

The third conjunct in (NG1) unifies with the consequent in (D1); the regression yields, after corresponding substitutions of variables:

$\Diamond \exists$ r: Reviewer, p: Paper, a: Author, rep: Report
AuthorOf (a, p) $\wedge$ Gets (a, rep, p, r) $\wedge$ Identifiable (r, rep)

The last subformula in this formula unifies with the consequent in (D3); the regression yields:

$\Diamond \exists$ r: Reviewer, p: Paper, a: Author, rep: Report (B)
AuthorOf (a, p) $\wedge$ Gets (a, rep, p, r) $\wedge$ Reviews (r, p, rep)
$\wedge$ French(r) $\wedge \neg \exists$ r' $\neq$ r: [Expert (r', p) $\wedge$ French (r')]

This condition is satisfiable through a report produced by a French reviewer who is the only well-known French expert in the domain of the paper, and sent unaltered to the author (as variable *rep* is the same in the *Reviews* and *Gets* predicates). We thus formally derived a boundary condition making the divergent goals *Maintain[ReviewerAnonymity]* and *Maintain[ReviewIntegrity]* logically inconsistent.

The space of derivable boundary conditions can be explored by backtracking on each applied property to select another applicable one. After having selected (D3), we could select (D2) to derive another boundary condition:

$\Diamond \exists$ r: Reviewer, p: Paper, a: Author, rep: Report (B')
AuthorOf(a, p) $\wedge$ Gets(a, rep, p, r) $\wedge$ Reviews(r, p, rep) $\wedge$ SignedBy(rep, r)

which captures the situation of an author receiving the same report as the one produced by the reviewer with signature information found in it.

Approach 2: Formal conflict patterns. Alternatively we may sometimes shortcut such derivations by instantiating common patterns of divergence among goals that highlight generic boundary conditions. Fig. 16 shows one such pattern that occurs frequently in practice.

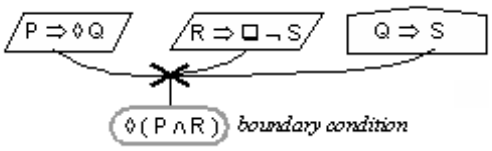

Figure 16. *Achieve-Avoid* divergence pattern

9.2. Resolving divergences

The principle here again is to generate alternative resolutions through resolution operators and then to compare them in order to select a best resolution. Here is a sample of conflict resolution operators.

- *Avoid boundary condition*: A new goal is introduced which takes the form:
$$\Box \neg B.$$
- *Restore divergent goals*: A new goal is introduced which takes the form:
$$B \Rightarrow \Diamond \bigwedge_{1 \leq i \leq n} G_i$$

- *Anticipate conflict*: This strategy can be applied when some persistent condition *P* can be found such that, in some context *C*, we inevitably get into a conflict after some time if the condition *P* has persisted over a too long period:
$$C \wedge \Box_{\leq d} P \Leftrightarrow \Diamond_{\leq d} \neg \bigwedge_{1 \leq i \leq n} G_i$$
In such a case we may introduce the following new goal to avoid the conflict by anticipation:
$$C \wedge P \Rightarrow \Diamond_{\leq d} \neg P$$
- *Weaken* the formulation of one of the divergent goals.
- *Specialize* the target objects concerned by the divergent goals so that the latter now refer to non-overlapping specializations.
- Etc. [Rob03].

In our journal reviewing example, we might resolve the detected divergence by avoiding the boundary condition (that is, not asking a French reviewer in case she is the only French expert in the domain of the paper), or by weakening a divergent goal (e.g., weakening the integrity requirement to allow for correction of typical French errors of English usage).

Conflicts should be carefully considered in safety-critical systems too. An interesting example comes from a document describing some of the requirements for the San Francisco Bay Area Rapid Transit System (BART). One goal stated that the speed commanded to trains may not be "too high", because otherwise it forces the distance between trains to be too high (for safety reasons). Another goal stated that the commanded speed may not be "too low", because otherwise it may force accelerations felt uncomfortable by passengers. These goals are more precisely specified as follows:

Goal *Maintain* [CmdedSpeedCloseToPhysicalSpeed]
FormalSpec $\forall$ tr: Train
tr.Acc$_{CM} \geq 0 \Rightarrow$ tr.Speed$_{CM} \leq$ tr.Speed + f(distance-to-obstacle)
Goal *Maintain* [CmdedSpeedAbove7mphOfPhysicalSpeed]
FormalSpec $\forall$ tr: Train
tr.Acc$_{CM} \geq 0 \Rightarrow$ tr.Speed$_{CM} >$ tr.Speed + 7

The boundary condition for making these goals logically inconsistent is easily derived:
$\Diamond$ ($\exists$ tr: Train) (tr.Acc$_{CM} \geq 0 \wedge$ f(dist-to-obstacle) ≤ 7)
The selected resolution operator should be goal weakening; we should keep the safety goal as it is and weaken the convenience goal in order to remove the divergence by covering the boundary condition:

Goal *Maintain* [CmdedSpeedAbove7mphOfPhysicalSpeed]
FormalSpec $\forall$ tr: Train
tr.Acc$_{CM} \geq 0 \Rightarrow$ tr.Speed$_{CM} >$ tr.Speed + 7 $\vee$ f(dist-to-obstacle) ≤ 7

10. Synthesizing behavior models from scenarios and goals

Goals, scenarios, and state machines form a win-win partnership for system modeling and analysis.

- Goal models support various forms of early, declarative, and incremental reasoning, as seen in the previous section. On the downside, goals are sometimes felt

too abstract by stakeholders. They cover classes of intended behaviors but such behaviors are left implicit. Goals may also be hard to elicit and make fully precise in the first place.

- Scenarios support a concrete, narrative expression style, as discussed in Section 2.7. They are easily accessible to stakeholders. On the downside, scenarios cover few behaviors of specific instances. They leave intended system properties implicit.
- State machines provide visual abstractions of explicit behaviors for any agent instance in some corresponding class (see Section 2.7). They can be composed sequentially and in parallel, and are executable for requirements validation through animation. They can be verified against declarative properties. State machines also provide a good basis for code generation. On the downside, state machines are too operational in the early stages of requirements elaboration. Their construction may be quite hard.

Those complementary strengths and limitations call for an approach integrating goal, scenario, and state machine models where portions of one model are synthesized from portions of the other models.

Recent efforts were made along this line. For example, a labelled transition system (LTS) model can be synthesized from message sequence charts (MSC) taken as positive examples of system behavior [Uch03]. MSC specifications can be translated into statecharts [Kru98]. UML state diagrams can be generated from sequence diagrams capturing positive scenarios ([Whi00], [Mak01]). Goal specifications in linear temporal logic can also be inferred inductively from MSC scenarios taken as positive or negative examples [Lam98b]. These various techniques all require additional input information beside scenarios, namely, a high-level message sequence chart showing how MSC scenarios are to be flowcharted [Uch03]; pre- and post-conditions of interactions, expressed on global state variables ([Lam98b], [Whi00]); local MSC conditions [Kru98]; or state machine traces local to some specific agent [Mak01]. Such additional input information may be hard to get from stakeholders, and may need to be refactored in non-trivial ways in case new positive or negative scenario examples are provided later in the requirements/design engineering process [Let05].

State machine models can be synthesized inductively from positive and negative scenarios without requiring such additional input information [Dam05]. Let us have a closer look at how this synthesis technique works. We start with some background first.

As introduced in Section 2.7, a positive scenario illustrates some desired system behavior. A negative scenario captures a behavior that may not occur. It is captured by a pair *(p, e)* where *p* is a positive MSC, called precondition, and *e* is a prohibited subsequent event. The meaning is that once the admissible MSC precondition has occurred, the prohibited event may not label the next interaction among the corresponding agents.

Fig. 17 shows a collection of input scenarios for state machine synthesis. The upper right scenario is a negative one. The intuitive, end-user semantics of two consecutive events along a MSC timeline is that the first is *directly* followed by the second. The actual semantics of MSCs is defined in terms of LTS and parallel composition [Uch03]. A MSC timeline defines a unique finite LTS execution that captures a corresponding agent behavior. Similarly, the semantics of an entire MSC is defined in terms of the LTS modeling the entire system. MSCs define executions of the parallel composition of each agent LTS.

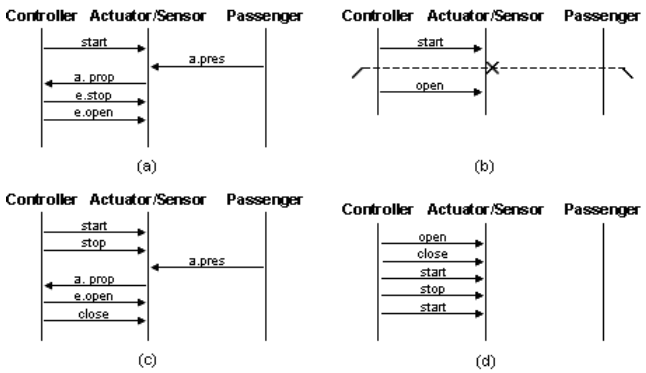

Figure 17. Input scenarios for a train system

For goal injection in the synthesis process, we take a fluent-based variant of LTL where the atomic assertions are explicitly defined in terms of the events making them true and false, respectively [Gia03]. A fluent Fl is a proposition defined by a set $Init_{Fl}$ of initiating events, a set $Term_{Fl}$ of terminating events, and an initial value $Initially_{Fl}$ that can be true or false. The sets of initiating and terminating events must be disjoint. A fluent definition takes the form:

fluent Fl = <$Init_{Fl}$, $Term_{Fl}$> initially $Initially_{Fl}$

In our train example, the fluents *DoorsClosed* and *Moving* are defined as follows:

fluent DoorsClosed = <{close doors}, {open doors, emergency open}> initially **true**

fluent Moving = <{start}, {stop, emergency stop}> initially **false**

A fluent *Fl* holds at some time if either of the following conditions holds:

(a) *Fl* holds initially and no terminating event has yet occurred;

(b) some initiating event has occurred and no terminating event has occurred since then.

LTS synthesis proceeds in two steps [Dam05]. First, the input scenarios are generalized into a LTS for the entire system, called *system LTS*. This LTS is then projected on each agent using standard automaton transformation algorithms [Hop79].

The system LTS covers all positive scenarios and excludes all negative ones. It is obtained by an interactive extension of a grammar induction algorithm known as RPNI [Onc92]. Grammar induction aims at learning a language from a set of positive and negative strings defined on a specific alphabet. The alphabet here is the set of event labels; the strings are provided by positive and negative scenarios.

RPNI first computes an initial LTS solution, called *Prefix Tree Acceptor* (PTA). The PTA is a deterministic LTS built from the input scenarios; each scenario is a branch in the tree that ends with a "white" state, for a positive scenario, or a "black" state, for a negative one. As in the other aforementioned synthesis approaches, scenarios are assumed to start in the same system state.

Fig. 18 shows the PTA computed from the scenarios in Fig. 17. A black state is an error state for the system. A path leading to a black state is said to be *rejected* by the LTS; a path leading to a white state is said to be *accepted* by the LTS. By construction, the PTA accepts all positive input scenarios while rejecting all negative ones.

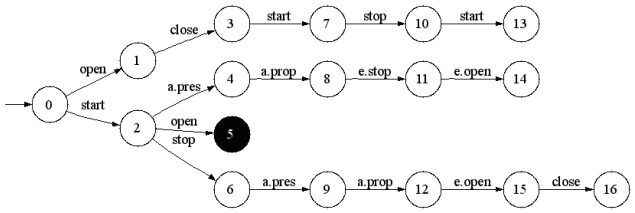

Figure 18. PTA built from the scenarios in Fig. 17

Behavior generalization from the PTA is achieved by a generate-and-test algorithm that performs an exhaustive search for equivalent state pairs to *merge* them into equivalence classes. Two states are considered *equivalent* if they have *no incompatible continuation*, that is, there is no subsequent event sequence accepted by one and rejected by the other.

At each generate-and-test cycle, RPNI considers merging a state q in the current solution with a state q' of lower rank. Merging a state pair (q,q') may require further merging of subsequent state pairs to obtain a deterministic solution; shared continuations of q and q' are folded up by such further merges. When this would end up in merging black and white states, the merging of (q,q') is discarded, and RPNI continues with the next candidate pair. (See [Dam05] for details.)

Fig.19 shows the system LTS computed by the synthesizer for our train example, with the following partition into equivalence classes:

π = { {0,3,6,10,16}, {1,14,15}, {2,7,13}, {4}, {5}, {8}, {9}, {11,12} }

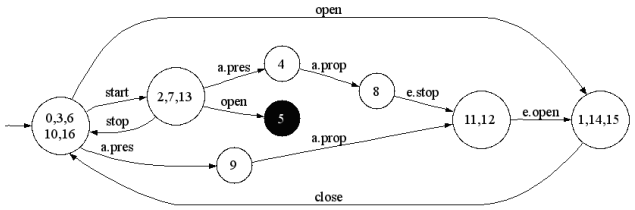

Figure 19. Synthesized system LTS for the train example

The equivalence relation used by this inductive algorithm shows the important role played by negative scenarios to avoid merging non-equivalent system states and derive correct generalizations. RPNI is guaranteed to find the correct system LTS when the input sample is rich enough [Onc92]; two distinct system states must be distinguished in the PTA by at least one continuation accepted from one and rejected from the other. When the input sample has no enough negative scenarios, RPNI tends to compute a poor generalization by merging non-equivalent system states.

To overcome this problem, the LTS synthesizer extends RPNI in two directions:

- Blue Fringe search: The search is made heuristic through an evaluation function that favors states sharing common continuations as first candidates for merging [Lan98].
- Interactive search: The synthesis process is made interactive through scenario questions asked by the synthesizer whenever a merged state gets new outgoing transitions [Dam05].

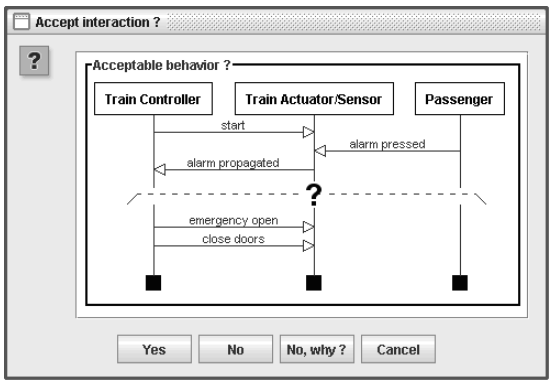

Figure 20. Scenario question generated during synthesis

To answer a scenario question, the user has just to accept or reject the new MSC scenario generated by the synthesizer. The answer results in confirming or discarding the current candidate state merge. Scenario questions provide a natural way of eliciting further positive and negative scenarios to enrich the scenario sample. Fig.20 shows a scenario question that can be rephrased as follows: "*if the train starts and a passenger presses the alarm button, may the controller then open the doors in emergency and close the doors afterwards?*". This scenario should be rejected as the train may not move with open doors.

There is a price to pay with this technique though. While interaction takes place in terms of simple end-user scenarios, and scenarios only, the number of scenario questions may sometimes become large for interaction-intensive applications with complex composite states - as experienced when applying the technique to non-trivial web applications.

This LTS synthesis technique was therefore recently extended to reduce the number of scenario questions significantly and produce a LTS model consistent with knowledge about the domain and about the goals of the target system [Dam06]. The general idea is to constrain the induction process in order to prune the inductive search space and, accordingly, the set of scenario questions. The constraints include:

a) state assertions generated along agent timelines;

b) LTS models of external components the system interacts with;

c) safety properties that capture system goals or domain properties.

Let us have a closer look at optimizations (a) and (c).

Propagating fluents. Fluent definitions provide simple and natural domain descriptions to constrain induction. For example, the definition

fluent DoorsClosed = <{close doors}, {open doors, emergency open}> initially **true**

describes train door states as being either closed or open, and describes which event is responsible for which state change. To constrain the induction process, we compute the value of every fluent at each PTA state by symbolic execution. The PTA states are then decorated with the conjunction of such values. The pruning rule for constraining induction is to avoid *merging inconsistent states*, that is, states whose decoration has at least one fluent with different values. The specific equivalence relation here is thus the set of state pairs where both states have the same value for every fluent. The decoration of the merged state is simply inherited from the states being merged.

To compute PTA node decorations by symbolic execution, we use a simplified version of an algorithm described in [Dam05] to propagate fluent definitions forwards along paths of the PTA tree.

Fig.21 shows the result of propagating the values of fluent DoorsClosed, according to its above definition, along the PTA shown in Fig.18.

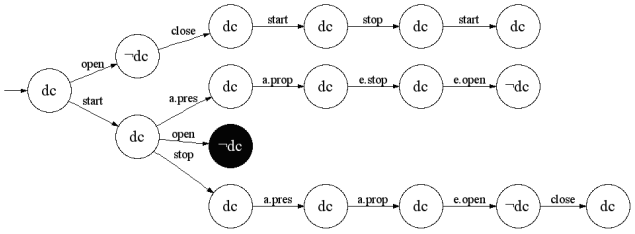

Figure 21. Propagating fluent values along a PTA (*dc* is a shorthand for *DoorsClosed*)

Injecting goals and domain properties in the synthesis process. For goals or domain properties that can be formalized as safety properties, we may generate a property *tester* [Gia03], that is, a LTS extended with an error state such that every path leading to the error state violates the property. Consider, for example, the goal:

DoorsClosedWhileMoving = □(Moving → DoorsClosed)

Fig.22 shows the tester LTS for this property (the error state is the black one). Any event sequence leading to the error state from the initial state corresponds to an undesired system behavior. In particular, the event sequence <*start, open*> corresponds to the initial negative scenario in Fig.17. As seen in Fig.22, the tester provides many more negative scenarios. Property testers can in fact provide potentially infinite classes of negative scenarios.

To constrain the induction process further, the PTA and the tester are traversed jointly in order to decorate each PTA state with the corresponding tester state. Fig. 23 shows the PTA decorated using the tester in Fig.22. The pruning rule for constraining the induction process is now to avoid merging states decorated with distinct states of the property tester. Two states will be considered for merging if they have the same property tester state.

This pruning technique has the additional benefit of ensuring that the synthesized system LTS satisfies the considered goal or domain property. A tester for a safety prop-

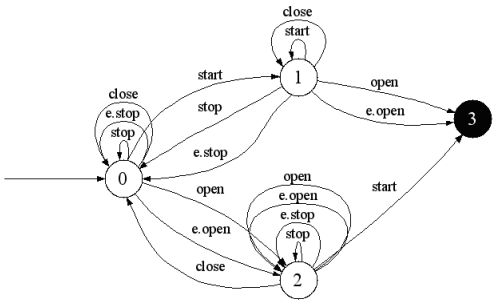

Figure 22. Tester LTS for the goal *DoorsClosedWhileMoving*

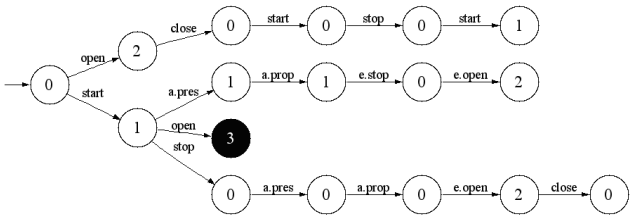

Figure 23. PTA decorated using the tester LTS from Fig. 22

erty is a canonical automaton, that is, minimal and deterministic [Gia03]. A bijection thus exists between states and continuations [Hop79]. In other words, two states are distinct if and only if there is at least one continuation to distinguish them. In the particular case of the tester LTS, two states are distinct if and only if they do not have the same set of continuations leading to the error state.

The question remains as to where these goals and domain properties are coming from. There are complementary answers to this:

- we can pick them up in the goal model, when they are available;
- we can get them systematically by asking the end-user the reason why a scenario is rejected as counterexample;
- we can infer some of them automatically by inductive inference from scenarios ([Lam98b], [Dam06]). In this case, the inferred property has to be validated by the user. If it turns to be inadequate, the user is asked to provide a counterexample scenario which will enrich the scenario collection.

11. Conclusion

It is important to verify that software applications implement their specifications correctly. However, do these specifications meet the software requirements (including non-functional ones)? Do these requirements meet the system's goals, and under realistic assumptions? Are these goals, requirements, and assumptions complete, adequate, and

consistent? These are critical, though still largely unexplored 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 such as intentional, structural, responsibility, operational, and behavioral perspectives. They must cover the entire system, comprising both the software and its environment - made of humans, devices, other software, mother Nature, attackers, attackees, etc. They should also cover the current system-as-is, the system-to-be, and future evolutions. In our framework, such coverage is achieved through alternative subtrees in the goal AND/OR graph. Rich RE models should make alternative options explicit - such as alternative goal refinements, alternative agent assignments, alternative conflict resolutions, or alternative countermeasures to threats. 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, supporting the analysis of partial models, and flexible to accommodate both top-down and bottom-up elaborations.

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

Goal completeness is a key issue. It can be achieved through multiple means such as refinement checking to find out missing subgoals, obstacle and threat analysis to find countermeasure goals, or requirements animation [Tra04].

Declarative specifications play an important role in the RE process - in particular, for communicating with stakeholders and decision makers, for early reasoning about models, and for optimizing model synthesis.

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 hazards, threats, and conflicts.

By discussing a variety of early analysis of RE models we hope we have been convincing on 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 lots of opportunities for formal methods.

Acknowledgement. Many of the ideas presented in this paper were developed over the years jointly with Robert Darimont, Emmanuel Letier, Christophe Damas, Anne Dardenne, Renaud De Landtsheer, David Janssens, Bernard Lambeau, Philippe Massonet, Christophe Ponsard, André Rifaut, Hung Tran Van, and Steve Fickas and his group at the University of Oregon. Warmest thanks to them all!

References

[Bel76] T.E. Bell and T.A. Thayer, "Software Requirements: Are They Really a Problem?", Proc. ICSE-2: 2nd Intrnational Conference on Software Enginering, San Francisco, 1976, 61-68.

[Boe81] B.W. Boehm, Software Engineering Economics. Prentice-Hall, 1981.

[Bor93] A. Borgida, J. Mylopoulos and R. Reiter, "And Nothing Else Changes: The Frame Problem in Procedure Specifications", Proc. ICSE'93 - 15th International Conference on Software Engineering, Baltimore, May 1993

[Bro87] F.P. Brooks "No Silver Bullet: Essence and Accidents of Software Engineering". IEEE Computer, Vol. 20 No. 4, April 1987, pp. 10-19.

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

[Cla00] E.M. Clarke, S. Jha, and W. Marrero, "Verifying Security Protocols with Brutus", *ACM Trans. Software Engineering and Methodology* Vol. 9 No. 4, October 2000, 443-487.

[Dam05] C. Damas, B. Lambeau, P. Dupont and A. van Lamsweerde, "Generating Annotated Behavior Models from End-User Scenarios", *IEEE Transactions on Software Engineering*, Special Issue on Interaction and State-based Modeling, Vol. 31, No. 12, December 2005, 1056-1073.

[Dam06] C. Damas, B. Lambeau, and A. van Lamsweerde, "Scenarios, Goals, and State Machines: a Win-Win Partnership for Model Synthesis", *14th ACM International Symp. on the Foundations of Software Engineering*, Portland (OR), Nov. 2006.

[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.

[Del03] R. De Landtsheer, E. Letier and A. van Lamsweerde, "Deriving Tabular Event-Based Specifications from Goal-Oriented Requirements Models", Requirements Engineering Journal Vol.9 No. 2, 104-120.

[Del05] R. De Landtsheer and A. van Lamsweerde, "Reasoning About Confidentiality at Requirements Engineering Time", *Proc. ESEC/FSE'05*, Lisbon, Portugal, Sept. 2005.

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

[Dos00] A. dos Santos, G. Vigna, and R. Kemmerer, "Security Testing of the Online Banking Service of a Large International Bank", *Proc. 1st Workshop on Security and Privacy in E-Commerce*, Nov. 2000.

[Dwy99] M.B. Dwyer, G.S. Avrunin and J.C. Corbett, "Patterns in Property Specifications for Finite-State Verification", Proc. ICSE-99: 21th Intl. Conference on Software Enginering, Los Angeles, 411-420.

[ESI96] European Software Institute, "European User Survey Analysis", Report USV_EUR 2.1, ESPITI Project, January 1996.

[Fag95] R. Fagin, J. Y. Halpern, Y. Moses, and M. Y. Vardi. *Reasoning About Knowledge*. MIT Press, 1995.

[Fea87] M. Feather, "Language Support for the Specification and Development of Composite Systems", ACM Trans. on Programming Languages and Systems 9(2), Apr. 87, 198-234.

[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.

[Fin94] A. Finkelstein, D. Gabbay, A. Hunter, J. Kramer, and B. Nuseibeh, "Inconsistency Handling in Multi-perspective Specifications", *IEEE Trans. on Software Engineering* Vol. 20 No. 8, 1994, 569-578.

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

[Ham01] J. Hammond, R. Rawlings, A. Hall, "Will it Work?", Proc. RE'01 - 5th Intl. IEEE Symp. on Requirements Engineering, Toronto, IEEE, 2001, 102-109.

[Hop79] J. E. Hopcroft and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, 1979.

[Jac95] D. Jackson, "Structuring Z specifications with views", ACM Transactions on Software Engineering and Methodology, Vol. 4 No. 4, October 1995, 365-389.

[Jan06] D. Janssens and A. van Lamsweerde, Synthesizing Threat Models for Security Requirements Engineering, Département d'Ingénierie Informatique, Université Catholique de Louvain, August 2006.

[Jar98] M.Jarke and R. Kurki-Suonio (eds.), Special Issue on Scenario Management, IEEE Trans. on Sofware. Engineering, December 1998.

[Kem94] R. Kemmerer, C. Meadows, and J. Millen, "Three systems for cryptographic protocol analysis", *Journal of Cryptology* 7(2), 1994, 79-130.

[Kem03] R. Kemmerer, "Cybersecurity", Proc.ICSE'03 - *25th Intl. Conf. on Softw. engineering*, Portland, 2003, 705 - 715.

[Kru98] I. Kruger, R. Grosu, P. Scholz and M. Broy, From MSCs to Statecharts, *Proc. IFIP WG10.3/WG10.5 Intl. Workshop on Distributed and Parallel Embedded Systems* (SchloSS Eringerfeld, Germany), F. J. Rammig (ed.), Kluwer, 1998, 61-71.

[Lad95] P. Ladkin, in The Risks Digest, P. Neumann (ed.), ACM Software Engineering Notes 15, 1995.

[Lam98a] A. van Lamsweerde, R. Darimont and E. Letier, Managing Conflicts in Goal-Driven Rquirements Engineering, *IEEE Transactions on Software Engineering*, Special Issue on Managing Inconsistency in Software Development, Vol. 24 No. 11, November 1998, pp. 908 - 926.

[Lam98b] A. van Lamsweerde and L. Willemet, "Inferring Declarative Requirements Specifications from Operational Scenarios", *IEEE Trans. on Sofware. Engineering*, Special Issue on Scenario Management, December 1998, 1089-1114.

[Lam00a] A. van Lamsweerde, "Requirements Engineering in the Year 00: A Research Perspective". Keynote Paper, *Proceedings ICSE'2000 - International Conference on Software Engineering*, Limerick. IEEE Computer Society Press, June 2000, pp.5-19.

[Lam00b] A. van Lamsweerde and E. Letier, Handling Obstacles in Goal-Oriented Requirements Engineering, *IEEE Transactions on Software Engineering*, Special Issue on Exception Handling, Vol. 26, No. 10, October 2000.

[Lam04a] A. van Lamsweerde, "Elaborating Security Requirements by Construction of Intentional Anti-Models", *Proceedings of ICSE'04 - 26th International Conference on Software Engineering*, Edinburgh, May. 2004, ACM-IEEE , 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.

[Lam07] A. van Lamsweerde, Requirements Engineering - *From System Goals to UML Models to Software Specifications*. Wiley, 2007.

[Lan98] K.J. Lang, B.A. Pearlmutter, and R.A. Price, "Results of the abbadingo one DFA learning competition and a new evidence-driven state merging algorithm", In *Grammatical Inference*, Lecture Notes in Artificial Intelligence Nr. 1433, Springer-Verlag, 1998, 1-12.

[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.

[Let05] E. Letier, J. Kramer, J. Magee, and S. Uchitel, "Monitoring and Control in Scenario-Based Requirements Analysis", *Proc. ICSE 2005 - 27th Intl. Conf. Software Engineering*, St. Louis, May 2005.

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

[Low96] G. Lowe, "Breaking and fixing the Needham-Schroeder public-key protocol using FDR", in *TACAS'96: Tools and Algorithms for Construction and Analysis of Systems*, 1996.

[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.

[Mag00] J. Magee, N. Pryce, D. Giannakopoulou and J. Kramer, "Graphical Animation of Behavior Models", *Proc. ICSE'2000: 22nd Intl. Conf. on Software Engineering*, Limerick, May 2000, 499-508.

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

[Mak01] E. Mäkinen and T. Systä, "MAS - An Interactive Synthesizer to Support Behavioral Modelling in UML", *Proc. ICSE'01 - Intl. Conf. Soft. Engineering*, Toronto, Canada, May 2001.

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

[Man96] Z. Manna and the STeP Group, "STeP: Deductive-Algorithmic Verification of Reactive and Real-Time Systems", Proc. CAV'96 - 8th Intl. Conf. on Computer-Aided Verification, LNCS 1102, Springer-

Verlag, July 1996, 415-418.

[Myl92] Mylopoulos, J., Chung, L., Nixon, B., "Representing and Using Nonfunctional Requirements: A Process-Oriented Approach", IEEE Trans. on Sofware. Engineering, Vol. 18 No. 6, June 1992, pp. 483-497.

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

[Onc92] J. Oncina and P. García, "Inferring Regular Languages in Polynomial Update Time", *In* N. Perez de la Blanca et al (Ed.), *Pattern Recognition and Image Analysis*, Vol. 1 Series in Machine Perception & Artificial Intelligence, World Scientific, 1992, 49-61.

[Par95] D.L. Parnas and J. Madey, "Functional Documents for Computer Systems", Science of Computer Programming, Vol. 25, 1995, 41-61.

[Pon04] Ch. Ponsard, P. Massonet, A. Rifaut, J.F. Molderez, A. van Lamsweerde, H. Tran Van, "Early Verification and Validation of Mission-Critical Systems", Proc. FMICS'04, 9th International Workshop on Formal Methods for Industrial Critical Systems, Linz (Austria) Sept. 2004.

[Rob03] W.N. Robinson, S. Pawlowski and V. Volkov, "Requirements Interaction Management", *ACM Computing Surveys* Vol. 35 No. 2, June 2003, 132-190.

[Sta95] The Standish Group, "Software Chaos", http:// www.standishgroup.com/chaos.html.

[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.

[Uch03] S. Uchitel, J. Kramer, and J. Magee, "Synthesis of Behavioral Models from Scenarios", *IEEE Trans. Softw. Engineering*, 29(2), 2003, 99-115.

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

[Whi00] J. Whittle and J. Schumann, "Generating Statechart Designs from Scenarios", *Proc. ICSE'2000: 22nd Intl. Conference on Software Engineering*, Limerick, 2000, 314-323.

[Yue87] K. Yue, "What Does It Mean to Say that a Specification is Complete?", Proc. IWSSD-4, Fourth International Workshop on Software Specification and Design, Monterey, 1987.