Requirements Engineering: From Craft to Discipline

Axel van Lamsweerde

Department of Computing Science
Université catholique de Louvain
B-1348 Louvain-la-Neuve (Belgium)
avl@info.ucl.ac.be

ABSTRACT

Getting the right software requirements under the right environment assumptions is a critical precondition for developing the right software. This task is intrinsically difficult. We need to produce a complete, adequate, consistent, and well-structured set of measurable requirements and assumptions from incomplete, imprecise, and sparse material originating from multiple, often conflicting sources. The system we need to consider comprises software and environment components including people and devices.

A rich system model may significantly help us in this task. Such model must integrate the intentional, structural, functional, and behavioral facets of the system being conceived. Rigorous techniques are needed for model construction, analysis, exploitation, and evolution. Such techniques should support early and incremental reasoning about partial models for a variety of purposes including satisfaction arguments, property checks, animations, the evaluation of alternative options, the analysis of risks, threats, and conflicts, and traceability management. The tension between technical precision and practical applicability calls for a suitable mix of heuristic, deductive, and inductive forms of reasoning on a suitable mix of declarative and operational specifications. Formal techniques should be deployed only when and where needed, and kept hidden wherever possible.

The paper provides a retrospective account of our research efforts and practical experience along this route. Problem-oriented abstractions, analyzable models, and constructive techniques were permanent concerns.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Requirements/Specification.

General Terms

Documentation, Design, Verification.

Keywords

Requirements engineering, system design, problem modeling, goal orientation, operationalization, responsibility assignment, specification construction, lightweight analysis, formal derivation.

SIGSOFT 2008/FSE-16, November 9–15, Atlanta, Georgia, USA

1. INTRODUCTION

Requirements engineering (RE) is concerned with the elicitation, evaluation, specification, consolidation, and evolution of the objectives, functionalities, qualities, and constraints a software-based system should meet within some organizational or physical setting.

This task has a critical impact on software quality. Requirements-related errors were widely and recurrently recognized to be the most frequent, persistent, expensive, and dangerous types of software errors. The most serious error types include incomplete, inadequate, inconsistent, unmeasurable, or ambiguous requirements or assumptions. Requirements-related errors are the major cause of project cost overruns, delivery delays, failure to meet expectations, or degradations in the environment controlled by the software.

The RE task is difficult.

- We need to cooperate with multiple stakeholders having different background, interests, and expectations. Their concerns are generally partial and often conflicting.
- There are multiple transitions to handle. The problem world we need to investigate is informal whereas the machine solution we want to build is formal [12]. We need to move from partial, unstructured collections of sometimes inconsistent statements to a complete, structured set of consistent requirements. Hidden, implicit needs and assumptions must be made explicit. Imprecise formulations must be converted into precise specifications.
- The problem world may be unfamiliar. While investigating it we need to consider two system versions: the system as it exists before the machine is built into it, and the system as it should be when the machine will operate in it.
- A wide spectrum of concerns must be addressed, ranging from high-level, strategic objectives to detailed, technical requirements and assumptions. Different levels of concern are often intermixed.
- For system robustness and requirements completeness, we need to anticipate the unexpected –including hazardous or malicious environment behaviors. The scope of such investigation may be hard to delimit.
- We generally need to evaluate alternative options for decision making: alternative ways of satisfying system objectives, alternative assignments of responsibilities in the system, alternative resolutions of conflicts, and alternative countermeasures to threats or hazards.

In view of such impact and difficulties, the RE process should be made more disciplined through the use of systematic methods. Such methods should ideally meet the following requirements.

- *Model-driven:* An abstract representation of the system, as-is or to-be, highlights key features and interrelates them. A rich model may provide a comprehensive structure for what needs to be elicited, evaluated, specified, consolidated, and modified. It can be used for explanation, negotiation with stakeholders, and decision making. The requirements document can be semi-automatically generated from it.
- *Constructive:* For complex systems, models are hard to build. A method should provide effectice guidance in building adequate models and in exploiting them.
- *Incremental for early analysis:* A RE method should support stepwise reasoning on acquired fragments of information for early detection and fix of errors in requirements and assumptions.
- *Rigorous but lightweight:* A model-based method and its supporting tools must rely on semantically solid grounds to produce accurate models, beyond boxes and arrows, and enable sound analysis. For wide applicability in practical situations and for communicability of results among ordinary stakeholders, the method should however be easy to use, hiding formal details wherever possible.

This paper outlines our efforts to develop an integrated set of techniques intended to address those requirements [18]. The modeling framework is briefly introduced first (Section 2). Model construction, incremental analysis, and model exploitation are discussed in subsequent sections (Sections 3-5). We briefly report on the practical use of our techniques in industrial projects (Section 6) before concluding with current challenges.

2. MODELING THE PROBLEM WORLD

In order to fully capture the various system facets relevant to the RE process, a model should integrate multiple complementary views (see Fig. 1).

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

2.1 Goals as key RE abstractions

The target system is intended to meet a number of objectives. These are to be highlighted as first-class citizens and interrelated. A *goal* is a prescriptive statement of intent the system should satisfy through cooperation of its agents [4][18]. An *agent* is an active system component having to play some role in goal satisfaction through adequate control of system items.

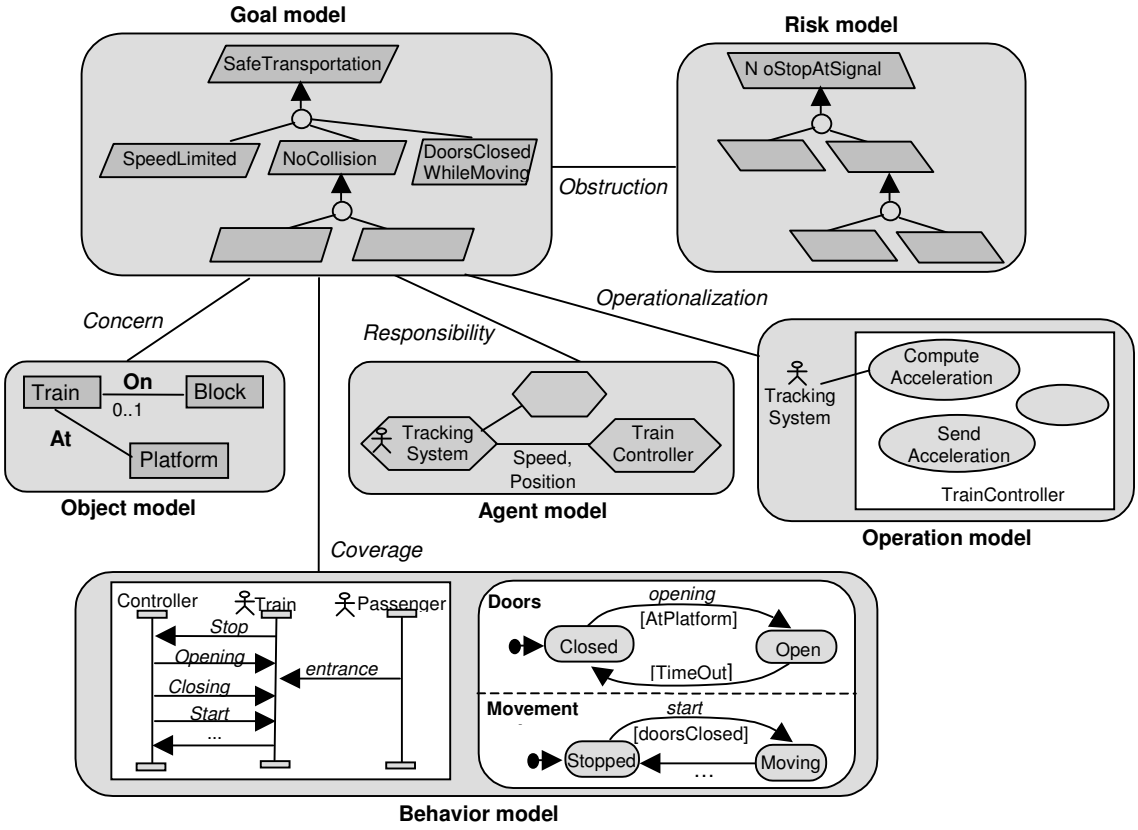

Figure 1. Multi-view modeling for requirements engineering [18]

Goal satisfaction may involve a variety of system agents defining the *system scope* –people, devices, existing software, or software to be developed. The finer-grained a goal is, the fewer agents are required to satisfy it. A *requirement* is a goal under responsibility of a single agent of the software-to-be. An *expectation* is a goal under responsibility of a single agent in the environment of the software-to-be. Expectations form one kind of assumption we need to make for the system to satisfy its goals.

To be under the sole responsibility of an agent, a goal must be *realizable* by this agent [10][25]. This roughly means that the agent must be able to control the state variables constrained by the goal specification and to monitor the state variables to be evaluated in this specification.

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

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

Unlike behavioral goals, a *soft goal* cannot be established in a clear-cut sense. It prescribes preferences among alternative system behaviors, being more satisfied along some alternatives and less satisfied along others. Behavioral goals are therefore used for deriving system operations to satisfy them [4][26] whereas soft goals are used for comparing alternative options to select most preferred ones [1][27].

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

A *goal model* is basically an annotated AND/OR graph showing how higher-level goals are satisfied by lower-level ones (goal *refinement*) and, conversely, how lower-level goals contribute to the satisfaction of higher-level ones (goal *abstraction*) [14]. The top goals are the highest-level ones still in the system scope whereas the bottom goals are assignable requirements or expectations. In such graph, an AND-refinement link relates a goal to a set of subgoals called *refinement*; this means that the parent goal can be satisfied by satisfying all subgoals in the refinement. A goal node can be OR-refined into multiple AND-refinements; each of these is called *alternative* for achieving the parent goal. The meaning of multiple alternative refinements is that the parent goal can be satisfied by satisfying the conjoined subgoals in any of the alternative refinements.

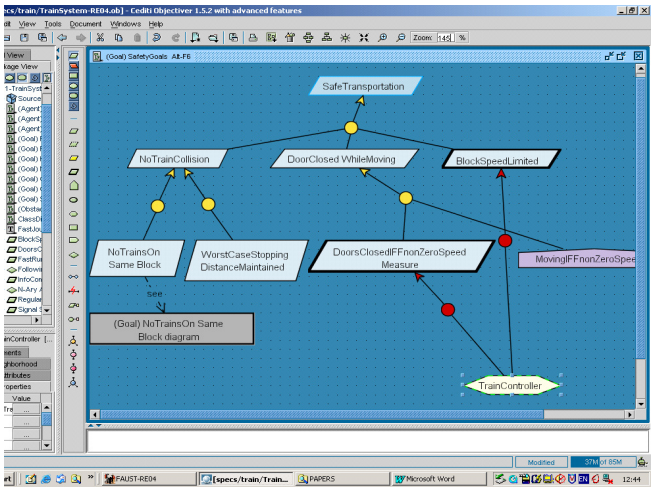

Figure 2. Goal diagram

Fig. 2 shows a goal model fragment as a goal diagram elaborated with our modeling tool [31]. An AND-refinement is denoted by an arrow joining subgoals to the parent goal; multiple incoming arrows indicate an OR-refinement. The figure also shows agent assignments to leaf goals. Home-shaped nodes represent domain properties required for refinement correctness.

Goal nodes in a goal model are annotated with individual features such as their name and precise specification in natural language, their type, category, priority level, elicitation source, etc. Such annotations act as placeholders for dedicated techniques used in the RE process. For example, priority levels are used by conflict management and requirements prioritization techniques during the evaluation phase.

In particular, behavioral goals may optionally be annotated with their formal specification for further analysis. The specification formalism is a first-order real-time linear temporal logic (LTL) [4][19], possibly extended with epistemic constructs for security goals [9].

The systems *as-is* and *to-be* can both be captured within the same model. The two versions share high-level goals and differ along refinement branches of common parent goals. We can thereby also capture multiple variants in a system family [18].

There are numerous reasons why a goal model is so important in the RE process.

- Goal refinement provides a natural mechanism for structuring complex specifications at different levels of concern.
- A goal model provides a rich structure of *satisfaction arguments*, each taking the form:

  $$\{SubGoals, DomProps\} \models ParentGoal$$

  By chaining such arguments bottom-up, we may show that a set of requirements and expectations together ensure some parent goal, the latter ensuring its own parent goal together with others, and recursively, until some high-level goal of interest is thereby shown to be satisfied. In particular, the goal model can be used to show decision makers how the system-to-be will be aligned with the organization's strategic objectives –this proves quite helpful in practice.

- Goals drive the identification of requirements to support them, and provide their rationale. They define a precise criterion for requirements completeness and pertinence [36]. A set of requirements is *complete* with respect to a set of goals if all the goals can be argued to be satisfied when the requirements are satisfied, assuming the environment assumptions and domain properties are satisfied. A requirement is *pertinent* with respect to a set of goals if it is used in the satisfaction argument of one goal at least.
- A goal model supports traceability management [18]. As chains of satisfaction arguments are available from a goal-oriented RE process, there is no need to create and maintain traceability links for evolution support – we get such links for free, from low-level technical requirements on system operations to high-level strategic objectives.
- Goals provide anchors for risk analysis, conflict management, and comparison of alternative options (see Section 4).

2.2 Agents, objects, operations, and behaviors

As introduced before, the intentional view of a system is complemented with other views.

The structural view. Conceptual objects capture domain-specific concepts referred to by prescriptive or descriptive statements about the system. Depending on their nature, they are defined precisely in an associated *object model* as entities, associations, agents, or events. This model is represented by an annotated UML class diagram, where annotations capture individual object features such as a precise definition of the object in natural language, its attributes, relevant domain properties associated with it, initial values when an object instance appears in the system, etc. [18].

An object model provides the concept definitions and domain properties used in the other models. In particular, the object attributes and associations define the system's *state variables* in terms of which goals, agents, operations, and behaviors are specified in the other system views. A tool can easily generate a glossary of terms from the object model which all involved parties need to agree on and use for unambiguous reference. At design time, this model provides a basis for generating a database schema (if any) and architectural fragments.

The responsibility view. Agents were already introduced as active system components that are responsible for the leaf goals in a goal model. The *agent model* captures the distribution of responsibilities within the system together with the capabilities of every agent. The latter are defined in terms of ability to monitor or control object attributes and associations from the object model.

An agent model thus shows the system scope and the boundary between the software-to-be and its environment. Agents can be decomposed into finer-grained ones with finer-grained goal responsibilities [18]. The model may also capture agent wishes on goals, for assignment heuristics and conflict management [4]; agent beliefs, for threat analysis [16]; and dependencies on other agents [1], for vulnerability analysis. An agent model also provides a basis for load analysis. It serves as input to the architectural design process [15].

The functional view. An *operation model* captures the system operations in terms of their individual features and their links to the goal, object, agent, and behavior models. This model specifies, for each operation, its signature, the *descriptive* pre-and postconditions that intrinsically characterize the state transitions produced by the operation in the problem world, and an additional set of *prescriptive* pre, trigger, and post conditions that must further constrain any application of the operation for each underlying goal to be satisfied [4][26]. An operation model is represented by operationalization diagrams.

The model part covering software-to-be operations yields software specifications for input to the development process. We can use them in particular for deriving external specifications of functional components in the software architecture [15]. The model part covering environment operations provides descriptions of tasks and procedures to be jointly performed in the environment for satisfaction of system goals. There are other products we can derive such as black box test data and executable specifications for system animation [35] or software prototyping. The explicit linking of operational specifications to the underlying system goals provides a rich basis for satisfaction arguments, traceability management, and evolution support [18].

The behavioral view. A behavior model captures current behaviors in the system-as-is or desired ones in the system-to-be. Global system behaviors are obtained by parallel composition of agent behaviors. The latter are made explicit through scenarios and state machines. A scenario shows sequences of interactions among specific agent instances. It is represented by a UML sequence diagram. A state machine shows sequences of state transitions for the variables controlled by any agent instance within some class. Such transitions are caused by operation applications or by external events. A state machine is represented by a UML state diagram or by a labelled transition system (LTS) [29] depending on the type of analysis we want to perform on it.

Intance-level scenarios provide partial, narrative representations that are useful for eliciting, validating, or explaining behavioral goals [24]. We can also produce acceptance test data from them. Class-level state machines provide executable representations that can be used for model validation through animation [35], for model checking against formal specifications of domain properties and goals [29][3], and for code generation.

2.3 View integration

The complementary views of the target system are integrated through inter-model links constrained by rules for structural consistency and completeness of the overall model. For example, *responsibility* links connect leaf goals in the goal model and agents in the agent model; *concern* links connect goals in the goal model and the conceptual objects in the object model these goals refer to; *operationalization* links connect leaf goals in the goal model and the operations ensuring them in the operation model; scenarios or state machines in the behavior model are connected to behavioral goals by *coverage* links; and so forth (see Fig. 1).

The rules constraining inter-model links allow us to check the structural completeness and consistency of the overall model, e.g., *"every conceptual item referenced by a goal specification in the goal model must appear as an attribute or object in the object model, and vice versa"; "an agent responsible for a goal must have the capability of controlling the variables constrained by the goal specification and of monitoring the variables to be evaluated in it", "every operation in the operation model must operationalize at least one leaf goal from the goal model"; "if an agent is responsible for a goal, it must perform all operations*

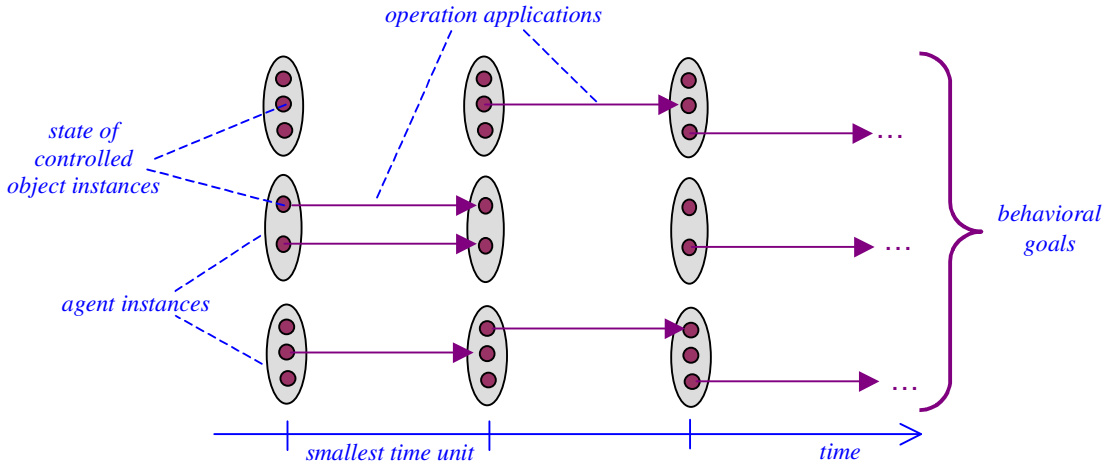

Figure 3. Goals, objects, agents, operations, and behaviors: the semantic picture [18]

operationalizing that goal"; "every state machine capturing the behavior of an agent in the behavior model must show a set of paths prescribed by goals assigned to this agent in the agent model"; etc.

To automate such checks, all view types are defined within a common meta-model [21][4]. The structural rules then constrain metamodel components. Many of them take the form:

> ***For every*** *instance of metaconcept C1 in the metamodel,* ***there must be*** *a corresponding instance of metaconcept C2, linked to it by an instance of the inter-view link type L.*

A modeling tool managing the model database can provide a list of precooked queries we can submit for checking such rules automatically [21][31].

Fig. 3 provides an overall picture of what a system model semantically conveys when the various system views are integrated formally in a LTL-based framework. The leaf behavioral goals together prescribe a maximal set of admissible system behaviors. These behaviors are composed of parallel agent behaviors. A behavior of an agent instance is captured by a sequence of state transitions for the object attributes and associations the agent controls. Such state transitions correspond to applications of operations performed by the agent, taking the smallest time unit. Agent instances evolve synchronously from state to state according to the obligations and permissions prescribed on their operations for goal satisfaction. An agent instance might do nothing along system state transitions while another might be required to apply multiple operations in parallel because of multiple trigger conditions becoming true in the same state. Do-nothing behaviors may arise from lack of permission, as preconditions required for some goals do not hold, or from non-deterministic agent behavior.

Such semantic framework can be integrated with others, e.g., for providing a goal layer on top of SCR [33] and LTSA [29], see [8][28]. Difficulties however arise from different semantic assumptions related to synchronicity and non-determinism.

3. MODEL CONSTRUCTION

An adequate, complete, and consistent multi-view model is difficult to obtain for a complex system. Beyond modeling notations, we need a method to guide us in the model building process.

The KAOS method and supporting toolset were developed, refined, and extended over years of research and practical experience in real projects [17]. (KAOS stands for "Keep All Objectives Satisfied".) Overall it consists of a number of intertwined steps linked by data dependencies (see Fig. 4).

Every elaboration step is supported by a blend of complementary techniques of different types.

- Heuristic rules help identifying, refining, or abstracting model items within a view.
- Formal or semi-formal derivation rules allow items in a view to be obtained from items in another view.
- Formal or semi-formal model building patterns can be reused through instantiation in matching situations.
- More sophisticated procedures, based on deductive or inductive inferencing mechanisms, allow candidate model fragments to be synthesized interactively.
- Analogical reuse techniques allow models of similar systems to be retrieved, transposed, and adapted [30][13].

Bad smells are also provided to let modellers avoid common pitfalls [18].

3.1 Building the goal model

As goals prove difficult to identify and structure in practice, we especially need guidance for elaborating the intentional view. Here is a sample of more or less elaborate techniques that work quite well in practice.

Goal identification.

- Search for prescriptive and intentional keywords in statements found in elicitation material [14].
- For every problem identified in the system-as-is, derive an improvement goal [4].
- Identify wishes of human agents [4].
- Browse goal taxonomies to instantiate leaf goal categories to system-specific objects [4][1][9] –e.g., satisfaction, information, accuracy, confidentiality, availability, or response time goals.
- Ask *WHY* and and WHY NOT questions about positive and negative scenarios, respectively, as they are provided by stakeholders [20] or generated by a model synthesizer [3].

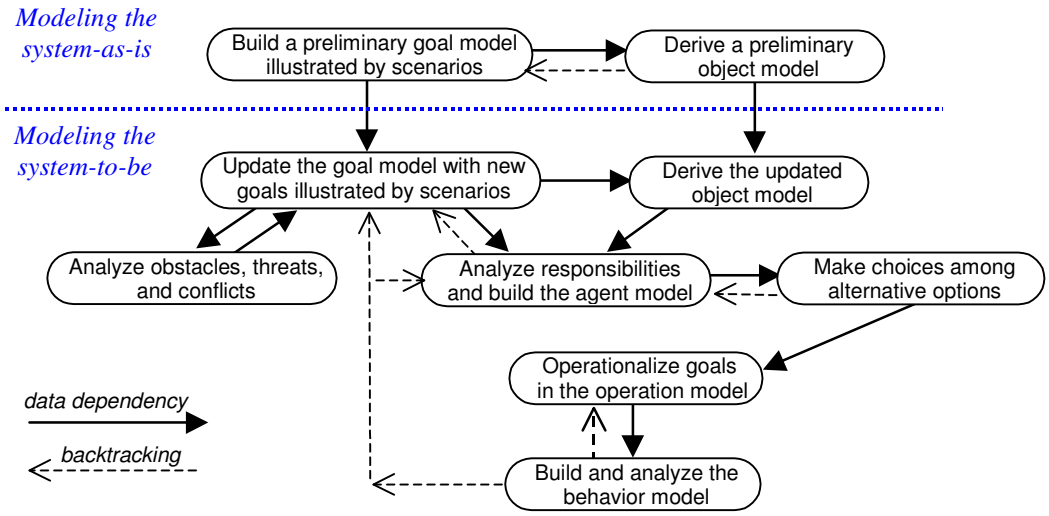

Figure 4. Main steps of a model building method for RE

- Infer goal specifications inductively from scenarios taken as positive or negative examples [24].
- Identify soft goals and contribution links by analyzing the pros and cons of alternative refinements [18].
- Check the converse of *Achieve* goals as candidate *Maintain* goals [18] –for example, the e-commerce goal *Achieve*[ItemSent**If**Paid] yields the goal *Maintain*[ItemSent**OnlyIf**Paid]; the flight management goal *Achieve*[ReverseThrust**If**PlaneOnRunway] yields the goal *Maintain*[ReverseThrust**Onlyf** PlaneOnRunway].

Goal refinement and abstraction

- Ask *HOW* and *WHY* questions about identified goals to obtain subgoals and parent goals, respectively [20].
- Split responsibilities among agents towards goals realizable by single agents [4][25].
- Use goal refinement patterns formally [5][25] or semi-formally [18]. Such patterns encode common refinement tactics on generic goals specified in LTL. Their correctness is formally proved once for all. As multiple patterns might be applicable in the same situation, we can thereby explore alternative refinements. Examples include the milestone-driven, decomposition-by-cases, guard-introduction, unrealizability-driven, and divide-and-conquer patterns (see Fig. 5).

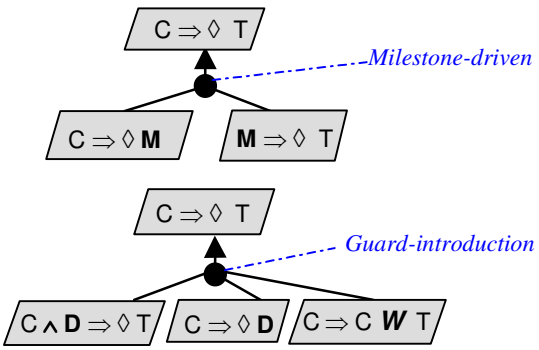

Figure 5. Refinement patterns

3.2 Building the object model

UML gurus often confess that the building of a complete and pertinent class diagram requires much experience and creativity. In our framework, such diagram is derived incrementally from the goal model, semi-formally or formally, using rules such as the following [14][18].

- Take all conceptual objects referred to in the specification of identified goals and domain properties, and only those.
- Derive associations and participating objects from atomic predicates in those specifications (or, informally, from linking expressions in them).
- If the same domain property might be attached to different objects, consider attaching it to an association between these objects.
- Use standard heuristics for deciding whether a concept should be an entity, association, attribute, or event (e.g., autonomous vs. dependent object, passive vs. active object, multi-state vs. single-state object, etc.)
- Identify specializations from classification expressions and discriminant factors in the specification of goals and domain properties. Identify generalizations from objects characterized by similar attributes, associations, or domain properties.
- Identify tracking associations between environment objects and software counterparts, together with corresponding accuracy goals [25].

3.3 Building the agent model

Some heuristic rules may help us here such as the following.

- Identify any active object referred to in the specification of leaf goals [4].
- Check for software counterparts of assignments to human agents that are overloaded in the system-as-is [4].
- Look for agents whose capabilities match the variables evaluated in and constrained by a leaf goal specification, respectively [25].

- Consider abstract agents and responsibilities first and then refine these until individual roles are reached [18].
- Avoid assigning a goal to an agent if this goal is conflicting with the agent's wished goals [4]. Avoid goal assignments resulting in critical dependencies among agents [1]. If not possible, introduce defensive goals against such vulnerabilities in the goal model. Favor trustworthy agents for assignment of security goals.

We can automatically generate a *context diagram* from an agent model that shows the interfaces among all system agents [18]. The detection of dataflow "holes" in the generated diagram calls for the introduction of missing agents to control or monitor the corresponding dataflow.

3.4 Building the operation model

Various techniques and heuristics may help us identify operations and elaborate goal operationalizations. Here are some.

- *Derive operations from goal fluents:* For each atomic state condition P in a goal specification, determine its initiating operation, with domain precondition $\neg P$ and domain postcondition P, and its terminating operation, with domain precondition P and domain postcondition $\neg P$ [18][29].
- *Derive operations from* scenarios: Identify operations from interaction events, and determine their domain pre- and postconditions from state conditions characterizing the agent timeline right before and right after the corresponding interaction [24].
- *Strengthen domain pre- and postconditions with permissions, obligations, and additional effects:* Consider any operation whose effect may affect some goal. If this effect can violate the goal under some condition C, take $\neg C$ as required precondition for this goal. If the goal prescribes that this effect must hold whenever some sufficient condition C became true, take C as required trigger condition. If the effect is not sufficient to ensure the target condition prescribed by the goal, take the missing subcondition C in the target as required postcondition [18].
- Use formal operationalization patterns that encode common ways of converting LTL specifications of behavioral goals into complete and consistent sets of required pre, trigger, and postconditions on operations [26].

UML use case diagrams can easily be generated from an operation model to provide an outline view of the system's functionalities in relation with the goal model [18][31].

3.5 Building the behavior model

This task is not easy. Like examples or test cases, instance-level scenarios raise a coverage problem as they are inherently partial. Class-level state machine models, on the other hand, may be very complex.

Elaborating scenarios. Heuristics available from the RE literature may be used for structuring and consolidating scenarios emerging from elicitation material [18]. For example, we may first express normal courses of interaction and then, at the end of each episode along a normal scenario, systematically consider exceptional conditions and their required abnormal episode. We may also identify auxiliary episodes, responses needed to external stimuli, etc.

The elaborated scenarios should be checked against unrelated concerns, irrelevant events, impossible interactions, and incompatible or inadequate granularities [24]. We may also use animation tools to check their adequacy [29][35].

Interesting scenarios can be produced by a model checker [29] or a goal refinement checker [34]. The scenario questions generated by our state machine synthesizer are another source of positive and negative scenarios [2].

For scenario-based reasoning during model construction, it is often useful to decorate scenario timelines with *state conditions* monitored or controlled by the corresponding agent. Such conditions are generated by propagating the domain pre- and postconditions of the operations associated with each interaction down the timeline [24].

Synthesizing state machine models. Several complementary approaches were explored in our work.

- *Goal-driven synthesis* [35]: In this approach, state diagrams are derived from goals and their operationalization. A state diagram for some controlled variable is obtained by retrieving all goal operationalizations where the variable appears as operation output. The states, transitions, and transition labels in the diagram are derived from the domain pre- and postconditions of those operations together with their required pre- and trigger conditions. The agent's behavior model is the parallel composition of such diagrams for the variables the agent controls. Such derivation additionally provides satisfaction arguments and derivational traceability links for model evolution.
- *Scenario-driven synthesis:* Two different approaches were considered dependent on whether the target behavior model is state-based or event-based.
  - A *state-based* model is obtained from a set of scenarios by generalizing the latter so as to refer to any agent instance and to cover all behaviors captured by the scenarios [18]. A state diagram for some variable controlled by an agent is obtained by deriving state machine paths from the sequences of state conditions on this variable along the agent's timelines in the scenarios. These paths are then merged to form the state diagram for this variable. The agent's state diagram is obtained as parallel composition of such diagrams.
  - An *event-based* model can be synthesized interactively, when formal state conditions along scenario timelines are not available, using grammar induction techniques and scenario questions [2]. The generalization search space and the number of generated scenario questions are substantially reduced by injection of fluents, goals, and domain properties to prune the search space [3]. With such additional knowledge the model adequacy is obviously improved too.

4. INCREMENTAL MODEL ANALYSIS

Whatever techniques are used for building our multi-view models, we need companion techniques to check their adequacy, completeness, and consistency. Such checks should be made early, for early fix, and stepwise while building the model. The goal model supports this as it is declarative and captures different levels of abstraction and precision.

The query-based structural checks in Section 2.3 are surface-level ones; they do not take into account the optional formal specifications annotating goals, objects and operations in the model. This section outlines different types of formal analysis that can be performed on formalized goal fragments when available. A weaker version of most of them can however be used when no LTL formalization is available [18].

4.1 Refinement checking

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

For such checks we can use a LTL theorem prover, formal refinement patterns, or a bounded SAT solver. To shortcut the heavyweight theorem proving approach, formal patterns prove quite effective in matching situations (see Fig. 5). As they are proved once for all to produce consistent and complete refinements, their instantiation can reveal missing subgoals [5].

A roundtrip use of a *SAT solver* is another effective, more general approach. A front-end tool to a bounded SAT solver for LTL can check any refinement and produce bounded trace counterexamples in case of incomplete refinement [34]. For a refinement of goal G into subgoals G_1, ..., G_n in some domain theory *Dom,* the tool (a) asks the user to select a trace bound and specific object instances for propositionalization of the submitted formulas; (b) translates the result in the input format required by the selected SAT solver; (c) runs the SAT solver to check whether the formula:

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

is satisfiable and, if so, generate a satisfying trace; and (d) translates the output back to the level of abstraction of the graphical input model. Fig. 6 shows the result produced by our tool when the two left subgoals only are taken in the refinement. The counterexample trace shows a train getting back to the preceding block instead of waiting until the block signal is set to 'go'. Such negative scenario may suggest the missing subgoal.

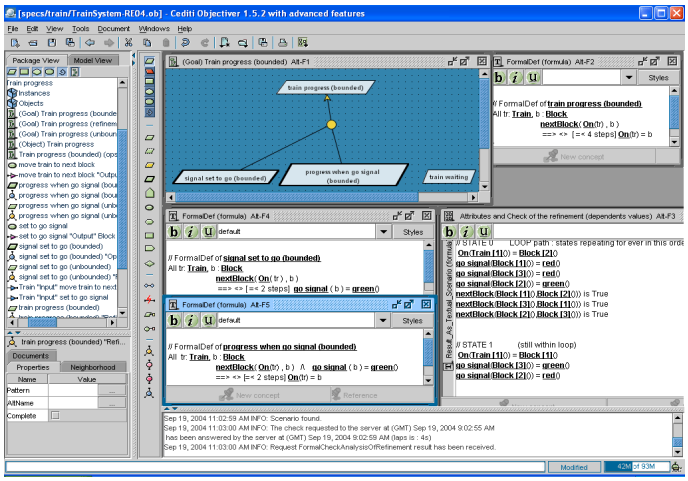

Figure 6. Bounded SAT solving for refinement checking

4.2 Checking operationalizations

A LTL semantics for operationalization allows us to formally derive correct operationalizations by use of *operationalization patterns*. Fig. 7 shows a very simple pattern that also suggests how a LTL goal can be mapped to a consistent and complete set of operations. The patterns are organized for easy retrieval according to a taxonomy of goal specification patterns [26]. They are proved correct once for all. We can use them forwards, to derive operations with their domain conditions and required pre-, trigger, and postconditions; or backwards, for goal mining from operational specifications.

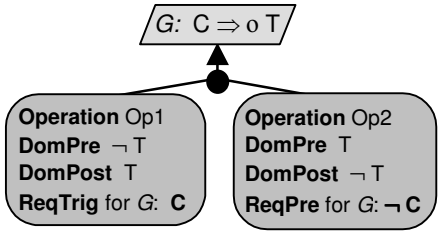

Figure 7. Operationalization pattern for *Immediate Achieve* goals

More generally, we can use our SAT solver tool to check the consistency and completeness of given operationalizations [34]. The formula checked for satisfiability is now:

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

where G is the operationalized goal and $[|\, R_i \,|]$ denotes the LTL formula expressing that the corresponding operation is applied under its required (pre, trigger, or post) condition R_i [26].

4.3 Risk analysis

For system robustness, we need to perform risk analysis early at RE time. Anticipating what could go wrong with an overideal system model is essential for getting an adequate and complete set of software requirements and environment assumptions. *Obstacle analysis* is a goal-based form of risk analysis aimed at identifying, assessing, and resolving the possibilities of breaking assertions in the goal model [22]. An *obstacle* is a precondition for non-satisfaction of some goal, assumption, or questionable domain property used in the goal model. It must be consistent with valid domain properties and hypotheses, and feasible through agent behaviors.

An obstacle diagram is a goal-anchored risk tree showing how a root obstacle to some assertion can be AND/OR refined into subobstacles whose feasibility, likelihood, and resolution are easier to establish. OR-refinements show different ways of obstructing the assertion. They should ideally be domain-complete. AND-refinements capture specific combinations of circumstances for obstruction. They should be complete, domain-consistent, and minimal. Leaf obstacles are connected to countermeasure goals through resolution links (see Fig. 8).

We can build obstacle diagrams systematically using techniques such as tautology-based refinement, refinement driven by necessary conditions in the domain for the obstructed target, and obstacle refinement patterns [18]. Fig. 9 illustrates the latter two techniques into a single pattern for obstructing an *Achieve* goal.

Obstacles estimated to be sufficiently likely and critical need be resolved. Alternative resolutions should be explored before a best one is selected based on estimations of risk reduction leverages and contributions to non-functional goals. Such exploration can be made systematic by use of operators that transform the goal model so as to eliminate the obstacle, reduce its likelihood, or

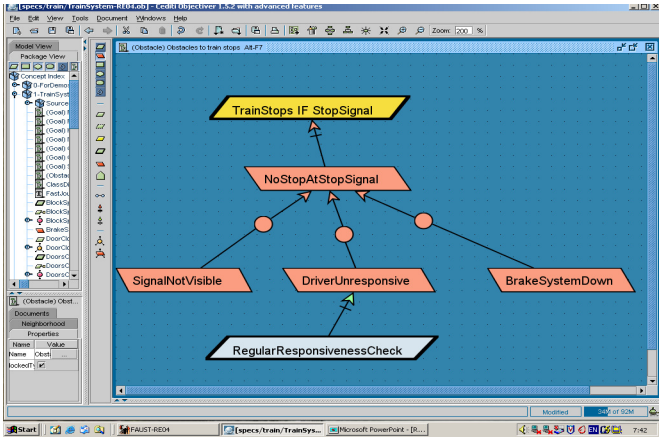

Figure 8. Portion of obstacle model with new goal as resolution

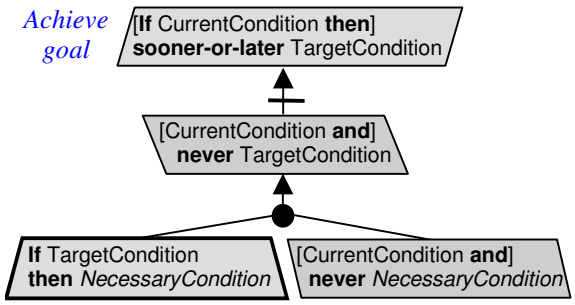

Figure 9. Obstacle refinement driven by necessary conditions for the obstructed target

attenuate its consequences. Such operators include goal substitution, agent substitution, obstacle prevention, goal weakening, obstacle reduction, goal restoration, and obstacle mitigation [22].

When a goal to be obstructed is formalized in LTL, we can generate obstacles formally by regressing the goal negation through formal specifications of domain properties. Regression is a declarative precondition calculus for abductive reasoning. It may be used to elicit domain properties as well [22].

Formal obstruction patterns allow us to shortcut the formal derivations involved in regression. The semi-formal pattern in Fig. 9 has an obvious LTL counterpart. Other patterns are available for *Maintain* goals and for other types of obstruction [22].

The example in Fig. 9 also suggests how formal patterns can be used semi-formally, e.g., by people having no LTL background.

4.4 Threat analysis

Threat analysis is aimed at breaking the model at RE time in order to anticipate security problems and complete the model with adequate security countermeasures. Threats amount to obstacles to security goals. They can be unintentional or intentional. The analysis of intentional threats calls for modeling malicious agents, their anti-goals, and their capabilities. A threat model and its associated countermeasures can be elaborated systematically, like a goal refinement model, starting from negations of security goals and ending up when fine-grained anti-goals are reached that are realizable by attackers in view of their assumed capabilities [16].

For formal threat analysis, we need to specify security properties in terms of epistemic constructs that capture what agents may or may not know [16][9]. Formal anti-goal regressions through properties of the attacker's environment allow us to derive portions of a threat model formally. We can explore security countermeasures on a more solid basis then.

Recently, we developed a threat model synthesizer that generates proof trees showing how the attacker's anti-goal can be satisfied through the attacked software in view of its capabilities. The approach relies on BDD technology and calculations of minimal or maximal fixpoints, depending on whether the anti-goal is an *Achieve* or a *Maintain* goal, respectively. Such fixpoint approach was developed in the good old time in a different context [23].

4.5 Conflict analysis

A goal model should be analyzed against potential conflicts among overlapping goals. *Divergence* is a weak form of conflict, more frequently found in RE than pure logical inconsistency. It captures a situation where some goals become logically inconsistent *if* some *boundary condition* holds.

Divergences are detected by finding boundary conditions that are feasible and consistent with domain properties [19]. Such conditions can be generated by regressing goal negations through overlapping goals. Alternatively, we may use heuristic rules based on the metamodel and on goal categories. Divergence patterns are also available.

Once detected, the conflicts may be recorded in the goal model, for later resolution, through *conflict* links connecting the divergent goal nodes. Like for obstacles, formal operators may help us explore conflict resolutions in more solid and accurate ways.

4.6 Goal-oriented animation

Animation is a widely recognized approach for checking model adequacy. Animators generally simulate behavioral models that do not directly reflect the objectives, constraints and assumptions stated declaratively by stakeholders.

Our animation tool animates goal-oriented models [35]. It compiles goal operationalizations into parallel state machines (see Section 3.5), instantiates these to user-selected object instances, executes the instantiated machines from concurrent events input by one or more users, and visualizes the concurrent simulations as animated scenes in the environment [29] –see Fig. 10. The tool also monitors property violations, and can "slice" the animation on specific model portions relevant to user-selected goals.

4.7 Reasoning about alternative options

Any RE process is faced with alternative options among which to decide. In a goal-oriented framework, alternative options refer to alternative goal refinements, conflict resolutions, obstacle resolutions, threat resolutions, and responsibility assignments. Different alternatives contribute to different degrees of satisfaction of the soft goals in the goal model. To select best options, we can use qualitative or quantitative approaches.

On the *qualitative* side, we can compare options against high-level soft goals in the model by assessing their contribution to each leaf soft goal through qualitative labels (+, ++, -, etc.). Such labels are propagated bottom-up along refinement and conflict links in the goal graph until the top-level soft goals are reached [1].

For *quantitative* reasoning, we may replace such labels by numerical scores. The total score of each option is determined as a

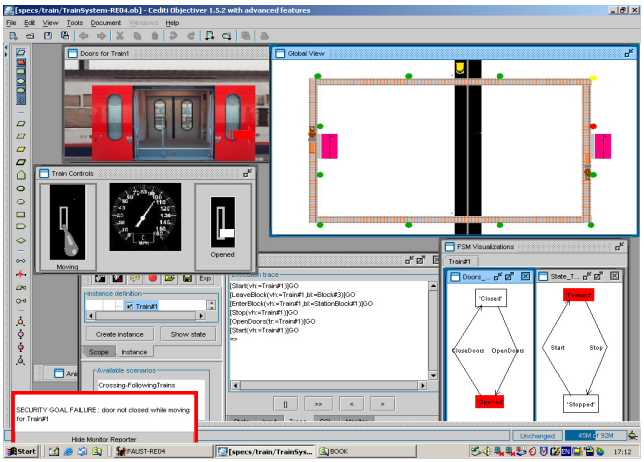

Figure 10. Goal-oriented animation

weighted summation of its scores with respect to each leaf soft goal in the model. For non-subjective conclusions, the option scores should be grounded on measurable system phenomena [27]. When the leaf soft goals are specified in measurable terms, we can identify gauge variables that provide measures for comparing options. A *gauge variable* for some soft goal captures an associated quantity such as something the soft goal prescribes to *Improve, Increase, Reduce, Maximize*, or *Minimize*; or the estimated cost of satisfying the soft goal; or the estimated time taken for satisfying it. Gauge variables propagate additively along the AND-trees refining alternative options in the model. We can thereby compare the overall scores of different options from the leaf values in their respective refinement trees [18].

5. MODEL EXPLOITATION

A good model for RE is a means, not an end. Beside early detection and fix of a variety of problems, we can use our goal-oriented models for generating the requirements document, for managing traceability to anticipate requirements evolution, for derivation of a preliminary software architecture, and for runtime system adaptation.

5.1 Generating the requirements document

A goal-oriented model allows us to produce the requirements document semi-automatically [31]. The textual structure of this document is first generated from the structure of the goal model according to some required documentation standards –e.g., IEEE Std-830. This structure is populated with the annotations of all model elements at some appropriate place. In particular, the glossary of terms is generated from the object model; the section on user requirements is produced by a pre-order traversal of the goal model from top to bottom; the section on assumptions and dependencies is generated from the expectations on environment agents and the unresolved obstacles. The resulting text is expandable by "drag-and-drop" of model diagram portions and any other item the model is hyperlinked with.

5.2 Traceability management

The model provides intra- and inter-view links among its elements that define *Use* and *Derivation* traceability links we can browse backwards and forwards to retrieve the source, rationale, and impact of decisions [18]. We thus save the cost of creating and maintaining traceability links separately; the model gives them for free.

5.3 From requirements to architecture

A goal-oriented model can also be used for deriving a preliminary software architecture [15]. An abstract dataflow architecture is first generated from the agent and operation models. This architecture is transformed to meet some architectural style matching the architectural requirements found in the goal model. The components and connectors in the resulting architecture are refined next by use of architectural refinement patterns that meet the non-functional goals in the model.

5.4 Runtime self-adaptation

We may sometimes want to defer the resolution of some obstacle until system runtime when the obstacle occurs (if it does). On another hand, a selected option in the goal model might contain some requirements or assumptions that might no longer be adequate when the system will be running. In such cases, a monitoring-based component of the software-to-be can let the latter run the resolution or dynamically shift to some other model option when necessary. If such obstacle, requirement, or assumption is formalized in LTL, a dedicated monitor can be automatically generated. At system runtime, this monitor runs concurrently with the software to detect event sequences that violate the monitored assertion (or satisfy it, in case of an obstacle). Violations are reported to a rule-based compensator for system reconfiguration to a more adequate alternative option in the model [11]. The monitoring part of this mechanism was in fact implemented for detection of goal violations at system animation time [35], see the red-lined box on the left bottom of Fig. 10.

6. FROM RESEARCH TO PRACTICE

Our experience convinced us that RE research, tool development, and practice should be highly intertwined. In most projects we were involved in, the customers felt that the method and supporting tools produced much better requirements documents. Conversely, the techniques and tools resulting from research were significantly refined, simplified and polished over the years from feedback from practice.

KAOS was applied in about 25 industrial projects in a wide variety of domains to engineer requirements for fairly different types of systems [17]. The method was also used to build goal-oriented models for strategic planning and business process reengineering projects, to reengineer unintelligible requirements documents, to generate calls for tenders and tender evaluation forms, and to elaborate a model of on-board terrorist threats in civil aviation as a basis for a software-based detection/reaction system [7]. Others have integrated some of our techniques in their commercial product [32].

Examples of system models elaborated with our techniques include a phone system on TV cable, ATC systems for inter-controller communication support and conflict handling between ground and on-board collision avoidance systems, a test suite design system for rocket launch, production management systems in the steel and automotive industries, a complex copyright management system, a newspaper back office system, various healthcare systems, and business management systems in the pharmaceutics, food, and insurance sectors.

Table 1 gives an idea on the size of the models constructed in some of these projects. The generated requirements documents were ranging from 100 to 300 pages. Medium-size projects typically range from 3 to 8 person-months, with 1-2 consultants working over a period of 2-5 months. The effort required for model building and validation was about twice the effort required for stakeholder interviews.

Concept type	**A**	**B**	**C**	**D**	**E**	**F**	**G**
Goal	370	56	141	341	640	171	151
Requirement	160	50	164	256	440	311	108
Agent	80	24	32	8	315	116	21
Entity	240	123	106	102	215	166	127
Association	90	71	48	13	77	126	5
Operation	NA	59	42	NA	86	36	80

Table 1: Number of modelled concepts for systems *A-G* [17]

The successful completion of such projects would never have been possible without fully reworking the GRAIL research prototype [6] into a professional version for model editing, browsing, querying, and requirements document generation [31]. (The formal analysis [34] and synthesis [2] tools are still research protypes at this stage.)

Goal graphs consistently proved to be quite an effective support for validation and negotiation with stakehoders. Decision makers, in particular, don't care about class diagrams or state machine models; they want to see alternative options in the goal model to think about and make decisions.

The informal use of formal refinement, obstruction, and conflict patterns by analysts having no LTL expertise worked surprisingly well in practice. In particular, refinement patterns revealed questionable refinements in a number of cases.

7. CONCLUSION

A multi-view model integrating system goals, objects, agents, operations, and behaviors is a key artifact for articulating requirements elicitation, evaluation, specification, consolidation, and evolution. Building an adequate, complete, and consistent model is far from trivial. Completeness is especially difficult as we need to pessimistically anticipate unexpected system behaviors.

Constructive techniques may help elaborate such models both bottom-up, from scenario examples and other operational material, and top-down, from goals and other declarative material. A synergistic blend of modeling heuristics, patterns, reuse mechanisms, derivation rules, and more sophisticated forms of deductive and inductive reasoning may guide analysts in the model building process. Such techniques are to be complemented with others for early, incremental analysis of partial models. They are rooted in research in software engineering, artificial intelligence, databases, and formal methods.

In many cases, the formal techniques can have a semi-formal counterpart. A "two-button" method and toolset, where the formal button is pressed only when and where needed, proves invaluable in practice for wider accessibility.

There are indications that such techniques have reached a certain degree of maturity. On the one hand, they were applied successfully in a variety of challenging projects. On the other hand, they are currently exported and adapted to other areas, including safety-critical medical processes which we are currently focussing on.

The road is paved with many challenges. In particular, we lack precise yet simple techniques for dealing with goals to be satisfied only partially –in *X%* of the cases, say. The integration in the RE process of lightweight quantitative techniques for decision support is much needed. Beyond traceability management, effective support for requirements evolution is missing. Dedicated RE techniques for certain kinds of systems, such as product lines, and certain kinds of non-functional requirements, such as security, are still in their infancy. The interplay between RE and architectural design is not well understood yet. Multiple formal frameworks are hard to integrate and hide to ordinary users.

Yet the biggest challenge, we believe, remains in technology transfer and the moving from best practices to normal practices. Practitioners often seem reluctant to spend some effort in the RE process. They are, in a sense, like cigarette smokers who know that smoking is pretty unhealthy but keep smoking.

Requirements engineering is traditionally seen as a craft. There are some prospects for turning it into a discipline established on more solid, technical grounds. Further research is needed along this way. It is worth the effort though. After all, the most automated form of software engineering will always require requirements engineering.

8. ACKNOWLEDGMENTS

Special thanks are due to Emmanuel Letier and Robert Darimont for significant contributions to the material outlined in this paper. Many other people contributed to the KAOS project. I wish to thank in particular Christophe Damas, Anne Dardenne, Renaud De Landtsheer, Emmanuelle Delor, Martin Feather, Steve Fickas, Bernard Lambeau, Philippe Massonet, Cédric Nève, Christophe Ponsard, André Rifaut, Jean-Luc Roussel, Hung Tran Van, and Laurent Willemet. Much of the work has been supported by the Walloon Region, the Belgian National Research Council, and the European Union.

9. REFERENCES

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

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

[3] Damas, C., Lambeau, B. and van Lamsweerde, A., "Scenarios, Goals, and State Machines: A Win-Win Partnership for Model Synthesis", *Proc. FSE'06, 14th ACM Sigsoft Symp. on the Foundations of Software Engineering*, Portland (OR), November 2006.

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

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

[6] Darimont, R., Delor, E., Massonet, P. and van Lamsweerde, A., "GRAIL/KAOS: An Environment for Goal-Driven Requirements Engineering", *Proc. ICSE'98 - 20th Intl. Conf. on Software Engineering*, Kyoto, April 1998, vol. 2, 58-62.

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

[8] De Landtsheer, R., Letier, E. and van Lamsweerde, A., "Deriving Tabular Event-Based Specifications from Goal-Oriented Requirements Models", *Requirements Engineering Journal*, Vol.9 No. 2, 2004, 104-120.

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

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

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

[12] Jackson, M., "The World and the Machine", *Proc. ICSE'95: 17th International Conference on Software Engineering*, ACM Press, 1995, pp. 283-292.

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

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

[15] van Lamsweerde, A., "From System Goals to Software Architecture", in *Formal Methods for Software Architecture*, LNCS 2804, Springer-Verlag, 2003.

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

[17] van Lamsweerde, A., "Goal-Oriented Requirements Engineering: A Roundtrip from Research to Practice", Keynote paper, *Proc. RE'04, 12th IEEE Joint Intl. Requirements Engineering Conf.*, Kyoto, Sept. 2004, 4-8.

[18] van Lamsweerde, A., *Requirements Engineering: From System Goals to UML Models to Software Specifications*. Wiley, 2008.

[19] van Lamsweerde, A., Darimont, R. and Letier, E., "Managing Conflicts in Goal-Driven Requirements Engineering", *IEEE Trans. on Sofware. Engineering*, Vol. 24 No. 11, November 1998 , 908-926.

[20] van Lamsweerde, A., Darimont, R. and Massonet, Ph., "Goal-Directed Elaboration of Requirements for a Meeting Scheduler: Problems and Lessons Learnt", *Proc. RE'95 - 2nd Intl. IEEE Symp. on Requirements Engineering*, March 1995, 194-203.

[21] van Lamsweerde, A., Delcourt, B., Delor, E., Schayes, M.C. and Champagne, R., "Generic Lifecycle Support in the ALMA Environment", *IEEE Transactions on Software Engineering*, Vol. 14, No. 6, June 1988, 720-741.

[22] van Lamsweerde, A. and Letier, E. "Handling Obstacles in Goal-Oriented Requirements Engineering", *IEEE Trans. on Software Engineering*, Special Issue on Exception Handling, Vol. 26 No. 10, October 2000, 978-1005.

[23] van Lamsweerde, A. and Sintzoff, M., Formal Derivation of Strongly Correct Concurrent Programs. *Acta Informatica Vol.12*, Springer Verlag, 1979, 1-31.

[24] van Lamsweerde, A. and Willemet, L., "Inferring Declarative Requirements Specifications from Operational Scenarios", *IEEE Trans. on Sofware. Engineering*, Vol. 24 No. 12, December 1998 , 1089-1114.

[25] Letier, E. and van Lamsweerde, A., "Agent-Based Tactics for Goal-Oriented Requirements Elaboration", *Proc. ICSE'02: 24th Intl. Conf. on Software Engineering*, Orlando, IEEE Press, May 2002.

[26] Letier, E. and van Lamsweerde, A., "Deriving Operational Software Specifications from System Goals", *Proc. FSE'10:* 10th *ACM Symp. Foundations of Software Engineering*, Charleston, November 2002.

[27] Letier, E. and van Lamsweerde, A., "Reasoning about Partial Goal Satisfaction for Requirements and Design Engineering", *Proc. ACM FSE'04*, 2004, 53-62.

[28] Letier, E., Kramer, J., Magee, J., and S. Uchitel, "Deriving Event-based Transition Systems from Goal-oriented Requirements Models", *Automated Software Engineering* Vol. 15 No. 2, 2008, 175-206.

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

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

[31] Objectiver, http://www.objectiver.com.

[32] http://www.kinetium.com/map/demo/demo_index.html.

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

[34] Ponsard, C., Massonet, P. Molderez, J.F., Rifaut, A., and van Lamsweerde, A. "Early Verification and Validation of Mission-Critical Systems", *Formal Methods in System Design* Vol. 30 No. 3, Springer, June 2007, 233-247.

[35] Tran Van, H., van Lamsweerde, A., Massonet, P. and Ponsard, Ch., "Goal-Oriented Requirements Animation", *Proc. RE'04, 12th IEEE Joint International Requirements Engineering Conference*, Kyoto, Sept. 2004, 218-22.

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