Building Formal Requirements Models for Reliable Software[1]

Axel van Lamsweerde

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

Abstract. Requirements engineering (RE) is concerned with the elicitation of the goals to be achieved by the system envisioned, the operationalization of such goals into specifications of services and constraints, and the assignment of responsibilities for the resulting requirements to agents such as humans, devices, and software. Getting high-quality requirements is difficult and critical. Recent surveys have confirmed the growing recognition of RE as an area of primary concern in software engineering research and practice.

The paper first briefly introduces RE by discussing its main motivations, objectives, activities, and challenges. The role of rich models as a common interface to all RE processes is emphasized. We review various techniques available to date for system modeling, from semi-formal to formal, and discuss their relative strengths and weaknesses when applied during the RE stage of the software lifecycle.

The paper then discusses some recent efforts to overcome such problems through RE-specific techniques for goal-oriented elaboration of requirements, multiparadigm specification, the integration of non-functional requirements, the anticipation of abnormal agent behaviors, and the management of conflicting requirements.

1 Introduction

The requirements problem is among the oldest in software engineering. An early empirical study over a variety of software projects revealed that inadequate, inconsistent, incomplete, or ambiguous requirements are numerous and have a critical impact on the quality of the resulting software [Bel76]. Late correction of requirements errors was observed to be incredibly expensive [Boe81]. A consensus has been growing that engineering high-quality requirements is difficult; as Brooks noted in his landmark paper on the essence and accidents of software engineering, "the hardest single part of building a software system is deciding precisely what to build" [Bro87].

In spite of such early recognition, the requirements problem is still with us – more than ever. A recent survey over 8000 projects in 350 US companies showed

[1] Invited Paper, appeared in *Reliable Software Technologies*, Ada-Europe'2001, Springer-Verlag Lecture Notes in Computer Science, LNCS 2043, May 2001.

that only 16% of them were considered to be successful; 33% of them failed while 51% were in the space between, providing only partial functionalties, with major cost overruns and late deliveries. [Sta95]; when asked about the main reasons for this, 50% of the project managers mentioned poor requirements as the major source of failure. An independent survey of 3800 European organizations in 17 countries confirmed this; when asked where their main software problems are, more than half of the managers mentioned requirements specification and requirements management in first position [ESI96]. The problem gets even more serious in areas where reliability is a key concern. Many accidents in the safety-critical systems literature have been attributed to poor requirements engineering [Lev95].

Software requirements should thus be engineered with great care and precision. *Methods* should therefore be used to elicit them in a systematic way, organize them in a coherent structure, and check them against desired qualities such as completeness, consistency, and adequacy with respect to the real needs.

In spite of much recent interest in RE, the current state of the art in RE methods is still fairly limited [Lam2Ka]. There are many reasons for this, notably,

- the broad scope and inherent complexity of the RE process,
- some frequent misunderstanding of what the basic notions are really about (such as "requirements", "consistency", or "completeness"),
- the lack of abstraction of most software modeling techniques when used for engineering requirements,
- the lack of support for reasoning about the software-to-be and its environment taken together,
- the natural propensity to invent new notations and a posteriori checking techniques rather than constructive approaches,
- the conflicting concerns of formality (for analyzability) and simplicity (for usability).

This paper elaborates on some of the issues raised in this list. Section 2 introduces the scope and concerns of RE; we discuss the role of "rich" models as a common interface to the multiple activities involved in the RE process. Section 3 briefly reviews the main software modeling techniques available to date, from semi-formal to formal; we argue that these techniques while appropriate for the later stages of software design are not the ones needed in the context of engineering requirements. The second part of the paper then outlines an approach aimed at supporting the elaboration, structuring, and analysis of requirements models [Dar93, Lam98a, Lam2Kc]. The approach combines a goal-oriented method for deriving requirements and responsibility assignments from system objectives (Section 4), a systematic technique for generating exceptional behaviors to be handled in order to produce more complete and realistic requirements (Section 5), and a systematic technique for detecting and resolving conflicts among requirements as they usually arise from multiple viewpoints among the stakeholders involved (Section 6).

2 What Is RE Really About?

The RE process addresses three kinds of intertwined issues.

- ***WHY*** issues: The goals for a new system need to be identified, analyzed, and refined. Such goals are usually obtained by analyzing problems with the current situation, identifying new opportunities, exploring scenarios of interaction, and so on. Beside functional goals (e.g., satisfaction of requests, information of the state of affairs) there are many non-functional ones (e.g., safety, security, performance, evolvability, etc.). The identification and refinement of such goals usually makes heavy use of domain-specific knowledge.
- ***WHAT*** issues: The requirements operationalizing the various goals identified need to be defined precisely and related to each other; in parallel, the assumptions made in the operationalization process need to be made explicit and documented. Beside functional requirements about services to be provided there is a wide spectrum of non-functional requirements about quality of service.
- ***WHO*** issues: The requirements need to be assigned as contractual obligations among the various agents forming the composite system-to-be. These include the software to be developed, human agents, sensors/actuators, existing software, etc. The boundary between the software-to-be and its environment results from this distribution of responsibilities; different agent assignments define different system proposals.

Requirements engineering is thus by no means limited to WHAT issues as often suggested in the literature on software specifications.

In view of some confusions being frequently made, it is also worth clarifying what requirements are really about.

A first important distinction must be made between *requirements* and *domain properties* [Jac95, Par95]. Physical laws, organizational policies, regulations, or definitions of objects or operations in the environment are not requirements. For example, the precondition that the doors must be closed for an operation OpenDoors to be applied in a train control system is a domain property, not a requirement; on another hand, a precondition requiring that the train be at some station is a requirement on that same operation in order to achieve the goal of safe transportation.

A second important distinction must be made between requirements and software specifications. *Requirements* are formulated in terms of objects in the real world, in a vocabulary accessible to stakeholders [Jac95]; they capture required relations between objects in the environment that are monitored and controlled by the software, respectively [Par95]. *Software specifications* are formulated in terms of objects manipulated by the software, in a vocabulary accessible to programmers; they capture required relations between input and output software objects. *Accuracy goals* are non-functional goals at the boundary between the software and the environment that require the state of input/output software objects to accurately reflect the state of the corresponding monitored/controlled objects they represent [Myl92, Dar93]. In our train example, there should be an accuracy goal stating that the (physical) doors are open iff the corresponding Doors.State software variable has the value 'open'.

A third distinction has to be made between requirements and assumptions. Although they are both optative properties, *requirements* are to be enforced by the software whereas *assumptions* can be enforced by agents in the environment only [Lam98a]. For example, the software can enforce that trains be at station when doors get open, but cannot enforce that passengers get in.

If *R* denotes the set of requirements, *As* the set of assumptions, *D* the set of domain properties, *S* the set of software specifications, *Ac* the set of accuracy goals, and *G* the set of goals under consideration, the following satisfaction relations must hold:

$$S, Ac, D \models R \quad \text{with} \; S, Ac, D \not\models \textbf{false}$$

$$R, As, D \models G \quad \text{with} \; R, As, D \not\models \textbf{false}$$

The reasons why RE is such a complex step of the software lifecycle should now appear in this overall setting.

- The scope of RE is fairly broad. It addresses two systems –the system as it is and the system as it will be. It includes the software to be developed but also for the environment in which the software will operate. The latter may embody complex organizational policies or physical phenomena that need to be taken into account. The scope also covers a whole range of concerns and descriptions, from very high-level objectives to low-level constraints and from the initially vague to the eventually precise, sometimes formal.
- The RE process is composed of multiple intertwined subprocesses such as domain analysis, stakeholder analysis, elicitation of goals and scenarios, exploration of alternatives, risk analysis, negotiation, documentation of choices, specification, validation and verification, and change management.
- Usually there are many different parties involved which do not necessarily share the same objectives and background –clients, domain experts, managers, analysts, developers, etc.
- The large number and diversity of raised concerns inevitably leads to conflicts that need to be detected and resolved appropriately. In our train control example, the goal of safe transportation requires trains not to be too close to each other which conflicts with the goal of serving more passengers. In an electronic payment system, anonymity and accountability are conflicting goals, security through passwords conflicts with usability, and so on. Requirements engineers live in a world where conflicts are the rule, not the exception; conflict management is a driving force of the RE process.
- There are many other types of errors and deficiencies a requirements specification can contain beside conflicts and inconsistencies; some of them may be critical, such as incompleteness, inadequacies, or ambiguities; others, such as noises, overspecifications, forward references, or wishful thinking, are generally less severe but hamper understandability and generate new problems [Mey85]. Errors in requirements specifications are known to be numerous, persistent, expensive, and dangerous [Boe81, Lev95].

Rich *models* appear to be the best interfaces to the various RE subprocesses mentioned above. They provide a structured way of capturing the output of domain analysis and goal/requirement elicitation; they offer good support for exploring alternatives and negotiating choices; they provide structure to complex specifications and individual units for their compositional analysis; they may guide further elicitation, specification, and analysis. Models also provide the basis for documentation and evolution.

3 Candidate Techniques for Requirements Modeling and Specification

If modeling turns out to be a core activity in the RE process, the basic questions to be addressed are:

- what aspects to model in the *WHY-WHAT-WHO* range,
- how to model such aspects,
- how to define the model precisely,
- how to reason about the model.

The answer to the first question determines the *ontology* of conceptual units in terms of which models will be built - e.g., data, operations, events, goals, agents, and so forth. The answer to the second question determines the *structuring relationships* in terms of which such units will be composed and linked together - e.g., input/output, trigger, generalization, refinement, responsibility assignment, and so forth. The answer to the third question determines the informal, semi-formal, or formal *specification technique* used to define the properties of model components precisely. The answer to the fourth question determines the kind of *reasoning technique* available for the purpose of elicitation, specification, and analysis.

As will be seen below, the main candidate approaches for requirements modeling and specification are limited to *WHAT* aspects only. We review them briefly before arguing about their limitation for RE tasks.

3.1 Semi-Formal Approaches

The principle here is to *formally declare* conceptual units and their links; the properties of such units and links are generally *asserted informally*. We just list some of the main standard techniques.

Entity-Relationship Diagrams. The conceptual units denote autonomous classes of objects of interest with distinct identities and shared features; the conceptual links denote subordinate classes of objects of interest. Both can be characterized by attributes with specific value ranges. Diagrams can be hierarchically structured through specialization mechanisms. Figure 1 depicts a fragment for our train control example.

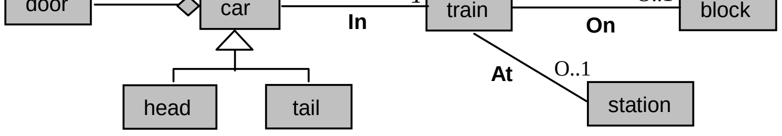


Figure 1 – Entities and relationships

State Transition Diagrams. The conceptual units denote object states of interest whereas the conceptual links denote guarded transitions triggered by events. Diagrams can be hierarchically structured through various parallel composition mechanisms. Figure 2 depicts a fragment for our train control example.

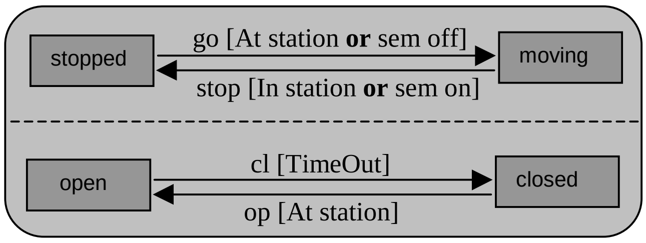

stopped
go [At station **or** sem off]
moving
stop [In station **or** sem on]
open
cl [TimeOut]
closed
op [At station]

Figure 2 – States and transitions

Dataflow Diagrams. The conceptual units denote operations of interest whereas the conceptual links denote input/output data flows. Diagrams can be hierarchically structured through functional decomposition mechanisms Figure 3 depicts a fragment for our train control example.

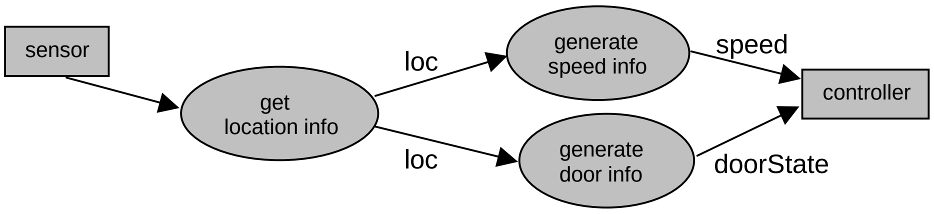


Figure 3 – Operations and data flows

The strengths of semi-formal approaches are fairly obvious:

- graphical notations are easier to use and communicate;
- different kinds of diagrams may provide complementary views of the same system; such views can be related to each other through inter-view consistency rules [Nus94].

These strengths together with notational standardization are probably the main reasons for the current popularity of UML [Rum99].

On the down side, these approaches have strong limitations:

- in general they can only cope with functional aspects;

- since they only capture declaration-level features of a system, they generally support highly limited forms of specification and analysis;
- the box-and-arrow semantics of their graphical notations is most often fairly fuzzy; the same model may often be interpreted in different ways by different people.

3.2 Formal Approaches

The principle here is to model a system in terms of structured collections of declarations and assertions, *both* specified in a formal language, which provides a precise syntax, semantics, and proof theory. The standard approaches here differ according to the specification paradigm they rely on (see [Lam2Kb] for more details and references).

History-Based Specification. A system is characterized by its maximal set of admissible behaviors over time. The properties of interest are specified by temporal logic assertions about system objects; such assertions involve operators referring to past, current and future states. The assertions are interpreted over time structures which can be discrete, dense or continuous. Most often it is necessary to specify properties over time bounds; real-time temporal logics are therefore necessary. In our train control example, we could specify a progress requirement about trains moving from one station to the other, such as

$$\begin{array}{l} \forall tr\text{: Train, } st\text{: Station} \\ \text{At }(tr, st) \Rightarrow \ \Diamond_{\leq T} \text{ At }(tr, \text{next}(st)) \end{array}$$

Our requirement of doors staying closed between stations would be specified by

$$\begin{array}{l} \forall tr\text{: Train, } st\text{: Station} \\ \bullet \text{ At }(tr, st) \wedge \neg \text{ At }(tr, st) \Rightarrow \\ \quad tr.\text{Doors} = \text{“closed”} \ \boldsymbol{U} \text{ At }(tr, \text{next}(st)) \end{array}$$

In the assertion above, $\Diamond_{\leq T}$ means “some time in the future before deadline T” [Koy92] whereas the $\bullet$ and $\boldsymbol{U}$ temporal operators mean “in the previous state” and “always in the future until”, respectively [Man92].

Built-in historical references allow the specifier to avoid introducing extra variables for encoding the past or the future in behavioral requirements; this paradigm seems thus particularly appropriate for the early stages of requirements specification.

State-Based Specification. Instead of characterizing the admissible system histories, one may characterize the admissible system states at some arbitrary snapshot. The properties of interest are specified by invariants constraining the system objects at any snapshot, and by pre- and post-assertions constraining the application of system operations at any snapshot. In our train control example, we could specify the operation to control door openings by a Z schema [Pot96] such as

```
OpenDoors ________________
|  Δ TrainSystem
|  st?: Station
|_________________________
|  Doors = "closed " ∧ At (tr, st?)
| Doors’ = “open”
```

The state-based paradigm is more operational than the previous one; it seems more appropriate for the later stages of requirements specification where specific software services have been elicited from more abstract requirements.

Transition-Based Specification. Instead of characterizing admissible system histories or system states, one may characterize the required transitions from state class to state class. The properties of interest are specified by a set of transition functions; the transition function for a system object gives, for each input state and triggering event, the corresponding output state. The occurrence of a triggering event is a sufficient condition for the corresponding transition to take place (unlike a precondition, it captures an obligation); necessary preconditions may also be specified to guard the transition. In our train control example, we could specify the required dynamics of doors by a SCR table [Par95, Heit96] such as

Old Mode	Event	New Mode
open	@T TimeOut	closed
closed	@T AtStation	open

Functional Specification. The principle here is to specify a system as a structured set of mathematical functions. Two approaches may be distinguished.

Algebraic specification. The functions are grouped by object types that appear in their domain or codomain, thereby defining abstract data types. The properties of interest are then specified as conditional equations that capture the effect of composing functions. In the context of our train control example, we might introduce a signature such as

```
WhichTrain: Blocks →Trains
EnterBlock: Trains × Blocks → Blocks
```

together with a law of composition such as

```
WhichTrain (EnterBlock (tr, bl)) =  tr
```

Higher-Order Functions. The functions are grouped into logical theories. Such theories contain type definitions, variable declarations, and axioms defining the various functions in the theory. Functions may have other functions as arguments which significantly increases the power of the language. In the context of our train control example, we might introduce a PVS [Owr95] function specification such as

```
TRACKING:  TYPE =  [Blocks →Trains]
trk: VAR TRACKING
AddTrain: [TRACKING, Blocks, Trains  → TRACKING ]
AddTrain (trk, bl, tr) =  trk    WITH   [ (bl) := tr ]
```

Operational Specification. Much closer to the programming level, a system may be characterized as a collection of concurrent processes that can be executed by some more or less abstract machine. Operational techniques such as Petri nets or process algebras rely on this general paradigm.

Formal approaches have numerous strengths:

- they offer precise rules of interpretation of statements;
- they support much more sophisticated forms of analysis, e.g., animation, algorithmic verification such as model checking, or deductive verification;
- they allow other useful products to be generated automatically, e.g., counterexamples, failure scenarios, test cases, proof obligations, refinements, code fragments, and so on.

On the down side, these approaches are not really accessible to practitioners. Formal specifications are hard to write, read, and communicate. Writing the input required by analysis tools is generally difficult, error-prone and thus requires high expertise; coping with their output is generally hard as well.

3.3 Common Inadequacies to RE Needs

Popular semi-formal and formal approaches both have common limitations with respect to the nature and scope of RE discussed in Section 2.

- *Restricted scope.* The approaches outined above address WHAT issues only; they do not cover WHY and WHO issues. RE methods need richer ontologies than those based on programming abstractions (viz. data, operation or state). To address WHY and WHO issues, RE ontologies must offer higher-level abstractions such as goals, goal refinements, agents, responsibility assignments, and so forth.
- *Alternative choices out of the picture.* The above approaches do not allow alternatives to be represented, related to each other, and compared for selection. RE methods must provide support for reasoning about alternative goal refinements, alternative conflict resolutions, alternative responsibility assignments, and so forth.
- *Non-functional aspects out of the picture.* The above approaches generally do not consider non-functional concerns. The latter form an important part of any requirements specification and play a prominent role in alternative selection, conflict management, and architecture derivation. RE methods must provide support for representing and reasoning about them.
- *Poor separation of concerns.* The above techniques in general make no distinction between domain properties, requirements, assumptions, and software specifications. In the above Z schema, for example, the domain property Doors = "closed " and the requirement At (tr, st?) have the same status. As discussed in Section 2, such distinctions are important in RE.
- *Monolithic frameworks.* With the techniques above there is no way to be formal at some critical places and semi-formal at others, or to combine multiple specification paradigms. In order to be usable, flexible, and customizable to specific types of concerns, RE methods should ideally be "multi-button", that is, support multiparadigm specification and integrate multiple forms of reasoning – from qualitative to formal, and from lightweight to heavyweight when needed.

- *Poor guidance.* Most techniques essentially provide sets of notations and tools for a posteriori analysis. Therefore they tend to induce an approach of elaborating models and specifications by iterative debugging. In view of the inherent complexity of the RE process, one should ideally favor a more constructive approach in which the quality of the requirements model and specification obtained is guaranteed by the method followed.

4 Shifting to Goal-Oriented RE

This section reviews some attempts to address those inadequacies, with special emphasis on a goal-oriented approach we have developed for that purpose.

Broadly speaking, a *goal* corresponds to an objective the system should achieve through cooperation of agents in the software-to-be and in the environment. It may refer to a functional or a non-functional concern.

Goals play a prominent role in the RE process [Dar93, Lam98a]. They drive the elaboration of requirements to support them. They provide a completeness criterion for the requirements specification; the specification is complete if all stated goals are met by the specification [Yue87]. Goals are generally more stable than the requirements to achieve them. They provide a rationale for requirements; a requirement exists because of some underlying goal which provides a base for it. In short, requirements "implement" goals much the same way as programs implement design specifications.

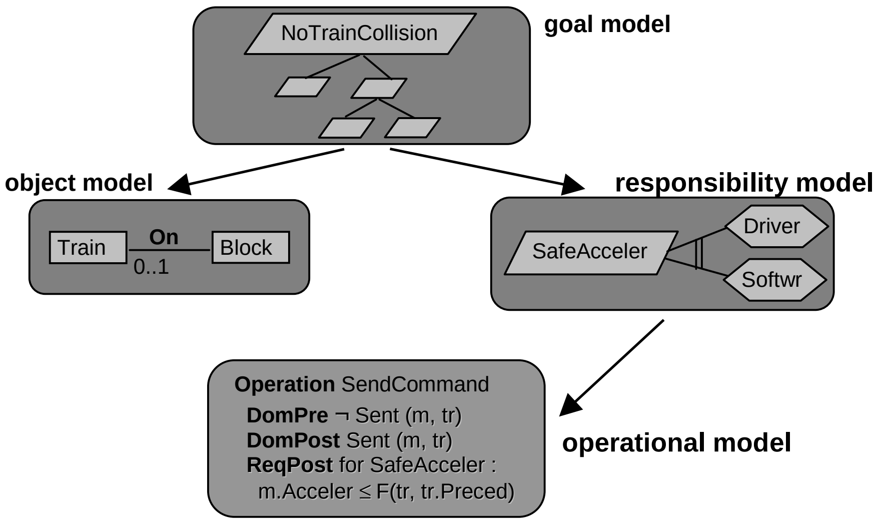
Goal-oriented approaches to RE are therefore receiving growing attention. Two complementary frameworks have been proposed for integrating goals and goal refinement in requirements models: a formal framework and a qualitative one.

In the *formal* framework [Dar93], goals can be specified semi-formally or formally. Goal refinements are captured through AND/OR graphs. AND-refinement links relate a goal to a set of subgoals (called refinement); this means that satisfying all subgoals in the refinement is a sufficient condition for satisfying the goal. OR-refinement links relate a goal to an alternative set of refinements; this means that satisfying one of the refinements is a sufficient condition for satisfying the goal. AND/OR operationalization links are also introduced to relate goals to requirements on operations/ objects; AND/OR responsibility links are introduced to relate primitive goals to individual agents. Goals are formalized in a real-time temporal logic whereas operations/objects are formalized by invariants and pre/post-conditions. Formal schemes are available for reasoning about goal refinement [Dar96], operationalization [Dar93, Let01], conflict [Lam98a], obstruction [Lam2Kc], assignment to agents [Let01], inference from scenarios [Lam98b], and acquisition by analogy [Mas97].

In the *qualitative* framework [Myl92], weaker link types are introduced to relate "soft" goals [Myl92]. The idea is that such goals cannot be said to be satisfied in a clear-cut sense. Instead of goal satisfaction, goal satisficing is introduced to express that lower-level goals or requirements are expected to achieve the goal within acceptable limits, rather than absolutely. A subgoal is then said to contribute partially to the goal, regardless of other subgoals; it may contribute positively or negatively. If a goal is AND-decomposed into subgoals and all subgoals are satisficed, then the goal is satisficeable; but if a subgoal is denied then the goal is deniable. If a goal contributes negatively to another goal and the former is satisficed, then the latter is deniable. In the AND/OR goal graph, goals are specified by names, parameters, and degrees of satisficing/denial by child goals. This framework is particularly well-suited for high-level goals that cannot be formalized . It can be used for evaluating alternative goal refinements. A qualitative labeling procedure may determine the degree to which a goal is satisficed/denied by lower-level requirements, by propagating such information along positive/negative support links in the goal graph.

The formal framework gave rise to the KAOS method for eliciting, specifying, and analyzing goals, requirements, scenarios, and responsibility assignments. Our aim here is to briefly suggest how the method works using a few excerpts from the requirements elaboration for a non-trivial, safety-critical system: the Bay Area Rapid Transit system [BART99, Let2K, Lam2Ka].

Figure 4 summarizes the steps of the method by showing the corresponding sub-models obtained. The goal refinement graph is elaborated by eliciting goals from available sources and asking *why* and *how* questions (goal elaboration step); objects, relationships and attributes are derived from the goal specifications (object modeling step); agents are identified, alternative responsibility assignments are explored, and agent interfaces are derived (responsibility assignment step); operations and their domain pre- and postconditions are identified from the goal specifications, and strengthened pre-/postconditions and trigger conditions are derived so as to ensure the corresponding goals (operationalization step). These steps are not strictly sequential as progress in one step may prompt parallel progress in the next one or backtracking to a previous one.



Operation SendCommand
DomPre ¬ Sent (m, tr)
DomPost Sent (m, tr)
ReqPost for SafeAcceler :
m.Acceler ≤ F(tr, tr.Preced)

Figure 4 – Goal-oriented RE with KAOS

Goal Identification from the Initial Document. A first set of goals is identified from a first reading of the available source [BART99] by searching for intentional keywords such as "objective", "purpose", "intent", "concern", "in order to", etc. A number of soft goals are thereby identified, e.g., "ServeMorePassengers", "NewTracksAdded", "Minimize[DevelopmentCosts]", "Minimize[DistanceBetweenTrains]", "SafeTransportation", etc. These goals are qualitatively related to each other through support links: Contributes (+), ContributesStrongly (++), Conflicts (-), ConflictsStrongly (- -). These weights are used to select among alternatives. Where possible, keywords from the semi-formal layer of the KAOS language are used to indicate the goal category. The *Maintain* and *Avoid* keywords specify "always" goals having the temporal pattern $\square (P \rightarrow Q)$ and $\square (P \rightarrow \neg Q)$, respectively. The *Achieve* keyword specifies "eventually" goals having the pattern $P \Rightarrow \Diamond Q$. The "$\rightarrow$" connective denotes logical implication; $\square (P \rightarrow Q)$ is denoted by $P \Rightarrow Q$ for short.

Figure 5 shows the result of this first elicitation. Clouds denote soft-goals, parallelograms denote formalizable goals, arrows denote goal-subgoal links, and a double line linking arrows denotes an OR-refinement into alternative subgoals.

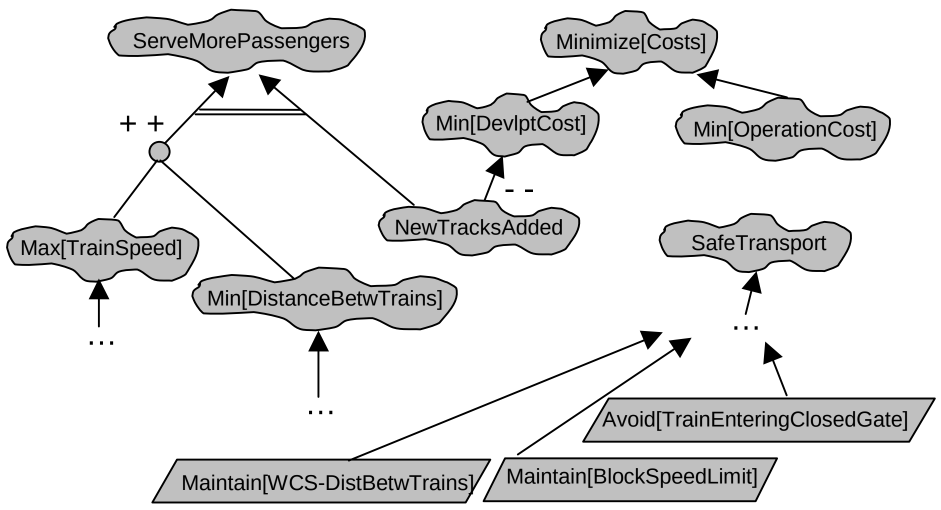
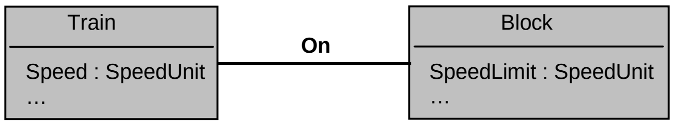


Figure 5 – Preliminary goal graph for the BART system

Formalizing Goals and Deriving the Object Model. The object modeling step can start as soon as goals can be formulated precisely enough. The principle here is to identify objects, relationships and attributes from goal specifications. Consider, for example, the goal Maintain[BlockSpeedLimit] at the bottom of Figure 5. It may be specified as follows:

Goal Maintain[BlockSpeedLimit]
InformalDef *A train should stay below the maximum speed the track segment can handle.*
FormalDef $\forall$ tr: Train, bl: Block:
On(tr, bl) $\Rightarrow$ tr.Speed $\leq$ bl.SpeedLimit

From the predicate, objects, and attributes appearing in this goal formalization we derive the following portion of the object model:



Similarly, the other goal at the bottom of Figure 5 is specified as follows:

Goal Maintain[WCS-DistBetweenTrains]
InformalDef *A train should never get so close to a train in front so that if the train in front stops suddenly (e.g., derailment) the next train would hit it.*
FormalDef $\forall$ tr1, tr2: Train :
Following (tr1, tr2) $\Rightarrow$ tr1.Loc - tr2.Loc > tr1.WCS-Dist

(The InformalDef statements in those goal definitions are taken literally from the initial document; *WCS-Dist* denotes the physical worst-case stopping distance based on the physical speed of the train.) This new goal specification allows the above portion of the object model to be enriched with *Loc* and *WCS-Dist* attributes for the *Train* object together with a reflexive *Following* relationship on it. *Goals thus provide a precise driving criterion for identifying elements of the object model*.

Eliciting More Abstract Goals by WHY Questions. It is often the case that higher-level goals underpinning goals easily identified from initial sources are kept implicit in such sources. They may, however, be useful for finding out other important subgoals of the higher-level goal that were missing for the higher-level goal to be achieved.

Higher-level goals are identified by asking WHY questions about the goals available. For example, asking a WHY question about the goal

Maintain[WCS-DistBetweenTrains]

yields the parent goal

Avoid[TrainCollision]

On another hand, asking a WHY question about the goal

Avoid[TrainEnteringClosedGate]

yields the parent goal

Avoid[TrainOnSwitchInWrongPosition].

The formalizations of this parent goal and of the initial subgoal Avoid[TrainEnteringClosedGate] match the root and one of the two child nodes of a formal refinement pattern from our pattern library [Dar96, Let2K]. This pattern, pre-proved once for all to produce a correct and complete goal refinement using a temporal logic verifier, reveals by reinstantiation that the companion subgoal was missing from the initial document, that is, the goal

Maintain[GateClosedWhenSwitchInWrongPosition].

Missing goals can thus be discovered formally by a combination of WHY questions and refinement patterns.

Eliciting More Concrete Goals by HOW Questions. Goals need to be refined until subgoals are reached that can be assigned to individual agents in the software-to-be and in the environment. Terminal goals become requirements in the former case and assumptions in the latter.

More concrete subgoals are elicited by asking HOW questions. For example, a HOW question about the goal

Maintain[WCS-DistBetweenTrains]

Yields the following three companion subgoals:

Maintain [SafeSpeed/AccelerationCommanded],
Maintain [SafeTrainResponseToCommand],
Maintain [NoSuddenStopOfPrecedingTrain].

The formalization of these subgoals may be used to formally prove that together they entail the father goal Maintain[WCS-DistBetweenTrains] formalized above [Let2K]. These subgoals have to be refined in turn until assignable subgoals are reached. A complete refinement tree may be found in [Lam2Ka].

Exploring Alternative Responsibility Assignments. The responsibility assignment step relies on precise formulations of goals from the goal elaboration step. Assignments of individual agents to terminal goals in the refinement graph are captured by AND/OR responsibility links. For example, the initial BART document suggests assigning the *Accuracy* goal

Maintain[AccurateSpeed/PositionEstimates]

to the TrackingSystem agent, the goal

Maintain[SafeTrainResponseToCommand]

to the OnBoardTrainController agent, and the goal

Maintain[SafeCmdMsg]

to the Speed/AccelerationControlSystem agent.

Alternative goal refinements and agent assignments could be explored. For example, the parent goal

Maintain[WCS-DistBetweenTrains]

may alternatively be refined by the following three *Maintain* subgoals:

PreceedingTrainSpeed/PositionKnownTo**Following**Train,
SafeAccelerationBasedOn**PreceedingTrain**Speed/Position,
NoSuddenStopOfPreceedingTrain

The second subgoal could now be assigned to the **OnBoard**TrainController agent. This alternative responsibility assignment would produce a fully distributed system. Qualitative reasoning techniques in the style of [Myl99] might then be applied to the soft goals identified in Figure 5 to help selecting the most preferable responsibility assignment.

Deriving Agent Interfaces. Once terminal subgoals have been assigned to individual software or environmental agents, the interfaces of each agent in terms of monitored and controlled variables can be derived systematically from the goal specifications. The formal technique is described in [Let01]; we just suggest the idea here on a simple example. Consider the goal Maintain[SafeCmdMsg] that has been assigned to the Speed/AccelerationControlSystem agent. We give its general form here for sake of simplicity:

Goal Maintain[SafeCmdMsg]
FormalDef $\forall$ cm: *CommandMessage*, tr1, tr2: *Train*
Sent (cm, tr1) $\wedge$ Following (tr1, tr2) $\wedge$ Refers (cm, tr2.Info)
$\Rightarrow$ cm.Accel $\leq$ F (tr1, tr2) $\wedge$ cm.Speed $>$ G (tr1)

To fulfil its responsibility for this goal the Speed/AccelerationControlSystem agent must be able to *evaluate* the goal antecedent and *establish* the goal consequent. The agent's monitored variable is therefore *Train.Info* whereas its controlled variables are *CommandMessage.Accel* and *CommandMessage.Speed*. The latter will in turn become monitored variables of the OnBoardTrainController agent, by similar analysis.

Identifying Operations. The final operationalization step starts by identifying the operations relevant to goals and defining their domain pre- and postconditions. Goals refer to specific state transitions; for each such transition an operation causing it is identified; its domain pre- and postcondition captures the state transition. For the goal Maintain[SafeCmdMsg] formalized above we get, for example,

Operation SendCommandMessage
Input Train {**arg** tr}
Output ComandMessage {**res** cm}
DomPre $\neg$ Sent (cm, tr)
DomPost Sent (cm, tr)

This definition minimally captures what any sending of a command to a train is about in the domain considered; it does not ensure any of the goals it should contribute to.

Operationalizing Goals. The next operationalization sub-step is to strengthen such domain conditions so that the various goals linked to the operation are ensured. For goals assigned to software agents, this step produces *requirements* on the operations for the corresponding goals to be achieved. Derivation rules for an operationalization calculus are available [Dar93, Let01]. In our example, they yield the following requirements that strengthen the domain pre- and postconditions:

Operation SendCommandMessage
Input …; **Output** …
DomPre ... ; **DomPost** ...
ReqPost for SafeCmdMsg:
Following (tr, t2)
$\rightarrow$ cm.Accel $\leq$ F (tr, tr2) $\wedge$ cm.Speed $>$ G (tr)
ReqTrig for CmdMsgSentInTime:
$\blacksquare_{\leq 0.5 \text{ sec}}$ $\neg\,\exists$ cm': CommandMessage:
Sent (cm', tr)

(The trigger condition captures an obligation to trigger the operation as soon as the condition gets true, and provided the domain precondition is true. In the example above the condition says that no command has been sent in every past state up to one half-second [BART99].)

Using a mix of semi-formal and formal techniques for goal-oriented requirements elaboration, we have reached the level at which most formal specification techniques would start.

5 Analyzing Obstacles to Requirements Satisfaction

First-sketch specifications of goals, requirements and assumptions are often too ideal; they are likely to be violated from time to time in the running system due to unexpected behavior of agents. The lack of anticipation of exceptional behaviors may result in unrealistic, unachievable and/or incomplete requirements. We capture such exceptional behaviors by formal assertions called *obstacles* to goal satisfaction.

An obstacle *O* is said to obstruct a goal *G* iff

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

$Dom \not\models \neg O$ *domain consistency*

Obstacles need to be identified and resolved at RE time in order to produce robustness requirements and hence more reliable software. We have developed a set of formal and heuristic techniques for:

- the abductive generation of obstacles from goal specifications and domain properties,
- the systematic generation of various types of obstacle resolution, e.g., goal substitution, agent substitution, goal weakening, goal restoration, obstacle mitigation, or obstacle prevention.

The interested reader may refer to [Lam2Kc] for details. We just illustrate a few results from obstacle analysis for some of the terminal goals in the goal refinement graph of the BART system.

The following obstacles were generated to obstruct the subgoal Achieve[CommandMsgIssuedInTime]:

CommandMsgNotIssued,
CommandMsgIssuedLate,
CommandMsgSentToWrongTrain

For the companion subgoal Achieve[CommandMsgDeliveredInTime] we similarly generated obstacles such as:

CommandMsgDeliveredLate,
CommandMsgCorrupted

The last companion subgoal Maintain[SafeCmdMsg] may be obstructed by the condition

UnsafeAcceleration,

and so on. The obstacle generation process for a single goal results in a goal-anchored fault-tree, that is, a refinement tree whose root is the goal negation. Compared with standard fault-tree analysis [Lev95], obstacle analysis is goal-oriented, formal, and produces obstacle trees that are provably complete with respect to what is known about the domain [Lam2Kc].

Alternative obstacle resolutions may then be generated to produce new or alternative requirements. For example, the obstacle CommandMsgSentLate above could be resolved by an alternative design in which accelerations are calculated by the on-board train controller instead; this would correspond to a *goal substitution* strategy. The obstacle UnsafeAcceleration above could be resolved by assigning the responsibility for the subgoal SafeAccelerationCommanded of the goal Maintain[SafeCmdMsg] to the VitalStationComputer agent instead [BART99]; this would correspond to an *agent substitution* strategy. An *obstacle mitigation* strategy could be applied to resolve the obstacle OutOfDateTrainInfo obstructing the accuracy goal Maintain[AccurateSpeed/PositionEstimates], by introducing a new subgoal of the goal Avoid[TrainCollisions], namely, the goal Avoid[CollisionWhenOutOfDateTrainInfo]. This new goal has to be refined in turn, e.g., by subgoals requiring full braking when the message origination time tag has expired.

6 Handling Conflicting Requirements

As mentioned before, requirements engineers live in a world where conflicts are the rule, not the exception [Eas94]. Conflicts generally arise from multiple viewpoints and concerns. They must be detected and eventually resolved even though they may be temporarily useful for eliciting further information [Hun98]. In [Lam98] we have studied various forms of conflict and, in particular, a weak form called *divergence* which occurs frequently in practice.

The goals G_1, ..., G_n are said to be divergent iff there exists a non-trivial *boundary condition B* such that :

$$\{ B, \forall i\ G_i, Dom\} \models \textbf{false} \qquad \textit{inconsistency}$$
$$\{ B, \forall_{j \neq i}\ G_j, Dom\} \not\models \textbf{false} \qquad \textit{minimality}$$

("Non-trivial" means that B is different from the bottom **false** and the complement $\neg \forall i\ G_i$). Note that the traditional case of conflict, in the sense of logical inconsistency, amounts to a particular case of divergence.

Divergences need to be identified and resolved at RE time in order to eventually produce consistent requirements and hence more reliable software. We have also developed a set of formal and heuristic techniques for:

- the abductive generation of boundary conditions from goal specifications and domain properties,
- the systematic generation of various types of divergence resolution.

The interested reader may refer to [Lam98] for details. The initial BART document suggests an interesting example of divergence [BART99, p.13]. Roughly speaking, the train commanded speed may not be too high, because otherwise it forces the distance between trains to be too high, in order to achieve the DistanceIncreasedWithCommandedSpeed subgoal of the SafeTransportation goal; on the other hand, the commanded speed may not be too low, in order to achieve the LimitedAccelerAbove7mphOfPhysicalSpeed subgoal of the SmoothMove goal. There seems to be a flavor of divergence here. We therefore look at the formalization of the suspect goals:

Goal Maintain [CmdedSpeedCloseToPhysicalSpeed]
FormalDef $\forall$ tr: Train

$$\text{tr.CmAccel} \geq 0 \Rightarrow \text{tr.CmSpeed} \leq \text{tr.Speed} + f\ (\text{dist-to-obstacle})$$

and

Goal Maintain [CmdedSpeedAbove7mphOfPhysicalSpeed]

FormalDef $\forall$ tr: Train

$$\text{tr.CmAccel} \geq 0 \Rightarrow \text{tr.CmSpeed} > \text{tr.Speed} + 7$$

These two goals are formally detected to be divergent using the regression technique described in [Lam98]. The generated boundary condition for making them logically inconsistent is

$$\Diamond \exists \text{ tr: Train}$$
$$\text{tr.CmAccel} \geq 0 \ \wedge\ f\ (\text{dist-to-obstacle}) \leq 7$$

The resolution operators from [Lam98] may be used to generate alternative resolutions; in this case one should keep the safety goal as it is and *weaken* the other conflicting goal to remove the divergence:

Goal Maintain [CmdedSpeedAbove7mphOfPhysicalSpeed]

FormalDef $\forall$ tr: Train

$$\text{tr.CmAccel} \geq 0 \Rightarrow \text{tr.CmSpeed} > \text{tr.Speed} + 7 \ \vee\ \mathbf{f\ (dist\text{-}to\text{-}obstacle)} \leq \mathbf{7}$$

7 Conclusion

Standard modeling and specification techniques were reviewed in this paper to argue that most of them are inappropriate to the scope, concerns, processes, and actors involved in requirements engineering. Although they provide useful paradigms for RE methodologies, these techniques provide too low-level ontologies, do not support the representation and exploration of alternatives, mix up different kinds of assertions, are too monolithic, and provide little guidance in the requirements elaboration process.

We used a real, complex safety-critical system as a running example to show the benefits of a constructive, multiparadigm, and goal-oriented approach to requirements modeling, specification and analysis. The key points illustrated are the following:

- object models and operational requirements can be derived constructively from goal specifications;
- goals provide the rationale for the requirements that operationalize them, and a correctness criterion for requirements completeness;
- the goal refinement structure provides a rich way of structuring the entire requirements document;
- alternative system proposals are explored by alternative goal refinements and assignments;
- a multiparadigm, multibutton framework allows one to combine different levels of expression and reasoning: semi-formal for modeling and navigation, qualitative for selection among alternatives, and formal, when needed, for more accurate reasoning;

- goal formalization allows RE-specific types of analysis to be carried out, such as
  - checking the correctness and completeness of a goal refinement,
  - completing an incomplete refinement,
  - generating obstacles to requirements satisfaction, and resolutions to yield new requirements for more robust systems,
  - generating boundary conditions for conflict among requirements together with alternative resolutions.

Goals, especially non-functional ones, may also play a leading role in the process of deriving a software architecture from requirements, and of defining architectural views [Lam98a]. Goal-based reasoning is thus central to RE but also to architectural design. From our experience in using KAOS in a wide variety of industrial projects at our tech transfer institute, we have observed that domain experts, managers, and decision makers are in fact much more interested by goal structures than, e.g., UML models. Getting such early involvement and feedback turns out to be crucial to the development of reliable software, as the empirical studies mentioned at the beginning of this paper suggest.

References

[BART99] Bay Area Rapid Transit District, Advance Automated Train Control System, Case Study Description. Sandia National Labs, http://www.hcecs.sandia.gov/bart.htm.

[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.

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

[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”, *Proc. FSE’4 - Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, San Francisco, October 1996, 179-190.

[Eas94] S. Easterbrook, “Resolving Requirements Conflicts with Computer-Supported Negotiation”. In *Requirements Engineering: Social and Technical Issues*, M. Jirotka and J. Goguen (Eds.), Academic Press, 1994, 41-65.

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

[Heit96] C. Heitmeyer, R. Jeffords and B. Labaw, “Automated Consistency Checking of Requirements Specificatons”, *ACM Transactions on Software Engineering and Methodology* Vol. 5 No. 3, July 1996, 231-261.

[Hun98] A. Hunter and B. Nuseibeh, “Managing Inconsistent Specifications: Reasoning, Analysis and Action”, *ACM Transactions on Software Engineering and Methodology,* Vol. 7 No. 4. October 1998, 335-367.

[Jac95] M. Jackson, Software Requirements & Specifications - A Lexicon of Practice, Principles and Pejudices. ACM Press, Addison-Wesley, 1995.

[Koy92] R. Koymans, Specifying message passing and time-critical systems with temporal logic, LNCS 651, Springer-Verlag, 1992.

[Lam98a] A. van Lamsweerde, R. Darimont and E. Letier, “Managing Conflicts in Goal-Driven Requirements Engineering”, *IEEE Trans. on Sofware. Engineering,* Special Issue on Inconsistency Management in Software Development, November 1998.

[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.

[Lam2Ka] A. van Lamsweerde, “Requirements Engineering in the Year 00: A Research Perspective”, Keynote paper, *Proc. ICSE’2000 - 22nd Intl. Conference on Software Engineering,* IEEE Press, June 2000.

[Lam2Kb] A. van Lamsweerde, “Formal Specification: a Roadmap”. In *The Future of Software Engineering*, A. Finkelstein (ed.), ACM Press, 2000.

[Lam2Kc] A. van Lamsweerde and E. Letier, “Handling Obstacles in Goal-Oriented Requirements Engineering”, *IEEE Transactions on Software Engineering*, Special Issue on Exception Handling, October 2000.

[Let2K]þE. Letier and A. van Lamsweerde, “KAOS in Action: the BART System”. IFIP WG2.9 meeting, Flims, http:// www.cis.gsu.edu/~wrobinso/ifip2_9/Flims00.

[Let01] E. Letier, Reasoning About Agents in Goal-Oriented Requirements Engineering. PhD Thesis, University of Louvain, 2001.

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

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

[Mas97] P. Massonet and A. van Lamsweerde, “Analogical Reuse of Requirements Frameworks”, Proc. RE-97 - *3rd Int. Symp. on Requirements Engineering*, Annapolis, 1997, 26-37.

[Mey85] B. Meyer, “On Formalism in Specifications”, IEEE Software, Vol. 2 No. 1, January 1985, 6-26.

[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.

[Myl99] J. Mylopoulos, L. Chung and E. Yu, "From Object-Oriented to Goal-Oriented Requirements Analysis", Communications of the ACM, Vol. 42 No. 1, January 1999, 31-37.

[Nus94] B. Nuseibeh, J. Kramer and A. Finkelstein, “A Framework for Expressing the Relationships Between Multiple Views in Requirements Specifications”, *IEEE Transactions on Software Engineering*, Vol. 20 No. 10, October 1994, 760-773.

[Owr95] S. Owre, J. Rushby, and N. Shankar, “Formal Verification for Fault-Tolerant Architectures: Prolegomena to the Design of PVS”, *IEEE Transactions on Software Engineering* Vol. 21 No. 2, Feb. 95, 107-125.

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

[Pot96] B. Potter, J. Sinclair and D. Till, *An Introduction to Formal Specification and Z.* Second edition, Prentice Hall, 1996.

[Rum99] J. Rumbaugh, I. Jacobson and G Booch, The Unified Modeling Language Reference Manual. Addison-Wesley, Object Technology Series, 1999.

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

[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*, IEEE, 1987.