Goal-Directed Elaboration of Requirements for a Meeting Scheduler: Problems and Lessons Learnt

Axel van Lamsweerde, Robert Darimont and Philippe Massonet

Université Catholique de Louvain, Département d'Ingénierie Informatique
B-1348 Louvain-la-Neuve (Belgium)

(Proceedings RE'95 - 2nd IEEE Symposium on Requirements Engineering, York, March 1995, 194-203)

Abstract - *Recently a number of requirements engineering languages and methods have flourished that do not only address* what *questions but also* why, who *and* when *questions. The objective of this paper is twofold: (i) to assess the strengths and weaknesses of one of these methodologies on a non-trivial benchmark, and (ii) to illustrate and discuss a number of challenging issues that need to be addressed for such methodologies to become effective in supporting real, complex requirements engineering tasks. The problem considered here is that of a distributed meeting scheduler system; the methodology considered is the KAOS goal-directed language and method. The issues raised from this case study include goal identification, the "deidelization" of unachievable goals, the handling of interfering goals, the impact of early formal reasoning, the merits of early reuse of abstract descriptions and categories, requirements traceability and the need to link requirements to retractable assumptions, and the potential benefits of hybrid acquisition strategies.*

1. Introduction

Requirements engineering encompass three concurrent activities: requirements *acquisition*, where alternative models for the target composite system are elaborated, requirements *specification*, where the various components of such models are made fully precise and possibly formalized, and requirements *evaluation*, where the specifications are analyzed against *correctness* properties (such as consistency, completeness and adequacy with respect to the actual needs) and *feasibility* properties (such as costs and resources required). At the end one of those alternative models for the target system is selected for subsequent implementation.

The term "composite system" refers to the automated system together with the environment with which it interacts. Modeling composite systems allows the interaction between human and automated agents to be made explicit. Formal reasoning on such interactions can be supported thereby [Fea87], [Fic92]. Henceforth a model for the target composite system will be called *requirements model*.

Most research efforts so far have been devoted to the requirements specification facet of requirements engineering. Myriads of languages have been proposed, some of which are becoming increasingly popular -e.g., Z [Spi92], VDM [Jon90] or LARCH [Gut93]. The latter languages, however, are not well suited for capturing requirements models because they are too restricted in scope; they address *what* questions only. Typically, the data and operations of the system envisioned are specified through first-order assertions like conditional equations or pre-, postconditions and invariants. Another limitation is that such languages have no built-in constructs for making a clear separation between domain descriptions and actual requirements [Jac93].

Recent attempts have been made to design languages and methods that support a much wider range of requirements. Within such frameworks *why*, *who*, and *when* questions can be addressed in addition to the usual *what* questions. It then becomes possible to reason formally about goal satisfaction [Rob89], [Dar91], [Myl92], [Dar93], [Fea93], agent responsibilities [Fea87], [Fin87], [Dar91], [Fic92], [Dar93], [Ken93], [Dub93] or triggering events and causalities [Fin87], [Dar93] to support the requirements acquisition and evaluation process.

The purpose of this paper is to assess the current strengths and weaknesses of one typical such approach to requirements acquisition, specification and evaluation. Our assessment is based on a rich, non-trivial benchmark. The approach selected is the one we are most familiar with -that is, the KAOS conceptual modelling language and requirements elaboration strategy [Dar91], [Dar93]. The benchmark proposed is a distributed meeting scheduler system; it was selected for two main reasons. First, the same persons could play the three different roles usually involved in the requirements engineering process -the client role, the domain expert role, and the specifier role. Second, this system provides a rich combination of challenging features - e.g., interfering goals, real-time and distribution aspects, multi-agent cooperation and communication, privacy concerns, optimization requirements, heuristics for defining near solutions, etc. Also the space of alternative decisions and compromises to be made throughout the process is fairly large. (Others have started working with the meeting scheduler exemplar, see, e.g., [Pot94]. The initial 4-page problem statement [Vla93] can be obtained from the authors on request.)

The focus here will be on the actual process followed by the authors working in parallel to acquire a significant portion of the requirements model. For each step of the strategy we will illustrate the problems that we could handle more or less satisfactorily and the problems that we could not. The most interesting and challenging aspects are obviously in the latter category. We strongly believe that many of these problems are not specific to the KAOS approach and need to be addressed seriously in the future.

The KAOS approach to requirements elaboration is first summarized briefly in Section 2. Section 3 then follows the goal-directed strategy step by step and discusses the various issues of interest from representative excerpts of our meeting scheduler specification. These issues are put together into perspective in the general discussion that follows in Section 4.

2. Requirements Elaboration in KAOS

The KAOS approach draws on a number of concepts and techniques from Artificial Intelligence work on knowledge representation and acquisition [Vla91b], [Dar93]. Requirements models are acquired as instances of a *conceptual meta-model*. The latter can be represented as a graph where each node captures an abstraction such as, e.g., goal, action, agent, entity, or event, and where the edges capture semantic links between such abstractions. Well-formedness properties on nodes and links constrain instances of such abstractions -that is, elements of requirements models.

Requirements acquisition processes then correspond to particular ways of traversing the meta-model graph to acquire appropriate instances of the various nodes and links according to such constraints. Acquisition processes are governed by *strategies* telling which way to follow systematically in that graph; at each node specific *tactics* can be used to acquire the corresponding instances.

2.1. Capturing requirements models

A distinction is thus made between the following three levels. The **meta** level defines meta-*concepts*, that is, abstractions supported by the specification language (such as "goal", "object", "agent", "action", etc.), meta-*relationships* that link such abstractions (such as goal "reduction" into subgoals, goal "operationalization" into constraints, agent "assignment" to constraints, etc.), meta-*attributes* that characterize meta-concepts/relationships (such as agent "load", "reliability" of assignment, etc.) and meta-*constraints* that constrain such abstractions and links (e.g., "a weak constraint must have a restoration action associated with it"). The **domain** level defines domain-specific instances of meta-concepts and meta-relationships, such as the "meeting" concept which is an instance of the "entity" meta-concept, the "scheduler" concept which is an instance of the "agent" meta-concept, the "MeetingRequestSatisfied" concept which is an instance of the "goal" meta-concept, the "agenda" concept which is an instance of the "relationship" meta-concept, etc. The **instance** level is made of specific instances of domain-level concepts, such as "the ICSE-16 Program Committee meeting" which is an instance of the "meeting" concept.

The KAOS meta-model is a model for the meta-level. It captures meta-level knowledge to be used at the domain level to guide the acquisition process. For example, knowledge of a meta-constraint such as "any weak constraint must have a restoration action associated with it" may prompt the acquisition system to ask the user what the restoration action should look like for a constraint like "the participants agendas and their automated image should be kept consistent", if this constraint has been declared to be a weak constraint. *The more knowledge at the meta level, the more knowledge-based guidance can be provided to acquire requirements fragments at the domain level.* This principle is at the core of several machine learning systems [Vla91a].

The following meta-concepts/relationships will be used in the sequel; see [Dar93] for their precise characterization.

- *Object:* an object is a thing of interest in the domain whose instances may evolve from state to state. A domain object is in general declared in a more specialized way -as an *entity*, *relationship*, or *event* according as the object is an autonomous, subordinate, or instantaneous object, respectively. Objects are characterized by invariant assertions.
- *Action:* an action is a (mathematical) input-output relation over objects; action applications define state transitions. Actions may be *caused/stopped* by events. They are characterized by pre-, post- and trigger conditions.
- *Agent:* an agent is another kind of object which acts as processor for some actions. An agent *performs* an action if it is effectively allocated to it; the agent *knows* an object if the states of the object are made observable to it. Agents can be humans, devices, programs, etc.
- *Goal:* a goal is a *nonoperational* objective that the composite system must meet. (An objective is said to be nonoperational if it cannot be formulated in terms of states that are controllable by some agent.) Goals can be AND/OR *reduced* into sub-goals, e.g., a goal *G* may be achieved through subgoals *G1* and *G2* or through subgoals *G3* and *G4*. The goal structure for a given system is thus in general an AND/OR directed acyclic graph. Goals often *conflict* with others so that conflict resolution tactics must be applied at some stage or another of the requirements acquisition process. (The distinction between SystemGoals, that *must* be achieved, and PrivateGoals associated with specific agents is introduced for that purpose.) Goals *concern* objects. They are classified by pattern of temporal behaviour they require (e.g., *Achieve*, *Cease*, *Maintain*, *Avoid*, *Optimize*) and by type of requirement they express with respect to the agents concerned (e.g., *SatisfactionGoal*, *InformationGoal*, *ConsistencyGoal*, *SafetyGoal*, *PrivacyGoal*, etc.). Such categories are used, among others, when reuse tactics are applied to retrieve generic or similar goal descriptions from the domain knowledge base.
- *Constraint* (or *policy*): a constraint is an *operational* objective, that is, an objective that can be formulated in terms of states controllable by some agent. Goals must be AND/OR *operationalized* into constraints. Constraints in turn are AND/OR *ensured* by actions and

objects through restricted conditions that strengthen their pre-, post-, trigger conditions and invariants, respectively. Alternative ways of assigning responsible agents to a constraint are captured through AND/OR *responsibility* links; the actual assignment of agents to the actions that ensure the constraint is captured in the corresponding *performance* links. Two classes of constraints are distinguished in the meta-model. *StrongConstraints* may never be violated (e.g., SafetyGoals are Avoid goals that must be operationalized into StrongConstraints.) *WeakConstraints* are likely to be temporarily violated; they need specific restoration actions to be associated with them.

- *Scenario:* a scenario expresses a typical combination of actions expected to take place in the composite system. It is composed from actions using parallel, sequential, repetitive and alternative combination modes.

The specification language in which requirements models are elaborated is traced from the meta-model. The language has a two-level structure: an outer level for *declaring* domain-level concepts in terms of meta-level concepts/links, and an inner level for expressing *assertions* that further characterize those concepts. The outer declaration level has an entity-relationship structure; the inner assertion level amounts to a typed first-order temporal logic equipped with real-time constructs [Koy92]. The use of this language will be illustrated extensively in Section 3.

2.2. A goal-directed acquisition strategy

Requirements about the composite system are acquired as domain-specific instances of elements of the conceptual meta-model. Such instances must satisfy the meta-constraints specified once for all -like cardinality constraints on meta-relationships or various other meta-level “axioms”.

Acquisition processes are guided by strategies and domain models. *Strategies* define specific ways of traversing the meta-model graph to acquire instances of its various nodes and links. Each step in a strategy is itself composed from finer steps like question-answering, input validation against meta-constraints, application of tactics to select preferred alternatives for the various AND/OR meta-relationship instances that arise during acquisition, deductive inferencing based on property inheritance through specialization links, or analogical reuse of domain models. *Domain models* are described in the same specification language as requirements are. They are organized as *IsA* inheritance hierarchies in the domain knowledge base [Reu91]. In the KAOS context, the latter contains various levels of specialization of goals, constraints, objects, actions and agents involved in resource management systems, transportation systems, communication systems, etc. [Dam93]. Ultimately the acquisition assistant's knowledge base should include a rich variety of domain models, strategies and tactics.

The strategy that has been considered mainly so far is a *goal-directed* one. It is made of the following steps.

(1) Acquisition of the goal structure and identification of the objects concerned by the goals;
(2) Preliminary identification of potential agents together with the actions such agents are capable of;
(3) Operationalization of goals into constraints;
(4) Refinement of objects and actions;
(5) Derivation of strengthened conditions on the actions and objects in order to ensure constraints;
(6) Identification of alternative agent responsibilities;
(7) Actual assignment of actions to responsible agents.

In this strategy, the steps are ordered but some of them may be running concurrently (notably, steps 1, 2 and 3). Moreover, backtracking is possible at every step. For example, information acquired during the responsibility identification step (step 6) may entail changes to the results of the operationalization step (step 3). The changes made to the latter step must then be propagated through the succeeding steps.

Tactics are proposed in [Dar93] for each step of this strategy. For example, “*Reduce goals into subgoals so that the latter require co-operation of fewer potential agents to achieve them*” is a tactics for step (1) above.

3. The Meeting Scheduler Case Study

We illustrate on a few examples how the *actual* elaboration proceeded according to the above strategy (and, in particular, where backtracking and "coroutining" were needed). For each step of the strategy, specific issues that arose during that step will be pointed out with an attempt to make them as much independent of KAOS as possible. Those issues are then put into perspective in the general discussion that follows in Section 4.

3.1. Goal acquisition and object identification

The following substeps were repeatedly applied to produce the AND/OR goal structure: (1) identify goals; (2) reduce goals and identify conflicts; (3) refine informal definition of goals; (4) formalize goals and identify the objects referred to. These substeps were proceeding in parallel; synchronization between them was governed by dependency links.

3.1.1. Identifying goals

This substep initiated the elaboration process. It consists of *eliciting* a number of goals from the various sources available (preliminary documents, interviews, domain knowledge, etc.), giving them a tentative *name* and a preliminary rough informal definition, and *classifying* them in terms of the goal taxonomies defined at the meta level. Generic or analogical goals found in similar systems may be reused to assist in the elicitation process.

For example, consider the following fragment of the initial problem statement.

"*The purpose of the meeting scheduler system is to support the organization of meetings - that is, to determine, for each meeting request, a meeting date and location so that most of the intended participants will effectively participate.*"

This suggests two (interfering) goals: get every meeting request satisfied and maximize participant attendance. Hence the preliminary declarations:

SystemGoal *Achieve* [MeetingRequestSatisfied]
InstanceOf SatisfactionGoal; ...

SystemGoal *Maximize* [NumberOfParticipants]
InstanceOf SatisfactionGoal; ...

These goals are quite sketchy and imprecise at this point. Their names cover a lot of possible interpretations. The exact meaning of each goal will get clearer and clearer as reduction links are developed and formalization attempts are made. At this stage, however, the first goal declaration already expresses that under some initial condition something must eventually happen; the goal definition has the formal pattern $P \Rightarrow \Diamond Q$.

This initial formulation might suggest treating a meeting as a resource instance; accordingly some relevant generic description fragments may be retrieved from the ResourceManagement knowledge base to help in the reduction/formalization process [Dar93], [Dam93], see below.

3.1.2. Reducing goals and identifying conflicts

Goals that appear gradually are analyzed to determine whether they contribute positively or negatively to others; reduction and conflict links are captured accordingly. Bottom-up or top-down processes can be followed here. In a bottom-up process, *why* questions allow higher-level goals to be acquired from goals that contribute positively to them. In a top-down process, *how* questions allow lower-level goals to be acquired as sub-goals that contribute positively to the goal considered. Both top-down and bottom-up processes were applied in the meeting scheduler case study; the rationale for following one tactic or the other has not been made explicit.

During the reduction/conflict identification substep, meta-level knowledge about taxonomies of reduction patterns and categories may be used again to guide the acquisition process. Various types of goal reduction emerged from our case study: *agent-driven* decomposition, where a goal is reduced into sub-goals that involve less agents; *case-driven* decomposition, where a goal is reduced by case analysis -e.g., normal cases and exceptional ones; and *time-driven* decomposition, where a goal is decomposed into subgoals that need to be achieved successively over time. A simple but frequent type of time-driven decomposition corresponds to the reduction of $P \Rightarrow \Diamond Q$ into

$$P \Rightarrow \Diamond R, \quad R \Rightarrow \Diamond Q$$

This reduction pattern was used to produce

SystemGoal *Achieve* [MeetingRequestSatisfied]
ReducedTo ParticipantsConstraintsKnown,
MeetingPlanned, ParticipantsNotified; ...

together with the following new declarations:

SystemGoal *Achieve* [ParticipantsConstraintsKnown]
InstanceOf InformationGoal; ...

SystemGoal *Achieve* [MeetingPlanned]
InstanceOf SatisfactionGoal; ...

SystemGoal *Achieve* [ParticipantsNotified]
InstanceOf InformationGoal; ...

An extensive library of non-trivial reduction templates was used to support the reduction process [Dam94]. These generic reductions are proved correct using the proof theory of temporal logic [Man92]. The benefit here is that *each time one such generic reduction is reused and instantiated one gets a formal correctness proof for free*. Such proofs are important since they provide formal guarantee that a goal *will* be achieved if its subgoals are achieved.

3.1.3. Refining informal goal definitions

As a goal gets reduced further and further it becomes possible to make its preliminary informal definition more and more accurate. For example, it was only after we reached consensus on the goal tree of the *Achieve* [MeetingRequestSatisfied] goal that we were able to write the following expanded version of the original fragment:

SystemGoal *Achieve* [MeetingRequestSatisfied]
InstanceOf SatisfactionGoal
ReducedTo ParticipantsConstraintsKnown,
MeetingPlanned, ParticipantsNotified
InformalDef *Every meeting request should be satisfied within some deadline associated with the request. Satisfying a request means proposing some "best" meeting date/location to the intended participants that fit their constraints, or notifying them that no solution can be found with those constraints.*

Goal reduction and specification refinement are thus much intertwinned, as already pointed out at a lower level of development [Swa82]; the way a goal is reduced often impacts on its informal definition and vice versa.

Our experience revealed that the diversity of conceivable decompositions for some goals can be a problem. Sometimes we felt uncomfortable in deciding which goal decomposition would be the most adequate. It is only later on in the specification elaboration process that the consequence of such or such decomposition became manifest; backtracking to consider an alternative decomposition or to complete an incomplete one was then required. The latter situation arose several times during attempts to formally prove the correctness of reductions (see Section 3.1.4).

The goal reduction graph evolved thus iteratively; it was also rearranged for explanatory and work parallelization purpose. All Satisfaction goals were regrouped into one functional part whereas all Information, Robustness, Privacy and Consistency goals were regrouped into one non-functional part. The evaluation and transformation of goal structures appears to be an important issue here.

3.1.4. Formalizing goals and identifying objects

Formalization started really when the goal reduction graph became sufficiently developed and stable. Goal formalization and object identification proceeded in parallel; the formalization of a goal leads to introducing new objects each time the current vocabulary is insufficient to define the goal formally. In the early elaboration phases, the introduction of an object consisted in defining it informally; in later stages when the vocabulary was sufficiently rich it became possible to transform informal definitions into formal invariants attached to the objects.

Goal formalization was guided by two sources of knowledge: (1) the formal pattern corresponding to the classification done during goal identification (*Achieve*, *Avoid*, *Maintain*, etc.), and (2) generic formalizations of abstract goals in the domain knowledge base. The latter were retrieved using as indexing "keys" the goal pattern, goal category, and the meta-type of the objects concerned [Dar93]; the retrieved descriptions were then customized by instantiation of meta-variables and appropriate adaptations of the resulting predicates.

For example, looking at a meeting as a "one-copy" resource may prompt, from the goal description fragment

SystemGoal *Achieve* [MeetingRequestSatisfied]
InstanceOf SatisfactionGoal; **Concerns** Meeting, Initiator, ...

a search in the ResourceManagement knowledge base for a goal description that concerns the *Resource* entity, in the *Satisfaction* category and with the *Achieve* pattern. One of the descriptions found is

SystemGoal Achieve [ResourceRequestSatisfied]
InstanceOf SatisfactionGoal; **Concerns** Resource, User ...
FormalDef $(\forall$ u: User, res: Resource, rep: Repository$)$
$\text{Requesting (u, res)} \wedge \text{InScope (res, rep)}$
$\Rightarrow \Diamond \text{ Using (u, res)}$

The obvious meta-variable instantiations here are *Resource*/Meeting, *User*/Initiator, *Using*/Scheduled. The *InScope* meta-variable captures an abstraction that here corresponds to the feasibility of the various participant's constraints put altogether. The *Repository* abstraction does not seem to have much sense here. Hence we get:

SystemGoal *Achieve* [MeetingRequestSatisfied]
InstanceOf SatisfactionGoal; **Concerns** Meeting, Initiator, ...
FormalDef $(\forall$ r: Initiator, m: Meeting$)$
$\text{Requesting (r, m)} \wedge \text{Feasible (m)} \Rightarrow \Diamond \text{ Scheduled (r, m)}$

The formal definition above is to be seen as a first approximation to start with. In general the approximation needs to be refined in accordance with the informal definition, e.g., by weakening or strengthening some of the predicates, adding or deleting variables, etc. For the goal above the refined formalization was

SystemGoal *Achieve* [MeetingRequestSatisfied]
InstanceOf ... ; **Concerns ... ; ReducedTo ...** ; **InformalDef** ...;
FormalDef $(\forall$ r: Initiator, m: Meeting, p: Participant$)$
$\text{Requesting (r, m)} \wedge \text{Feasible (m)} \Rightarrow \Diamond_{\leq R(r,m)d} \text{ Scheduled (m)}$
$\wedge \text{ Invited (p, m)} \Rightarrow \Diamond_{P(p,m)d} \text{Knows (p, m)}$

where *R(r,m)* denotes the deadline (in number of days) within which the meeting request should be satisfied and *P(p,m)* denotes the deadline within which the meeting date/location should be known to the corresponding participant, with R(r,m) P(p,m). Checking the approximation against the informal definition led thus to strengthen "eventually" conditions and to strengthen the predicate by an additional requirement of keeping participants informed.

In parallel with that formalization, a preliminary definition of the *Meeting* object is elaborated accordingly:

Entity Meeting
Has When: *TimeInterval* , Where: *Location*,
Feasible, Scheduled: ***Boolean***
Invariant $(\forall$ m: Meeting$)$ $\text{Feasible (m)} \Leftrightarrow \ldots$
$\text{Scheduled (m)} \Leftrightarrow \ldots$

Precise invariants to define the *Meeting* object further seem too difficult to write at this point. Their elaboration is deferred to a later stage of the process. Similarly the *Requesting* and *Invited* objects referenced in the formal definition of the MeetingRequestSatisfied goal can be declared as relationships, e.g.,

Relationship Requesting
Links Initiator {**Role** Requests, **Card** 0:N}
Meeting {**Role** RequestedBy, **Card** 1:N}
Has DateRange: *TimeInterval*
ParticipantsList: **SetOf**[*Participant*]

The formalization of goals raised five issues which we believe are of general interest beyond the KAOS approach, namely, goal deidealization, reduction debugging, conflict resolution, change propagation, and the stage at which formalization should start. We discuss them successively.

(i) Deidealizing goals. Initial formulations tend to be too ideal, in the sense that they could never be guaranteed by the composite system -because they make overoptimistic assumptions about agent behaviour, because they are too costly to achieve, etc. For example, the subgoal ParticipantsConstraintsKnown introduced above was first defined as follows:

SystemGoal *Achieve* [ParticipantsConstraintsKnown]
InstanceOf InformationGoal
Concerns Meeting, Participant, Scheduler, ...,
FormalDef $(\forall$ m: Meeting, p: Participant, s: Scheduler$)$
$\text{Invited (p, m)} \wedge \text{Scheduling (s, m)}$
$\Rightarrow \Diamond_{\leq C(p,m)d} \text{ Knows (s, p.Constraints)}$

This goal is too ideal because it cannot be operationalized; there is no way to guarantee that every intended participant will actually communicate her date constraints in due time (e.g., some participants can be unreachable or they may not react promptly enough).

Goals must be achievable; one should not specify requirements that cannot be met. For our example one possibility is to *weaken* the goal formulation as follows:

$(\forall$ m: Meeting, p: Participant, s: Scheduler$)$
$\text{Invited (p, m)} \wedge \text{Scheduling (s, m)}$
$\Rightarrow \Diamond_{C(p,m)d}[\text{ Knows (s, p.Constraints)} \vee \neg \text{ Expected (p, m) }]$,

that is, invited participants who did not react within the prescribed deadline are removed from the list of expected participants. Alternative deidealization decisions could of course be made; they would correspond to alternative reduction nodes in the AND/OR graph. In our case the companion goal *Maximize* [NumberOfParticipants] should guarantee that as few participants as possible will eventually be ruled out because of their constraints missing.

At a higher level, it would have been wishful thinking to require the system to ensure that *all* the intended participants do effectively participate. Even the *Achieve* [MeetingRequestSatisfied] goal as formalized above is still unrealistic, because it assumes that the feasibility of the meeting requested can be assessed in the state where the request is made. The deidealized version was obtained again by weakening the formal definition obtained above for that goal into

SystemGoal *Achieve* [MeetingRequestSatisfied]
InstanceOf ... ; **Concerns ... ; ReducedTo ...** ; **InformalDef** ...;
FormalDef ($\forall$ r: Initiator, m: Meeting, p: Participant)
Requesting (r, m) $\Rightarrow \Diamond_{\leq R(r,m)d}$ [Scheduled (m) $\vee \neg$ Feasible (m)]
$\wedge$ Invited (p, m) $\Rightarrow \Diamond_{P(p,m)d}$ Knows (p, m)

(ii) Debugging goal reductions formally. It was frequently the case that intuitive goal reductions were found to be incorrect in the later stage of trying to prove the correctness of reductions formally. Incomplete reduction was the most frequent error. For example, the goal *Achieve* [MeetingRequestSatisfied] was seen in step 3.1.2 above to be initially refined into three subgoals that were formalized as follows:

SystemGoal *Achieve* [ParticipantsConstraintsKnown]
FormalDef ($\forall$ m: Meeting, p: Participant, s: Scheduler)
Invited (p, m) $\wedge$ Scheduling (s, m)
$\Rightarrow \Diamond_{C(p,m)d}$ Knows (s, p.Constraints)

SystemGoal *Achieve* [MeetingPlanned]
FormalDef ($\forall$ r: Initiator, m: Meeting, s: Scheduler)
Requesting (r, m) $\wedge$ Scheduling (s, m)
$\Rightarrow \Diamond_{R(r,m)d}$[Scheduled (m) $\vee$ DeadEnd (m)]

SystemGoal *Achieve* [ParticipantsNotified]
FormalDef ($\forall$ m: Meeting, p: Participant)
Invited (p, m) $\Rightarrow \Diamond_{P(p,m)d}$Knows (p, m)

It was only when trying to prove formally that those three assertions imply the formal definition of the goal *Achieve* [MeetingRequestSatisfied] hereabove that we found a subgoal missing for the proof to be carried out. This led to the revised reduction

SystemGoal *Achieve* [MeetingRequestSatisfied]
ReducedTo *SchedulerAvailable*,
ParticipantsConstraintsKnown,
MeetingPlanned, ParticipantsNotified

where the missing sub-goal to allow the proof is defined as follows:

SystemGoal *Achieve* [SchedulerAvailable]
InstanceOf SatisfactionGoal
Concerns Meeting, Initiator, Scheduler
FormalDef ($\forall$ r: Initiator, m: Meeting)
Requesting (r, m)
$\Rightarrow \Diamond_{Sd}$ ($\exists$ s: Scheduler) Scheduling (s, m)

(iii) Handling conflicting goals. Finding clever, smooth ways of resolving conflicts is among the essential tasks of any meeting scheduler. For example, goals like *Achieve* [MeetingRequestSatisfied] and *Achieve* [MeetingReplannedOnRequest] can be seen to be conflicting as dynamic replanning can interfere with the normal planning of other meetings; even multiple normal planning instances can be conflicting with each other. As another typical example of interfering goals, the InformationGoal *Maintain* [ParticipantAgendaKnown] contributes positively to the goal *Achieve* [ParticipantsConstraintsKnown]; at the same time it may contribute negatively to the PrivacyGoal *Maintain* [ParticipantPrivacy]. We experienced that the elaboration of adequate conflict resolution policies may in itself represent a non-trivial, *problem-dependent* task in the elicitation process [Vla93]. Some preliminary work on goal conflict detection and resolution suggests that temporal logic formalization again is much helpful in identifying various patterns of conflicts, determining their weakest cause and defining corresponding families of resolution tactics.

(iv) Propagating changes. Every change in a goal definition has to be propagated downwards and upwards along *Reduction* links and also along *Operationalization*, *Conflict* and *Concerns* links. As in any complex document this turns out to be rather awkward in some cases. The KAOS language still lacks modular constructs to support *conceptual packages* with low interaction and well-defined interfaces. During our case study it was decided to split the goal graph into two parts: satisfaction goals in one part and and all other goals in the other. This allowed us to work in parallel without too much interaction, under the constraint that common "interface" objects could only be changed if everyone agreed on the necessity of the change. Language constructs to support such natural ways of proceeding are definitely needed.

(v) When should goal formalization start? Top-level or qualitative goals were in general non-formalizable because they were interpretable in many different ways. It is only when their reduction (and sometimes their operationalization) was made sufficiently clear and precise that formalization could start. A typical example of this is the very high-level goal Maintain[UsabilityByNonExperts]. There were other situations where a goal formulation could not be made formal because it was deliberately intended to capture some degree of freedom left to the appreciation of one agent or another. For example, the following goal could not be formalized whereas one alternative set of operationalizations could:

SystemGoal Maximize [ScheduleConvenience]
InstanceOf SatisfactionGoal
Concerns Meeting, Initiator, Participant ...
InformalDef *When the scheduler finds multiple solutions to a meeting request, she/it should select some "best" one according to contextual and situation-dependent criteria which are left at the initiator's appreciation.*
OperationalizedTo NoParticipantOverload, OneScheduleKept
or OperationalizedTo ...

Satisfaction, information and consistency goals often were formalizable rather early in the process.

To end up this Section, it should be stressed that the goal structure elaborated in this first step of the strategy did not cover all the goals found in the final document. Discovering implicit goals is a non-trivial reverse engineering task. In our case we departed from a strict application of the strategy from time to time and used *scenarios* to validate the goal structure and find out new goals and constraints (see Section 3.5).

3.2. Identifying agents/actions from goals and operationalizing goals

During our meeting scheduler case study, the last stages of goal elaboration were performed in parallel with the preliminary stages of agent/action identification and goal operationalization. Goals need to be converted into constraints, that is, operational versions that could be enforced by agents available in the composite system. Operationalization thus requires some knowledge about which human or automated agents might be at hand and what their capabilities are. Given a goal and a set of possible agents the analyst must then determine whether the goal could be enforced by one of them through appropriate control over state transitions. If this is the case the goal becomes an assignable constraint; otherwise it must be reduced further.

To illustrate how goal reduction, agent identification and goal operationalization may be intertwinned, consider again the *MeetingRequestSatisfied* goal and its four subgoals formalized in Section 3.1.4.

Agent/action identification. Among all the objects referenced in those formal definitions, the following ones can be classified as being instances of the *Agent* meta-type:

– *Initiator* objects could control state transitions of the *Requesting*, *Invited*, and *Knows[p,m]* relationships (referenced in the *MeetingRequestSatisfied* goal) through applications of some actions; one may thus write preliminary definition fragments such as

Agent Initiator
  CapableOf Request, Invite, Notify, ... ; ...

Action Request
  Input Initiator {**Arg**: r}, Meeting {**Arg**: m}; **Output** Requesting
  Pre ¬ Requesting (r, m); **Post** Requesting (r, m)

– *Scheduler* objects could control state transitions of *Meeting* entities (referenced in the *MeetingPlanned* goal) and of *Knows[p,m]* relationships through applications of some actions; hence a preliminary sketch such as

Agent Scheduler
  CapableOf DetermineSchedule, Notify, ... ; ...

Action DetermineSchedule
  Input Requesting, Meeting {**Arg**: m}; **Output** Meeting {**Res**: m}
  Pre Requesting (-,m) $\wedge \neg$ Scheduled (m)
  Post Feasible (m) $\Rightarrow$ Scheduled (m)
    $\wedge \neg$ Feasible (m) $\Rightarrow$ DeadEnd (m)

– *Participant* objects could control state transitions of *Knows* meta-relationship instances that link *Scheduler* and *Participant* objects in the expression of the *ParticipantsConstraintsKnown* goal; hence the preliminary definition fragments

Agent Participant
  CapableOf CommunicateConstraints, ... ; ...
  Has Constraints: **Tuple** [ExcludedDates: **SeqOf** [*TimeInterval*],
    PreferredDates: **SeqOf** [*TimeInterval*]]
  Invariant ($\forall$p: Participant, m: Meeting)
    Invited (p, m) $\Leftrightarrow$ p $\in$ Requesting[-,m]**.**ParticipantsList

Note that the actions preliminarily defined are given *elementary* pre- and post-conditions; the latter only capture the corresponding state transitions identified. Those conditions will be refined and strengthened in the later steps of the strategy to make them ensure that the operational constraints will be met.

Goal operationalization. The goal *Achieve* [ParticipantsConstraintsKnown] above is operationalizable into two constraints: *Achieve* [ParticipantsConstraintsRequested] and *Achieve* [ParticipantsConstraintsProvided]. The former is assignable either to the *Scheduler* or to the *Initiator* agent (actual responsibility assignments are made at the last step of the strategy); the latter is assignable to the corresponding *Participant* agent. This is just *one* operationalization alternative; another alternative would correspond to the use of an agenda made directly accessible to the scheduler, see below. The goal *Achieve* [ParticipantsNotified] is easily operationalized into a *NotificationSent* constraint assignable to the *Scheduler* or the *Initiator* agent. The goal *Achieve* [MeetingPlanned] can be OR-operationalized in two alternative ways: the first alternative yields the *Achieve* [MeetingPlannedWithoutNegotiation] constraint assignable to the *Scheduler* agent. The second alternative yields the *Achieve* [MeetingPlannedWithNegotiation] constraint assignable to the *Initiator* or *Scheduler* agents. The latter alternative will eventually be preferred as it contributes positively to the goal *Maximize* [NumberOfParticipants] whereas the former contributes negatively to the goal *Minimize* [DeadEnds]. (Note that *Achieve* [MeetingPlannedWithNegotiation] became a constraint assignable to, e.g., the *Scheduler* agent because we saw no way to split the responsibility for that objective among the *Scheduler* and *Participant* agents -a participant should by no means be made responsible for the system's wrong behaviour in case she fails to respond to scheduler requests for weakening her constraints.)

Goal operationalization often requires a *change in level of abstraction*. Abstract concepts involved in goal formulations need to be mapped to concrete ones to make it possible to formulate constraints/actions assignable to agents. This turned out to be hard in some cases. It somewhat corresponds to the choice of good representation functions to map abstract objects to concrete variables [Hoa72], in a way similar to data reification [Jon90]. To illustrate this, consider the qualitative goal

SystemGoal *Minimize* [participant-schedulerInteraction]
  ReducedTo ParticipantAgendaKnown, FinalApprovalAsked; ...

The reduction introduces an abstract *Agenda* concept in one of the subgoals:

SystemGoal *Maintain* [ParticipantAgendaKnown]
  InstanceOf InformationGoal; **Concerns** Agenda, ...;
  FormalDef ($\forall$ m: Meeting, p: Participant, s: Scheduler)
    Invited (p, m) $\wedge$ Scheduling (s, m) $\Rightarrow$ Knows (s, Agenda[p,-])
  ReducedTo AgendaAccessible, AgendaUpToDate; ...

The new subobjective *Maintain* [AgendaUpToDate] is clearly assignable as an operational constraint to *Participant* agents. Formalizing that constraint in terms of states controllable by Participant agents requires the *Agenda* abstract concept to be represented in more concrete terms, e.g.,

Relationship Agenda
Links Participant {**Role** KeepsAppointmentIn, **Card** 1:N}
  Calendar {**Role** TracksAppointmentFor, **Card** 1:N}
Has BusyPeriods: **SeqOf** [*TimeInterval*]
  FreePeriods: **SeqOf** [*TimeInterval*]
  CoveredPeriods: **SeqOf** [*TimeInterval*]
Invariant

BusyPeriods $\cap^*$ FreePeriods = [ ]
BusyPeriods $\cup^*$ FreePeriods = CoveredPeriods

Thanks to this representation the *Maintain* [AgendaUpToDate] constraint can now be formalized as follows:

WeakConstraint *Maintain* [AgendaUpToDate]
InstanceOf ConsistencyConstraint
UnderResponsibilityOf Participant
FormalDef $(\forall$ p: Participant, tp: TimeInterval$)$
$\neg$ Free (p, tp) $\Leftrightarrow$ tp $\in$ BusyPeriods

3.3. Object/action refinement and strengthening

The formulation of finer subgoals and constraints may involve new objects/actions to be defined correspondingly (see, e.g., the *Agenda* object discussed above). Such formulations may also require specifications of already identified objects/actions to be refined and/or revised. For example, the following constraint was introduced to "implement" the three goals specified in its *Operationalizes* clause:

StrongConstraint *Achieve* [BestSchedule]
Operationalizes *Achieve* [MeetingPlannedWithNegotiation], *Maximize* [ScheduleConvenience], *Minimize* [DeadEnds]
FormalDef $(\forall$ r: Initiator, m: Meeting, s: Scheduler$)$
Requesting (r, m) $\wedge$ Scheduling (s, m)
$\Rightarrow \Diamond_{R(r,m)d}$[Feasible (m) $\Rightarrow$ Scheduled (m) $\wedge$ Preferred (m)
$\wedge$ NearlyFeasible (m)
$\Rightarrow$ ScheduledByNegotiation (m)
$\wedge \neg$ Feasible (m) $\wedge \neg$ NearlyFeasible (m)
$\Rightarrow$ DeadEnd (m)]

The formulation of that constraint called for a refinement of the *Meeting* object introduced in Section 3.1.4, such as

Entity Meeting
Has When: *TimeInterval,* Where: *Location,*
Feasible, *NearlyFeasible*, Scheduled, *Preferred*: ***Boolean***
Invariant $(\forall$ m: Meeting$)$
Feasible (m) $\Leftrightarrow$ $\exists$ d: Plannable (m, d)
Plannable (m, d) $\Leftrightarrow$ d **in** Requesting[-,m]**.**DateRange
$\wedge\ \forall$p $\in$ Requesting[-,m]**.**ParticipantsList:
d $\notin$ ExcludedDates (p**.**Constraints)
Scheduled (m) $\Leftrightarrow$ m.When = **OneOf** {d | Plannable (m, d)}
Preferred (m)) $\Leftrightarrow$ ... , NearlyFeasible (m) $\Leftrightarrow$ ...

Besides it is often the case that some actions and objects need to be *strengthened* in order to ensure that every constraint will be met in the composite system. This may require strengthening the elementary pre- and postconditions, introducing trigger conditions, or introducing new specific actions for constraint satisfaction. A number of inference rules are proposed in [Dar93] for the formal derivation of such strengthenings. The general principle for a constraint *C* and action *A* is to match *A*'s initial/final conditions against *C* to factor out subsidiary conditions on initial/final states for *A* to ensure *C*.

Using such rules the *DetermineSchedule* action identified in Section 3.2 was strengthened as follows to ensure the *BestSchedule* constraint specified hereabove:

Action DetermineSchedule
Input Requesting, Meeting {**Arg**: m}; **Outpu**t Meeting {**Res**: m}
Ensures BestSchedule
Pre Requesting (-,m) $\wedge \neg$ Scheduled (m)
StrengthenedPost
Feasible (m) $\Rightarrow$ Scheduled (m) $\wedge$ Preferred (m)
$\wedge$ NearlyFeasible (m) $\Rightarrow$ ScheduledByNegotiation (m)
$\wedge \neg$ Feasible (m) $\wedge \neg$ NearlyFeasible (m) $\Rightarrow$ DeadEnd (m)

3.4. Assigning responsibilities to agents

The *identification* of possible responsibility links during operationalization was in general fairly easy, because the number of potential agents was small and their capabilities were obvious and not overlapping too much (see the examples suggested in Section 3.2). For a set of candidate agents assignable to a constraint, the *decision* of actual assignment of one of them to the actions ensuring that constraint was sometimes less obvious. Among the attributes of responsibility links provided by the KAOS meta-model, *Reliability* was the most frequently used criterion to select effective assignments. (The *Cost* attribute was never used due to the lack of appropriate cost evaluation models.) Backtracking on operationalization choices was sometimes required to select a more reliable operationalization and agent assignment. For example, the operationalization of the *ParticipantsConstraintsKnown* goal through the *ParticipantsConstraintsRequested* and *ParticipantsConstraintsProvided* constraints was eventually selected over the alternative operationalization through the use of an electronic agenda. The reason was that the *AgendaUpToDate* constraint found in the latter alternative could only be assigned to *Participant* agents but with lower reliability than the assignments in the former alternative where *ParticipantsConstraintsRequested* and *ParticipantsConstraintsProvided* are assigned to the *Scheduler* and *Participant* agents, respectively.

The KAOS approach was felt to be limited in the assignment decision step; like other approaches it provides little support for formal reasoning about *alternative* assignments -an issue that we feel of upmost importance in the requirements engineering process, especially in new application areas where the decision about what should be automated and what should not is less obvious than usual.

3.5. Using scenarios for validation

The use of operational scenarios proved to be effective in getting an overall picture of the composite system during its elaboration and also in identifying missing actions together with their underlying (implicit) goals. Usually an action was found to be missing from some typical scenario being built; the underlying constraints/goals were then made explicit by answering *why* questions about that missing action. Consider, for example, the following scenario.

Scenario HandleMeetingRequest
Is **(***IssueRequest:* SubmitRequest; ValidateRequest**);**
AskParticipantsConstraints**;**
(*GetConstraints:*: FormulateConstraints;
CommunicateConstraints;
ValidateConstraints **)***;
PlanMeeting;
(*NotifyResults:* **(**NotifyDate&Location | NotifyDeadEnd**))**

This scenario revealed that the *ValidateRequest* and *ValidateConstraints* actions had been overlooked in the elaboration process. A *why* question about the latter action led to identify a new constraint *AcceptableParticipantsConstraints* that completes the AND-list of constraints operationalizing the *ParticipantsConstraintsKnown* goal (and positively contributes to the goal *Avoid [PhysicalLawBroken].*)

4. Discussion

The Meeting Scheduler was felt to be a rich and challenging exemplar for Requirements Engineering research. Compared with the traditional examples used in the literature it offers a rather unusual combination of features found in real requirements documents. It covers many different types of requirements expressed with varying levels of precision -beside functional requirements there are real-time performance constraints, privacy concerns, concurrency and distribution aspects, optimization requirements (such as maximizing meeting convenience or minimizing interaction among participants), suggested heuristics for defining near solutions, very high-level objectives such as transparency, reliability, flexibility, usability and changeability, etc. It also includes many open issues that need to be resolved as part of the requirements engineering process -e.g., interfering objectives, automation alternatives, alternative patterns of communication and cooperation, etc. The satisfactory behaviour of the automated part of the composite system heavily relies on the satisfactory behaviour of agents in the environment part. The precise functionalities the system should provide are not quite clear from the beginning. While still being of very small size with respect to "real world" documents, the preliminary problem statement is considerably larger than the usual toy problems that have been popular in the past. At the same time it lends itself to implementations that are potentially useful in real-life situations.

For this problem the ability to address *why*, *what*, *who*, *how* questions within separate but linked specification units was felt to be a major advantage of goal- and agent-oriented languages over functional and "conceptually flat" representation languages such as Z or VDM. Moreover in KAOS the outer ERA-like declaration level allows for traceability between such units; the inner temporal logic level allows for formal reasoning during their elaboration. Applying a goal-directed strategy allowed us to *trace* decisions regarding the definition of objects and actions back to their underlying goals.

We found the distinction between domain definitions and system requirements [Jac93] very helpful in our case study. This important separation of concerns is supported in KAOS by the partitioning of assertions -the elementary invariants and pre- and postconditions describe objects and actions in the domain whereas the requirements themselves consist of *goals*, operationalizing *constraints*, and *strengthenings* of the domain assertions to ensure such constraints. Sometimes we concluded that some assertion fragment was not proper to the domain but rather a specific requirement; this led us to to ask *why* questions and discover new constraints and goals therefrom.

The usefulness of meta-level constraints to guide the acquisition process at specific points was confirmed. For instance, when it was realized that the *ParticipantsConstraintsProvided* constraint was in fact a *weak* constraint the question was triggered as to what the associated restoration action should be in case of temporary violation; a missing action *SendReminderForConstraints* was thereby identified together with its trigger condition derived from the formal definition of the *ParticipantsConstraintKnown* constraint. As suggested in Section 3.1.4, the benefits of sometimes reusing and adapting abstract descriptions instead of starting from scratch became apparent too.

Our specification did not cover all aspects described in the preliminary definition [Vla93]. One reason is that the aspects we covered in all details already resulted in a very long document. Writing all details was time consuming and tedious. (The unavailability of a syntax-directed editor was especially harmful in that respect.) Another reason is that there are a number of aspects from the initial problem statement that we could not capture -notably, usability and changeability. Yet another reason is that our initial version of the specification turned out to have a significant number of bugs -many of them being detected while trying to make formal proofs (see the example in Section 3.1.4). Therefore we shifted our focus towards formal techniques for avoiding such problems -in particular, techniques for proving the correctness of goal reductions and operationalizations. We started some work in that direction by building a rather extensive library of reduction templates that are proved correct using standard temporal logic techniques.

In Section 3 we discussed various KAOS-independent problems encountered at each step of the elaboration process, notably, the difficulty of identifying the right goals and reductions, the need to sometimes deidealize goals by weakening their formulation or finding alternative reductions, the problem of dealing with conflicting goals and finding reasonable compromises, and the need to occasionally restructure the AND/OR goal graph so as to permit parallel elaboration on disjoint types of goals.

There are other issues we did not discuss there. The granularity of conceptual units is one of them. The choice of granularity of a "leaf" concept impacts on the granularity of the concepts depending on it. For example, if the *BestSchedule* constraint introduced in Section 3.3 had been split into a number of sub-constraints, the agents and actions to enforce it would have needed to be decomposed. Similarly, if the *Scheduler* agent had been split into a number of sub-agents, the actions assigned to it would have needed to be decomposed further. Currently we do not have precise criteria for deciding which level of granularity seems the best.

Sometimes the need was felt to formulate some assertions as *assumptions* that could be retracted in some specific versions of the system. In particular, it is very often the case that goals/constraints are actually achievable only under some optimistic assumptions about normal behaviour of agents. For example, one might record the fact that participants are highly reliable in maintaining their electronic agenda as an explicit assumption. (Similar assumptions were made about the participants willingness to respond to scheduler requests for communicating date constraints or weakened versions of them.) The formulation of such assertions as *explicit* assumptions attached to the goals/constraints that depend on them would make it possible to retract them in some specific versions of the system; all depending specification units could then easily be pointed out for appropriate revision. To support this sort of truth maintenance the KAOS meta-model would need to be extended with an *Assumption* meta-concept and *Dependency* meta-relationship. We are currently working further on this as assumptions/dependencies are frequently encountered when trying to use our meta-model as a *process meta-model* for modelling software processes.

As mentioned in Section 3.5, we had frequent recourse to *usage scenarios* to validate our specification and try to point out aspects of the system that we had overlooked. We experienced that typical scenarios are sometimes much easier to identify in the preliminary stages of acquisition than some abstract goals that can be made explicit only after some deeper understanding of the system has been gained. Scenarios may be a source of complementary insights and are a popular acquisition vehicle [Jac92], [Pot94]. They can also be generated from the specification to exhibit deficiencies in it [Fic92]. Besides, *viewpoints* capture and integration [Nus93] may provide another complementary means for acquiring goals, objects and actions. Viewpoints seem especially meaningful in applications such as meeting scheduling where the inderdependency between human and automated agents is high. We are thus more and more led to the conclusion that a *hybrid strategy* that would combine goal-directed, scenario-directed and viewpoint-directed acquisition tactics would contribute to building a more adequate, complete and consistent set of requirements.

Acknowledgement. The Meeting Scheduler problem originated from a number of lively discussions with Steve Fickas' group at the University of Oregon. The work reported herein was partially funded by the Commission of the European Communities (ESPRIT Project 2537 "Icarus") and the Belgian Ministery of Scientific Affairs (SPPS Project IT/IF/10 "Oasis").

References

[Dam93] R. Darimont, "Resource Models in KAOS", Res. Rep. RR-93-23, UCL Unité d'Informatique, 1993.

[Dam94] R. Darimont, "A Taxonomy of Reduction Patterns", Res. Rep. RR-94-1, UCL Unité d'Informatique, 1994.

[Dar91] A. Dardenne, S. Fickas S. and A. van Lamsweerde, "Goal-directed Concept Acquisition in Requirements Elicitation", *Proc. IWSSD-6 - 6th Intl.Worksh. on Software Specification and Design*, IEEE, 1991, 14-21.

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

[Dub93] E. Dubois, Ph. Du Bois and M. Petit, "Object-Oriented Requirements Analysis: An Agent Perspective", *Proc. ECOOP'93 - 7th European Conf. on Object-Oriented Programming*, Springer-Verlag LNCS 707, 1993, 458-481.

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

[Fea93] M. Feather, "Requirements Reconnoitering at the Juncture of Domain and Instance", *Proc. RE'93 - 1st Intl. IEEE Symp. on Requirements Engineering*, Jan. 1993, 73-77.

[Fic92] S. Fickas and R. Helm, "Knowledge Representation and Reasoning in the Design of Composite Systems", *IEEE Trans. on Software Engineering*, June 1992, 470-482.

[Fin87] A. Finkelstein and C. Potts, "Building Formal Specifications Using "Structured Common Sense", *Proc. IWSSD-4 - 4th Intl. Workshop on Software Specification and Des*ign, IEEE, 1987, 108-113.

[Gut93] J.V. Guttag and J.J. Horning, *LARCH: Languages and Tools for Formal Specification*, Springer-Verlag, 1993.

[Hoa72] C.A.R. Hoare, "Proof of Correctness of Data Representations", *Acta Informatica* Vol. 1, 1972, 271-281.

[Jac92] I. Jacobson, *Object-Oriented Software Engineering - A Use Case Driven Approach*, Addison-Wesley, 1992.

[Jac93] M. Jackson and P. Zave, "Domain Descriptions", *Proc. RE'93 - 1st Intl. IEEE Symp. on Requirements Engineering*, Jan. 1993, 56-64.

[Jon90] C.B. Jones, *Systematic Software Development Using VDM*, 2nd edition, Prentice Hall, 1990.

[Ken93] S. Kent, T. Maibaum and W. Quirk, "Formally Specifying Temporal Constraints and Error Recovery", *Proc. RE'93 - 1st Intl. IEEE Symp. on Requirements Engineering*, Jan. 1993, 208-215.

[Koy92] R. Koymans, *Specifying Message Passing and Time-Critical Systems with Temporal Logic*, Springer-Verlag LNCS 651, 1992.

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

[Myl92] J. Mylopoulos, L. Chung and B. Nixon, "Representing and Using Non-Functional Requirements: A Process-Oriented Approach", *IEEE Trans. on Software Engineering*, June 1992, 483-497.

[Nus93] B. Nuseibeh, J. Kramer and A. Finkelstein, "Expressing the Relationships Between Multiple Views in Requirements Specification", *Proc. ICSE-15 - 15th Intl. Conf. on Software Engineering*, 1993, 187-197.

[Pot94] C. Potts, K. Takahashi and A.I. Anton, "Inquiry-Based Requirements Analysis", *IEEE Software*, March 1994, 21-32.

[Reu91] H.B. Reubenstein and R.C. Waters, "The Requirements Apprentice: Automated Assistance for Requirements Acquisition", *IEEE Trans. on Software Engineering* 17(3), March 1991, 226-240.

[Rob89] W.N. Robinson, "Integrating Multiple Specifications Using Domain Goals", *Proc. IWSSD-5 - 5th Intl. Workshop on Software Specification and Design*, IEEE, 1989, 219-226.

[Spi92] J.M. Spivey, *The Z Notation - A Reference Manual*, 2nd edition, Prentice Hall, 1992.

[Swa82] W. Swartout and R. Balzer, "On the Inevitable Intertwining of Specification and Implementation", *Communications of the ACM*, Vol. 25 No. 7, July 1982, 438-440.

[Vla91a] A. van Lamsweerde, "Learning Machine Learning", in: *Introducing a Logic Based Approach to Artificial Intelligence* - Vol. 3, Wiley, 1991, 263-356.

[Vla91b] A. van Lamsweerde, A. Dardenne, B. Delcourt and F. Dubisy, "The KAOS Project: Knowledge Acquisition in Automated Specification of Software", *Proc. AAAI Spring Symp. Series*, Track: "Design of Composite Systems", Stanford University, March 1991, 59-62.

[Vla93] A. van Lamsweerde, R. Darimont and Ph. Massonet,

"The Meeting Scheduler System - Preliminary Definition",
Internal Report, University of Louvain, 1993.