Managing Conflicts in Goal-Driven Requirements Engineering

Axel van Lamsweerde, *Member*, *IEEE*, Robert Darimont, *Member*, *IEEE*, and Emmanuel Letier

Abstract—A wide range of inconsistencies can arise during requirements engineering as goals and requirements are elicited from multiple stakeholders. Resolving such inconsistencies sooner or later in the process is a necessary condition for successful development of the software implementing those requirements. The paper first reviews the main types of inconsistency that can arise during requirements elaboration, defining them in an integrated framework and exploring their interrelationships. It then concentrates on the specific case of conflicting formulations of goals and requirements among different stakeholder viewpoints or within a single viewpoint. A frequent, weaker form of conflict called divergence is introduced and studied in depth. Formal techniques and heuristics are proposed for detecting conflicts and divergences from specifications of goals/ requirements and of domain properties. Various techniques are then discussed for resolving conflicts and divergences systematically by introduction of new goals or by transformation of specifications of goals/objects toward conflict-free versions. Numerous examples are given throughout the paper to illustrate the practical relevance of the concepts and techniques presented. The latter are discussed in the framework of the KAOS methodology for goal-driven requirements engineering.

Index Terms—Goal-driven requirements engineering, divergent requirements, conflict management, viewpoints, specification transformation, lightweight formal methods.

1 INTRODUCTION

REQUIREMENTS engineering (RE) is concerned with the elicitation of high-level goals to be achieved by the envisioned system, the refinement of such goals and their operationalization into specifications of services and constraints, and the assignment of responsibilities for the resulting requirements to agents such as humans, devices, and software.

Goals play a prominent role in the RE process; they drive the elaboration of requirements to support them [49], [8], [50]; they provide a completeness criterion for the requirements specification—the specification is complete if all stated goals are met by the specification [55]; they are generally more stable than the requirements to achieve them [3]; they provide a rationale for requirements—a requirement exists because of some underlying goal which provides a base for it [8], [52]. In short, requirements "implement" goals much the same way as programs implement design specifications.

The elicitation of goals, their organization into a coherent structure, and their operationalization into requirements to be assigned to the various agents is a critical part of requirements engineering. One significant problem requirements engineers have to cope with is the management of various kinds of inconsistency resulting from the acquisition, specification, and evolution of goals/requirements from multiple sources [51]. Such inconsistencies may be desirable, for instance, to allow further elicitation of information that would have been missed otherwise [23]. However, their resolution at some stage or another of the process is a necessary condition for successful development of the software implementing the requirements.

Various approaches have been proposed to tackle the inconsistency problem in requirements engineering.

Robinson has convincingly argued that many inconsistencies originate from conflicting goals; inconsistency management should, therefore, proceed at the goal level [46]. Reasoning about potential inconsistencies requires techniques for representing overlapping descriptions and inconsistency relationships. Beside goal refinement links, binary conflict links have been introduced in goal structures to capture situations where the satisfaction [8] or satisficing [46], [39] of one goal may preclude the satisfaction/satisficing of another. Mechanisms have also been proposed for recording independent descriptions into modular structures called viewpoints; such structures are linked by consistency rules and associated with specific stakeholders involved in the elicitation process [13], [42].

Various inconsistency management techniques have been worked out using such representations. Much work has been done on the *qualitative reasoning* side. For example, the labeling procedure described in [39] can be used to determine the degree to which a goal is satisficed/denied by lower-level requirements; this is achieved by propagating such information along positive/negative support links in the goal graph. Robinson [46] suggests a procedure for identifying conflicts at requirements level and characterizing them as differences at goal level; such differences are

• *A. van Lamsweerde and E. Letier are with the Département d'Ingénierie Informatique, Université Catholique de Louvain, Place Sainte Barbe 2, B-1348 Louvain-la-Neuve, Belgium. E-mail: {avl, eleteier}@info.ucl.ac.be.*
• *R. Darimont is with CEDITI-UCL, Avenue Jean Mermoz 30, B-6041 Charleroi, Belgium. E-mail: rd@cediti.be.*

Manuscript received 15 Sept. 1997; revised 13 Mar. 1998.
Recommended for acceptance by C. Ghezzi and B. Nuseibeh.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number 107210.

resolved (e.g., through negotiation [47]) and then down propagated to the requirements level. In the same vein, [4] proposes an iterative process model in which:

1) all stakeholders involved are identified together with their goals (called win conditions);
2) conflicts between these goals are captured together with their associated risks and uncertainties;
3) goals are reconciled through negotiation to reach a mutually agreed set of goals, constraints, and alternatives for the next iteration.

Some work has also been done more recently on the *formal reasoning* side. Spanoudakis and Finkelstein [53] propose a framework for defining and detecting conceptual overlapping as a prerequisite to inconsistency. Hunter and Nuseibeh [23] discuss the benefits of reasoning in spite of inconsistency and suggest a paraconsistent variant of first-order logic to support this.

The various inconsistency management techniques above refer to inconsistencies among goals or requirements. Other forms of inconsistencies have been explored like deviations of viewpoint-based specifications from process-level rules [42], or deviations of the running system from its specifications [15]; the latter may result from evolving assumptions [16] or from unanticipated obstacles that obstruct overideal goals, requirements or assumptions [45], [32]. (The term "deviation" is used here for a state transition leading to inconsistent states [7].)

The current state of the art in inconsistency management for requirements engineering suffers from several problems.

- The specific kind of inconsistency being considered is not always clear. In fact, there is no common agreement on what a conflict between requirements does really mean. The lack of precise definition is a frequent source of confusion. Moreover, most techniques consider binary conflicts only, that is, conflicts among pairs of requirements. As we will see, there may be conflicts among three requirements, say, that are nonconflicting pairwise.
- There is no systematic support for detecting conflicts among nonoperational specifications of goals or requirements—if one excepts theorem proving techniques for logically inconsistent specifications. Current conflict management techniques take it for granted that conflicts have been identified in some way or another.
- There is a lack of systematic techniques for resolving inconsistencies through goal/requirement transformations—one notable exception is [48], which proposes a set of operators for restructuring objects involved in conflicting goals.

The purpose of this paper is to tackle those three problems in a formal reasoning setting by:

1) Reviewing various types of inconsistency frequently encountered in requirements engineering, defining them in a common framework and studying their relationships;
2) Proposing formal techniques and heuristics for identifying n-ary conflicts from specifications of goals/requirements and from known properties about the domain; and
3) Presenting formal techniques and heuristics for conflict resolution by specification transformation.

Special emphasis is put on a weak form of conflict which has received no attention so far in the literature although frequently encountered in practice. Roughly speaking, a *divergence* between goals or requirements corresponds to situations where some particular combination of circumstances can be found that makes the goals/requirements conflicting (that is, logically inconsistent). Such a particular combination of circumstances will be called a *boundary condition*.

To give an example from a real situation, consider the electronic reviewing process for a scientific journal, with the following two security goals:

1) maintain reviewers' anonymity;
2) achieve review integrity.

One can show that these goals are *not* logically inconsistent. A boundary condition to *make* them logically inconsistent would arise from a French reviewer notably known as being the only French expert in the area of the paper and who makes typical French errors of English usage. One way to resolve the divergence is to prevent this boundary condition from occurring (e.g., not asking a French reviewer if she is the only French expert in the domain of the paper); another way would be to weaken the divergent assertions (e.g., weakening the integrity requirement to allow correcting typical errors of English usage).

A key principle is to manage conflicts *at the goal level* so that more freedom is left to find adequate ways to handle conflicts—like, e.g., alternative goal refinements and operationalizations which may result in different system proposals.

The integration of conflict management into the RE process is detailed in the paper in the context of the KAOS requirements engineering methodology [9], [30], [10], [32], [33]. KAOS provides a multiparadigm specification language and a goal-directed elaboration method. The *language* combines semantic nets [5] for the conceptual modeling of goals, requirements, assumptions, agents, objects, and operations in the system; temporal logic [34], [28] for the specification of goals, requirements, assumptions, and objects; and state-based specifications [43] for the specification of operations. Unlike most specification languages, KAOS supports a strict separation of requirements from domain descriptions [24]. The *method* roughly consists of:

1) eliciting and refining goals progressively until goals assignable to individual agents are obtained,
2) identifying objects and operations progressively from goals,
3) deriving requirements on the objects/operations to meet the goals, and
4) assigning the requirements and operations to the agents.

An environment supporting the KAOS methodology is now available and has been used in various large-scale, industrial projects [11].

The rest of the paper is organized as follows: Section 2 provides some background material on the KAOS methodology that will be used in the sequel. Section 3 reviews various kinds of inconsistency that can arise in requirements engineering, introduces some notational support for making different viewpoints explicit, and defines the concepts of conflict and divergence precisely. Sections 4 and 5 then discuss techniques for conflict/divergence detection and resolution, respectively.

2 GOAL-DRIVEN RE WITH KAOS

The KAOS methodology is aimed at supporting the whole process of requirements elaboration—from the high-level goals to be achieved to the requirements, objects, and operations to be assigned to the various agents in the composite system. (The term "composite system" is used to stress that the system is not only comprised of the software but also its environment [14].) Thus, WHY, WHO, and WHEN questions are addressed, in addition to the usual WHAT questions addressed by standard specification techniques.

The methodology is comprised of a specification language, an elaboration method, and meta-level knowledge used for local guidance during method enactment. Hereafter, we introduce some of the features that will be used later in the paper; see [9], [30], [10] for details.

2.1 The KAOS Language

The specification language provides constructs for capturing various types of concepts that appear during requirements elaboration.

2.1.1 The Underlying Ontology

The following types of concepts will be used in the sequel:

- *Object:* An object is a thing of interest in the composite system whose instances may evolve from state to state. It is, in general, specified in a more specialized way—as an *entity, relationship,* or *event* dependent on whether the object is an autonomous, subordinate, or instantaneous object, respectively. Objects are characterized by attributes and invariant assertions. Inheritance is, of course, supported.
- *Operation:* An operation is an input-output relation over objects; operation applications define state transitions. Operations are characterized by pre-, post-, and trigger conditions. A distinction is made between *domain* pre/postconditions, which capture the elementary state transitions defined by operation applications in the domain, and *required* pre/postconditions, which capture additional strengthenings to ensure that the goals are met.
- *Agent:* An agent is another kind of object which acts as processor for some operations. An agent *performs* an operation if it is effectively allocated to it; the agent *monitors/controls* an object if the states of the object are observable/controllable by it. Agents can be humans, devices, programs, etc.
- *Goal:* A goal is an objective the composite system should meet. *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. The goal refinement structure for a given system can be represented by an AND/OR directed acyclic graph. Goals *concern* the objects they refer to.
- *Requisite, requirement,* and *assumption:* A requisite is a goal that can be formulated in terms of states controllable by some individual agent. Goals must be eventually AND/OR *refined* into requisites assignable to individual agents. Requisites, in turn, are AND/OR *operationalized* by operations and objects through strengthenings of their domain pre/postconditions and invariants, respectively, and through trigger conditions. Alternative ways of assigning responsible agents to a requisite are captured through AND/OR *responsibility* links; the actual assignment of an agent to the operations that operationalize the requisite is captured in the corresponding *performance* links. A requirement is a requisite assigned to a software agent; an assumption is a requisite assigned to an environmental agent. Unlike requirements, assumptions cannot be enforced in general [15], [32].
- *Scenario:* A scenario is a domain-consistent composition of applications of operations by corresponding agent instances; domain-consistency means that the operations are applied in states satisfying their domain precondition together with the various domain invariants attached to the corresponding objects, with resulting states satisfying their domain postcondition. The composition modes include sequential, alternative, and repetitive composition.

2.1.2 Language Constructs

Each construct in the KAOS language has a two-level generic structure: an outer semantic net layer [5] for *declaring* a concept, its attributes, and its various links to other concepts; an inner formal assertion layer for *formally defining* the concept. The declaration level is used for conceptual modeling (through a concrete graphical syntax), requirements traceability (through semantic net navigation), and specification reuse (through queries) [11]. The assertion level is optional and used for formal reasoning [9], [10], [32], [33].

The generic structure of a KAOS construct is instantiated to specific types of links and assertion languages according to the specific type of the concept being specified. For example, consider the following goal specification for a meeting scheduler system:

Goal Achieve[ParticipantsConstraintsKnown]
 Concerns Meeting, Participant, Scheduler
 Refines MeetingPlanned
 RefinedTo ConstraintsRequested, ConstraintsProvided
 InformalDef *A meeting scheduler should know the constraints of the various participants invited to the meeting within some deadline d after invitation.*
 FormalDef $\forall$ m: Meeting, p: Participant, s: Scheduler
 Invited(p, m) $\wedge$ Scheduling(s, m)
 $\Rightarrow \Diamond_{\leq d}$ Knows(s, p.Constraints)

The declaration part of this specification introduces a concept of type "goal," named **ParticipantsConstraintsKnown**, stating a required property that should eventually hold ("**Achieve**" verb), referring to objects such as **Participant** or **Scheduler**, refining the parent goal **MeetingPlanned**, refined into subgoals **ConstraintsRequested** and **ConstraintsProvided**, and defined by some informal statement. (The semantic net layer is represented in textual form in this paper for reasons of space limitations; the reader may refer to [11] to see what the alternative graphical concrete syntax looks like.)

The assertion defining this goal formally is written in a real-time temporal logic inspired from [28]. In this paper, we will use the following classical operators for temporal referencing [34]:

$\circ$	(in the next state)	$\bullet$	(in the previous state)
$\Diamond$	(some time in the future)	$\blacklozenge$	(some time in the past)
$\Box$	(always in the future)	$\blacksquare$	(always in the past)
$\mathcal{W}$	(always in the future *unless*)	$\mathcal{U}$	(always in the future *until*)

Formal assertions are interpreted over historical sequences of states. Each assertion is in general satisfied by some sequences and falsified by some other sequences. The notation

$$(H, i) \models P$$

is used to express that assertion P is satisfied by history H at time position i (i ∈ T), where T denotes a linear temporal structure assumed to be discrete in this paper for sake of simplicity. States are global; the *state* of the composite system at some time position i is the aggregation of the local states of all its objects at that time position. The state of an individual object instance *ob* at some time position is defined as a mapping from *ob* to the set of values of all *ob*'s attributes and links at that time position. In the context of KAOS requirements, an historical sequence of states corresponds to a behavior or *scenario*.

The semantics of the above temporal operators is then defined as usual [34], e.g.,

$(H, i) \models \circ P$	iff	$(H, next(i)) \models P$
$(H, i) \models \Diamond P$	iff	$(H, j) \models P$ for some $j \geq i$
$(H, i) \models \Box P$	iff	$(H, j) \models P$ for all $j \geq i$
$(H, i) \models P \mathcal{U} Q$	iff	there exists a $j \geq i$ such that $(H, j) \models Q$ and for every k, $i \leq k < j$, $(H, k) \models P$
$(H, i) \models P \mathcal{W} Q$	iff	$(H, i) \models P \mathcal{U} Q$ or $(H, i) \models \Box P$

Note that $\Box$ P amounts to P$\mathcal{W}$ **false**. We will also use the standard logical connectives ∧ (and), ∨ (or), ¬ (not), → (implies), ↔ (equivalent), ⇒ (entails), ⇔ (congruent), with

$$P \Rightarrow Q \quad \text{iff} \quad \Box(P \rightarrow Q)$$

$$P \Leftrightarrow Q \quad \text{iff} \quad \Box(P \leftrightarrow Q)$$

There is thus an implicit outer $\Box$-operator in every entailment.

To handle real requirements, we often need to introduce real-time restrictions. We therefore introduce bounded versions of the above temporal operators in the style advocated by [28], such as

$\Diamond_{\leq d}$ (some time in the future within deadline *d*)

$\Box_{\leq d}$ (always in the future up to deadline *d*)

To define such operators, the temporal structure T is enriched with a metric domain D and a temporal distance function *dist:* T × T → D, which has all desired properties of a metrics [28]. A frequent choice is

T:	the set of naturals
D:	{ d : there exists a natural *n* such that d = n × *u*}, where *u* denotes some chosen time unit
dist(i, j):	| j - i | × u

Multiple units can be used (e.g., second, day, week); they are implicitly converted into some smallest unit. The o-operator then yields the nearest subsequent time position according to this smallest unit.

The semantics of the real-time operators is then defined accordingly, e.g.,

$(H, i) \models \Diamond_{\leq d} P$	iff	$(H, j) \models P$ for some $j \geq i$ with $dist(i, j) \leq d$
$(H, i) \models \Box_{<d} P$	iff	$(H, j) \models P$ for all $j \geq i$ such that $dist(i, j) < d$

Back to the formal assertion of the goal **ParticipantsConstraintsKnown** above, one may note that the scheduler **s** in the current state when **Invited(p,m)** ∧ **Scheduling(s,m)** holds should be the scheduler at the future time within deadline **d** when **Knows(s,p.Constraints)** will hold; this will be expressed in the formal assertion capturing another subgoal required to achieve the parent goal **MeetingPlanned**, namely, the goal **SameScheduler**. The conjunction of the formal assertions of subgoals **SchedulerAppointed**, **ParticipantsConstraintsKnown**, **ConvenientMeetingScheduled**, and **SameScheduler** must entail the formal assertion of the parent goal **MeetingPlanned** they refine altogether. Every formal goal refinement thus generates a corresponding proof obligation [10].

In the formal assertion of the goal **ParticipantsConstraintsKnown**, the predicate **Invited(p,m)** means that, in the current state, an instance of the **Invited** relationship links variables **p** and **m** of sort **Participant** and **Meeting**, respectively. The **Invited** relationship, **Participant agent**, and **Meeting** entity are defined in other sections of the specification, e.g.,

```
Agent Participant
  CapableOf CommunicateConstraints, ...
  Has Constraints:
        Tuple [ExcludedDates: SetOf[TimeInterval],
               PreferredDates: SetOf[TimeInterval]]
Relationship Invited
  Links Participant {card 0:N}, Meeting {card 1:N}
  InformalDef A person is invited to a meeting iff
    she appears in the list of expected participants
    specified in the meeting initiator's request.
 DomInvar ∀p: Participant, m: Meeting, i: Initiator
          Invited(p, m) ⇔
              p∈ Requesting[i,m].ParticipantsList
```

In the declarations above, **Constraints** is declared as an attribute of **Participant** (this attribute was used in the formal definition of **ParticipantsConstraintsKnown**); **ParticipantsList** is used as an attribute of the **Requesting** relationship that links initiators and meetings.

As mentioned earlier, operations are specified formally by pre- and postconditions in the state-based style [20], [43], e.g.,

```
Operation DetermineSchedule
  Input Requesting, Meeting {Arg m};
  Output Meeting {Res m}
  DomPre ¬ Scheduled(m)
        ∧ (∃i: Initiator) Requesting(i, m)
  DomPost Feasible(m) ⇒ Scheduled(m)
        ∧ ¬ Feasible(m) ⇒ DeadEnd(m)
```

Note that the invariant defining **Invited** is not a requirement, but a *domain description* in the sense of [24]; it specifies what being invited to a meeting does precisely mean in the domain. The pre- and postcondition of the operation **DetermineSchedule** above are domain descriptions as well; they capture corresponding elementary state transitions in the domain, namely, from a state where the meeting is not scheduled to a state where the meeting is scheduled under some condition.

The software requirements are found in the terminal goals assigned to software agents (e.g., the goal **Convenient MeetingScheduled** assigned to the **Scheduler** agent), and in the additional pre-/postconditions that need to strengthen the corresponding domain pre- and postcondition in order to ensure all such goals [9], [30], e.g.,

```
Operation DetermineSchedule
  ...
  RequiredPost for ConvenientMeetingScheduled:
        Scheduled(m) ⇒ Convenient(m)
```

2.2 The Elaboration Method

The following steps may be followed to systematically elaborate KAOS specifications from high-level goals:

- *Goal elaboration:* Elaborate the goal AND/OR structure by defining goals and their refinement/conflict links until assignable requisites are reached. The process of identifying goals, defining them precisely, and relating them through positive/negative contribution links is, in general, a combination of top-down and bottom-up subprocesses; offspring goals are identified by asking HOW questions about goals already identified, whereas parent goals are identified by asking WHY questions about goals and operational requirements already identified. Goals can also be identified in the first place from interviews and analysis of available documentation to find out problematic issues with the existing system, objectives that are explicitly stated about the envisioned one, operational choices whose rationale has to be elicited, etc. Other techniques for goal identification may include obstacle analysis [32], scenario-based elicitation [33], and analogical reuse of goal structures [36].
- *Object capture:* Identify the objects involved in goal formulations, define their conceptual links, and describe their domain properties.
- *Operation capture:* Identify object state transitions that are meaningful to the goals. Goal formulations refer to desired or forbidden states that are reachable by state transitions; the latter correspond to applications of operations. The principle is to specify such state transitions as domain pre- and postconditions of operations thereby identified and to identify agents that could have those operations among their capabilities.
- *Operationalization:* Derive strengthenings on operation pre-/postconditions and on object invariants in order to ensure that all requisites are met. Formal derivation rules are available to support the operationalization process [9].
- *Responsibility assignment:* Identify alternative responsibilities for requisites; make decisions among refinement, operationalization, and responsibility alternatives—with process-level objectives such as: reduce costs, increase reliability, avoid overloading agents, resolve conflicts (see below); assign the operations to agents that can commit to guaranteeing the requisites in the alternatives selected. The boundary between the system and its environment is obtained as a result of this process, and the various requisites become requirements or assumptions.

The steps above are ordered by data dependencies; they may be running concurrently, with possible backtracking at every step.

2.3 Using Metalevel Knowledge

At each step of the goal-driven method, domain-independent knowledge can be used for local guidance and validation in the elaboration process.

- A rich taxonomy of goals, objects and operations is provided together with rules to be observed when specifying concepts of the corresponding subtype. We give a few examples of such taxonomies.
  - Goals are classified according to the pattern of temporal behavior they require:

    Achieve: $P \Rightarrow \Diamond Q$ or *Cease:* $P \Rightarrow \Diamond \neg Q$
    Maintain: $P \Rightarrow Q \mathcal{W} R$ or *Avoid:* $P \Rightarrow \neg Q \mathcal{W} R$
  - Goals are also classified according to the category of requirements they will drive with respect to the agents concerned (e.g., **SatisfactionGoals** are functional goals concerned with satisfying agent requests; **InformationGoals** are goals concerned with keeping agents informed about object states; **SecurityGoals** are goals concerned with maintaining secure access to objects by agents; other categories include **SafetyGoals**, **AccuracyGoals**, etc.).

  Such taxonomies are associated with heuristic rules that may guide the elaboration process, e.g.,

  - **SafetyGoals** are **AvoidGoals** to be refined in **HardRequirements**;
  - **ConfidentialityGoals** are **AvoidGoals** on ***Knows*** predicates. (**Knows** is a KAOS built-in predicate that was already used in the goal **ParticipantsConstraintsKnown** above and will still be used further in this paper; for an agent instance **ag** and an object instance **ob**, the predicate **Knows(ag,ob)** means that the state of **ob** in **ag**'s local memory coincides with the actual state of **ob**.)

Similar rules will be presented for conflict detection and resolution in Sections 4.3 and 5.1.6, respectively.

- Tactics capture heuristics for driving the elaboration or for selecting among alternatives, e.g.,
  - Refine goals so as to reduce the number of agents involved in the achievement of each subgoal;
  - Favor goal refinements that introduce fewer conflicts (see Section 5.1.5).

Goal verbs such as ***Achieve/Maintain*** and categories such as ***Satisfaction/Information*** are language keywords that allow users to specify more information at the declaration level; for example, the declaration

Achieve[**ParticipantsConstraintsKnown**]

allows specifiers to state in a lightweight way that the property named **ParticipantsConstraintsKnown** should eventually hold, without entering into the temporal logic level. (We avoid the classical safety/liveness terminology here to avoid confusions with SafetyGoals.)

To conclude this short overview of KAOS, we would like to draw the reader's attention on the complementarity between the outer semiformal layer and the inner formal layer. At the semantic net level, the user builds her requirements model in terms of concepts whose meaning is annotated in **InformalDef** attributes; the latter are placeholders for the designation of objects and operations [57] and for the informal formulation of goals, requirements, assumptions, and domain descriptions. At the optional formal assertion level, more advanced users may make such formulations more precise, fixing problems inherent to informality [38], and apply various forms of formal reasoning, e.g., for goal refinement and exploration of alternatives [10], requirements derivation from goals [9], obstacle analysis [32], requirements/assumptions monitoring [15], or conflict analysis as shown in this paper. Our experience so far with five industrial projects in which KAOS was used reveals that the semiformal semantic net layer is easily accessible to industrial users; the formal assertion layer proved effective after the results of formal analysis by trained users were propagated back to the semiformal semantic net layer.

3 Inconsistencies in Goal-Driven Requirements Engineering

This section introduces the scope of inconsistency management in requirements engineering. Some linguistic support is then introduced for capturing multiple stakeholder views. Various types of inconsistency are then defined in this framework.

3.1 Scope of Inconsistency Management

Fig. 1 introduces the various levels at which requirements-related inconsistencies can occur.

At the *product level*, the requirements model is captured in terms of goals, agents, objects, operations, etc. These are artifacts elaborated according to the process model defined at the *process level*. The latter model is captured in terms of process-level objectives, actors, artifacts, elaboration operators, etc. (We use a different terminology for the same abstractions at the product and process levels in order to avoid confusions between these two levels.) At the *instance level*, operation instances are executed on object instances in the running system according to the requirements specified at the product level.

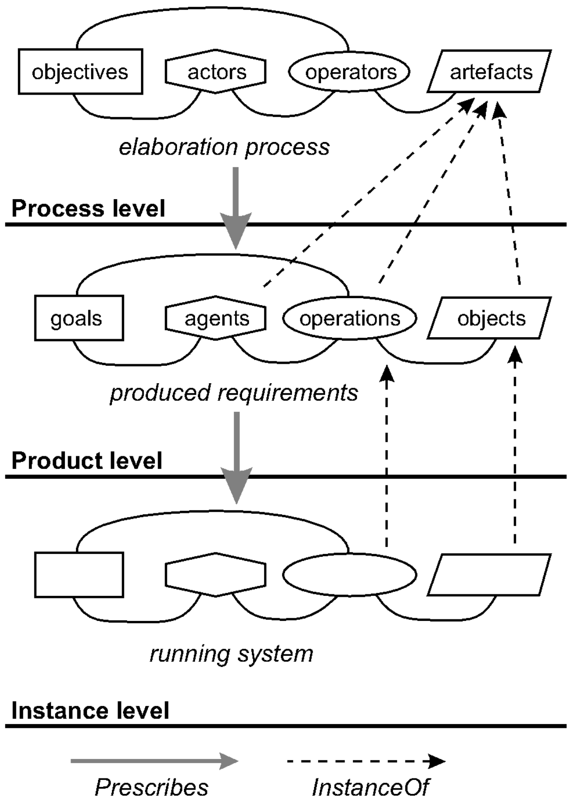

Fig. 1. The process, product, and instance levels.

Consider a meeting scheduler system to help visualize those levels. A **MeetingOrganizer** actor involved as stakeholder in the requirements engineering process may have identified a product-level goal like **Achieve[Participants ConstraintsKnown]** whose satisfaction requires the cooperation of product-level agents such as **Participant** and **Scheduler**; this goal has been produced at the process level through the **RefineGoal** operator applied to the **Achieve[MeetingPlanned]** artifact to meet the process-level objective **Achieve[GoalsOperationalized]**.

As will be seen in Section 3.3, inconsistencies can arise between levels or within the same level.

3.2 Capturing Multiple Views

The various activities at the process level involve multiple actors—clients, users, domain specialists, requirements engineers, software developers, and so forth. Different actors, in general, have different concerns, perceptions, knowledge, skills, and expression means. Requirements completeness and adequacy requires that all relevant viewpoints be expressed and eventually integrated in a consistent way. (In practice, viewpoints may be assigned different weights depending on the actor's status or system priorities; this aspect is not considered further in the paper.) Inconsistency management thus partly relies on some mechanism for capturing conflicting assertions from multiple viewpoints.

Support for multiple viewpoints has been advocated since the early days of requirements engineering [49]. Proposals for viewpoint-based acquisition, negotiation, and cooperative modeling appeared only later [17], [46], [44], [4], [40]. Two kinds of approaches have emerged. In the *multiparadigm* approach, specifications for different viewpoints can be written in different notations. Multiparadigm viewpoints are analyzed in a centralized or distributed way. Centralized viewpoints are translated into some logic-based "assembly" language for global analysis; viewpoint combination then amounts to some form of conjunction [41], [56]. Distributed viewpoints have specific process-level rules associated with them; checking the process-product consistency is made by evaluating the corresponding rules on pairs of viewpoints [42]. In the *single-paradigm* approach, specifications for different views are written in a single language; the same conceptual unit is manipulated from one view to another under different structures, different foci of concern, etc. Consistency may then be prescribed explicitly through inter-view invariants or implicitly through synchronization of operations [25].

Our view construct is introduced to

1) restrict the visibility of conceptual features to specific contexts associated with corresponding actors, and
2) allow product-level inconsistencies to be captured and recorded for later resolution.

Views here are simpler than [42] in that they contain no process-level information such as the workplan to be followed or the development history; they may also contain formal specifications; they may be explicitly related to each other and factored out in case of conceptual overlap [53].

A *view* is defined as a ternary relationship linking a process-level actor, a master concept and a facet of it (see Fig. 2; we use the generic term "concept" here to denote a product-level artifact such as a goal, an object, an operation, an agent, etc.). A master concept may thereby be viewed under different facets by different actors. Note that a single concept facet can be shared among different actors (see the cardinality constraints in Fig. 2).

For example, one might introduce the master concept

```
Entity Meeting
  Has Date: Calendar; Location: String
```

and the two following facets of it:

```
Entity MeetingToBeScheduled
  FacetOf Meeting SeenBy MeetingOrganizer
  Has    ExcludedDates: SetOf[Calendar]
         RequiredEquipment: SetOf[String]
  DomInvar  ∀ m: Meeting
            m.Date not in m.ExcludedDates
```

and

```
Entity MeetingToAttend
  FacetOf Meeting SeenBy PotentialAttendant
  Has  MeetingPriority: {low, medium, high}
```

Master and facet concepts are characterized by features. A *feature* corresponds to an attribute declaration, a link to other concepts in the semantic net, or a formal assertion. For example, the **MeetingToBeScheduled** facet seen by the **MeetingOrganizer** actor has features such as the declarations of **ExcludedDates** and **RequiredEquipment** and

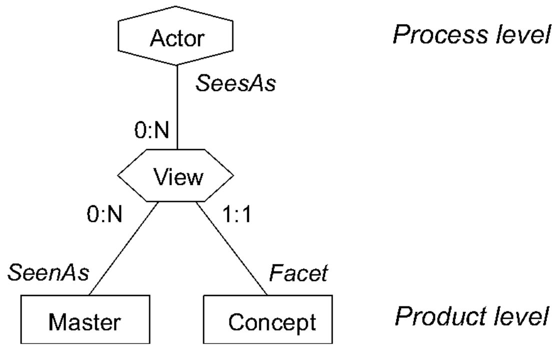

Fig. 2. Modeling views.

the corresponding domain invariant on date exclusions.

An actor's view of a concept is defined by joining the features of the master concept and the features of the facet seen by the actor. For example, the view of the **MeetingOrganizer** actor includes the features of the **MeetingToBeScheduled** facet plus the declarations of the **Date** and **Location** attributes of the master concept. A view thus imports all features from the corresponding master and facet concepts; the master features belong to all views of the concept. A master concept always has at least one feature, namely, the concept identifier; this mechanism allows different local names (e.g., **MeetingToBeScheduled**) to be associated with a unique master name (e.g., **Meeting**).

Different views of a same concept *overlap* when their respective facets share a common attribute, link, or predicate in their formal assertions. For example, the two following goals in a resource management system are overlapping as they refer to a common predicate **Using**:

```
Goal Achieve[ResourceKeptAsLongAsNeeded]
  FacetOf SatisfactoryAvailability SeenBy User
  FormalDef ∀u: User, r: Resource
            Using(u,r) ⇒ o[Needs(u,r) → Using(u,r)]
Goal Achieve[ResourceEventuallyAvailable]
  FacetOf SatisfactoryAvailability SeenBy Staff
  FormalDef ∀u: User, r: Resource
            Using(u,r) ⇒ ◊≤d ¬ Using(u,r)
```

As we will see in Section 3.3.7, the formal assertions that make them overlapping are in fact *divergent*. (In passing, note the use of implication rather than entailment in the consequent of the first goal above; replacing the "$\rightarrow$" connective by "$\Rightarrow$" would be too strong as this would require user **u**, once she has been using a resource, to get it in the future as soon as she needs it again, see the definition of these connectives in Section 2.1.2.)

Views thus allow conceptual features to be structured into specific contexts associated with corresponding actors; they also allow product-level inconsistencies to be captured and recorded for later resolution. Views associated with the same actor can be grouped into collections associated with the actor to form so-called *perspectives*.

3.3 Classifying Inconsistencies

A set of descriptions is inconsistent if there is no way to satisfy those descriptions all together. Depending on what such descriptions capture, one may identify various kinds of inconsistency.

3.3.1 Process-Level Deviation

A *process-level deviation* is a state transition in the RE process that results in an inconsistency between a process-level rule of the form

`(∀r: Requirement) R(r)`

and a specific process state characterized by

`¬ R(req)`

for some requirement **req** at the product level. This is a particular case of the notion of deviation in [7]. For example, assigning responsibility for the goal **Achieve[Participants ConstraintsKnown]** jointly to the **Participant** and **Scheduler** agent types would result in a violation of the process-level rule stating that responsibility must always be assigned to single agents (see Section 2.1). Violations of interviewpoint rules in [42] also illustrate this kind of inconsistency.

3.3.2 Instance-Level Deviation

An *instance-level deviation* is a state transition in the running system that results in an inconsistency between a product-level requirement of the form

`(∀x: X) R(x)`

and a specific state of the running system characterized by

`¬ R(a)`

for some specific value a in **X**. Going back to our meeting scheduler example, the participant instance **Jeff** failing to provide his constraints for the **Icse99-PC** meeting produces an instance-level deviation resulting in a runtime inconsistency with the goal **ConstraintsProvided**. Techniques for detecting and resolving runtime deviations are discussed in [15].

The above types of inconsistency involve two levels from Fig. 1. Intralevel inconsistencies correspond to inconsistent objectives and rules at the process level; inconsistent requirements at the product level; or inconsistent states at the instance level (like different values for **Icse99-PC.Date** in **Jeff**'s agenda and the scheduler's memory, respectively). We now focus on inconsistent goals/requirements at the product level.

3.3.3 Terminology Clash

A *terminology clash* occurs when a single real-world concept is given different syntactic names in the requirements specification. This type of inconsistency may often be found among views owned by multiple stakeholders. For example, a participant **p** attending some meeting **m** might be formalized as **Attends(p,m)** in some specification fragment and as **Participates(p,m)** in some other.

3.3.4 Designation Clash

A *designation clash* occurs when a single syntactic name in the requirements specification designates different real-world concepts. (As in [57], we use the term "designation" for the notion of interpretation function in logic.) This type of inconsistency again is typically found among views owned by multiple stakeholders who interpret a single name in different ways. For example, a predicate such as **Attends(p,m)** might be interpreted as "**attending meeting m until the end**" in some view and as "**attending part of meeting m only**" in some other. A study of the London Ambulance System report [18] reveals many inconsistencies of this type.

3.3.5 Structure Clash

A *structure clash* occurs when a single real-world concept is given different structures in the requirements specification. An example of this type of inconsistency within a single view is analyzed in [38]; Goodenough and Gerhart's informal specification of Naur's text formatting problem refers to a text as a sequence of characters at one point and as a sequence of words (that is, a sequence of sequences of characters) at another. Back to the declaration of the **Participant** agent in Section 2.1.2, a structure clash would occur if the **ExcludedDates** subattribute had been declared as **SetOf [TimePoint]** in another view instead of **SetOf [TimeInterval]**.

Resolving structure clashes can be done by application of restructuring operators [12] or by addition of mapping invariants in the style advocated in [25]. The absence of structure clash is called type correctness in some consistency checking tools [22].

The next types of inconsistency are defined in more technical terms as they will be studied throughout the rest of the paper. Let $A_1, ..., A_n$ be assertions, each of them formalizing a goal, a requirement or an assumption. Let *Dom* denote a set of domain descriptions [24] that captures the knowledge available about the domain.

3.3.6 Conflict

A *conflict* between assertions $A_1, ..., A_n$ occurs within a domain *Dom* iff the following conditions hold:

1) $\{Dom, \bigwedge_{1 \leq i \leq n} A_i\} \vdash \textbf{false}$ (*logical inconsistency*)

2) for every i: $\{Dom, \bigwedge_{j \neq i} A_j\} \nvdash \textbf{false}$ (*minimality*)

Condition 1 states that the assertions $A_1, ..., A_n$ are logically inconsistent in the domain theory *Dom* or, equivalently, that the negation of any of them can be inferred from the other assertions in this theory; Condition 2 states that removing any of those assertions no longer results in a logical inconsistency in that theory. We are indeed interested in focussing on minimal situations for conflicts to occur. To give a trivial example, the three propositional assertions below are conflicting, whereas they are not logically inconsistent pairwise:

$$P \qquad Q \qquad P \Rightarrow \neg Q$$

Thus, an n-ary conflict over a set *S* of *n* assertions cannot be "contracted," that is, it cannot be an *m*-ary conflict over a proper subset of *S* involving *m* assertions (m < n); this directly follows from the minimality condition. Likewise, an *n*-ary conflict over a set *S* of assertions cannot be "extended," that is, it cannot be a *p*-ary conflict over a proper superset *S′* of *S* (p > n), for if it was a conflict over *S′*, it won't be a conflict over the subset *S* of *S′* by the minimality condition again. The minimality condition thus implies that the removal of any assertion from a conflicting set makes the resulting set no longer conflicting.

It is easy to see that binary conflicts give rise to an irreflexive, nontransitive, and symmetrical relation.

Let us give a simple example of conflicting assertions. In the context of specifying a device control system, requirements that constrain the possible states of the **Device** agent are elicited from different stakeholders, each providing a fragmentary view as follows.

View1:	$\text{InOperation} \Rightarrow \text{Running}$
View2:	$\text{InOperation} \Rightarrow \text{Startup}$
	$\text{Startup} \Rightarrow \neg\, \text{Running}$
View3:	$\Box\, \text{InOperation}$

(For simplicity, the perceived states of the device are formalized by atomic propositional formulas.) In the first view, the **InOperation** mode is required to be covered by the **Running** state. The second view somewhat complements the first one by introducing the **Startup** state as another state covering the **InOperation** mode, disjoint from the **Running** state. The third view requires the **Device** agent to be always in the **InOperation** mode. It is easy to check that the four assertions from these three views are conflicting; a first derivation using modus ponens once yields **Running**, whereas a second derivation using modus ponens twice yields **¬Running**; removing any of those assertions makes it impossible to reach such contradictory conclusions.

The conflicting viewpoints in [13], in which a property and its negation can both be derived when putting the viewpoints together, adhere to the definition above. Violation of the disjointness property in [22], which requires that, in a given state, each controlled variable, mode class, and term in a SCR specification be defined uniquely, may also be seen as a particular case of this kind of inconsistency; nondisjointness may result in a single output variable/term being prescribed different, incompatible values. Another particular case is the inconsistency considered in [21], arising when the guarding condition on more than one transition in a finite state machine can be satisfied simultaneously.

It is our experience, however, that a weaker form of conflict, not studied before in the literature, occurs highly frequently in practice. We define it now.

3.3.7 Divergence

A *divergence* between assertions $A_1, ..., A_n$ occurs within a domain *Dom* iff there exists a *boundary condition* B such that the following conditions hold:

1) $\{\text{Dom}, B, \bigwedge_{1\le i\le n} A_i\} \vdash \textbf{false}$ (*logical inconsistency*)
2) for every i: $\{\text{Dom}, B, \bigwedge_{j\ne i} A_j\} \not\vdash \textbf{false}$ (*minimality*)
3) there exists a scenario S and time position i such that $(S, i) \models B$ (*feasibility*)

The boundary condition captures a particular combination of circumstances which makes assertions $A_1, ..., A_n$ conflicting if conjoined to it (see Conditions 1 and 2).

Condition 3 states that the boundary condition is satisfiable through one behavior, at least, that is, there should be at least one scenario that establishes the boundary condition. Clearly, it makes no sense to reason about boundary conditions that cannot occur through some feasible agent behavior.

Note that a conflict is a particular case of divergence in which B = **true**. Also note that the minimality condition precludes the trivial boundary condition B = **false**; it stipulates, in particular, that the boundary condition must be consistent with the domain theory *Dom*.

The following variant of Condition 1 above will be used frequently in practice:

$$\{\text{Dom}, B, \bigwedge_{j\ne i} A_j\} \vdash \neg\, A_i$$

To give a first, simplified example, consider a resource management system and the goal **SatisfactoryAvailability** that appeared at the end of Section 3.2. In the user's view, the goal

$$\forall u\text{: User}, r\text{: Resource}$$
$$\text{Using}(u, r) \Rightarrow \circ\, [\text{Needs}(u, r) \rightarrow \text{Using}(u, r)]$$

states that if a user is using a resource then she will continue to do so as long as she needs it, whereas in the staff's view the goal

$$\forall u\text{: User}, r\text{: Resource}$$
$$\text{Using}(u, r) \Rightarrow \Diamond_{\le d}\, \neg\, \text{Using}(u, r)$$

states that if a user is using a resource then within some deadline d she will no longer do so.

These two goals are *not* conflicting. However, they are divergent because there is an obvious boundary condition, namely,

$$\Diamond\, (\exists u'\text{:User}, r'\text{:Resource})\ [\text{Using}(u', r') \wedge \Box_{\le d}\, \text{Needs}(u', r')]$$

The latter is satisfied by some behavior in which, at some time point, some user is using some resource and needs it for two weeks at least. This clearly makes the three assertions conflicting altogether. (In Section 4.1, we will see how such a condition can be derived formally.)

Various subtypes of divergence can be identified according to the *temporal evolution* of the boundary condition B:

occasional divergence:	$\Diamond\, B$
intermittent divergence:	$\Box \Diamond\, B$
persistent divergence:	$\Diamond\, (B\ \mathcal{U}\ C)$
permanent divergence:	$\Diamond \Box\, B$
perpetual divergence:	$\Box\, B$

For example, the divergence on satisfactory availability above is likely to be intermittent and persistent. In the London Ambulance system [18], the divergence between the goal ***Achieve*[NearestFreeAmbulanceDispatched]** in the patient's view and the goal ***Maintain*[AmbulanceCloseToStation]** in the driver's view is intermittent and persistent as well because of the possibility of repeated accidents occurring far from the station with no other ambulance being available.

Divergences can also be classified according to the category of the diverging goals. Examples of divergence *categories* include **ResourceConflict** divergences between Satisfaction goals related to resource requests; **ConflictOfInterest** divergences between **Optimize** goals, in which one goal is concerned with maximizing some quantity whereas others are concerned with minimizing that quantity. For example, the goal **Maximize[Profit]** in a company's view and the goal **Minimize[Costs]** in the customer's view result in a **ConflictOfInterest** divergence; in the London Ambulance system, a **ConflictOfInterest** divergence results from the goal **Maximize[NumberOfAmbulances]** in a patient's view, that refines **Maximize[QualityOfService]**, and the goal **Minimize[CostOfService]** in the management's view. As will be shown in Sections 4.3 and 5.1.6, goal categories may be used for defining heuristics for divergence detection and resolution, respectively.

We conclude this classification of inconsistencies with two particular cases of divergence within a single view.

3.3.8 Competition

A *competition* is a particular case of divergence within a single goal/requirement; it is characterized by the following conditions:

1) The goal assertion takes the form $(\forall x{:}X)\ A[x]$
2) $\{Dom, B, \bigwedge_{i \in I} A[x_i]\} \vdash$ **false** for some I
3) $\{Dom, B, \bigwedge_{i \in J} A[x_i]\} \nvdash$ **false** for any $J \subset I$
4) There exists a scenario S and time position i such that

$$(S, i) \models B$$

A competition thus corresponds to the case where different instantiations $A[x_i]$ of the same universally quantified goal/requirement $\forall x{:}A[x]$ are divergent.

For example, there is a binary competition within the meeting scheduling goal

```
∀m: Meeting, i: Initiator, p: Participant
Requesting(i, m) ∧ Invited(p, m)
⇒ ◊ (∃d: Calendar) [ m.Date = d ∧ Convenient(d, m, p)],
```

because of a possible boundary condition:

```
◊∃m,m': Meeting, p: Participant, i,i': Initiator
Requesting(i,m) ∧ Requesting(i',m')
∧ Invited(p,m) ∧ Invited(p,m')
∧ □ [(∃ d: Calendar) Convenient(d,m,p)
      ∧ (∃ d': Calendar) Convenient(d',m',p)
      ∧ ¬ (∃ d,d': Calendar)
           Disjoint(d,d') ∧ Convenient(d,m,p)
                          ∧ Convenient(d',m',p)]
```

This condition captures a situation of two requested meetings involving a common invited participant for which convenient dates can be found in isolation but not jointly.

As another example found in the London Ambulance system, two instantiations of the goal **Achieve[NearestFree AmbulanceDispatched]** are competing because of a boundary condition of the same ambulance being the nearest to two simultaneous accidents.

3.3.9 Obstruction

An *obstruction* is a borderline case of divergence in which only one assertion is involved; it is defined by taking $n=1$ in the general characterization of divergence:

1) $\{Dom, B, A\} \vdash$ **false**
2) $\{Dom, B\} \nvdash$ **false**
3) There exists a scenario S and time position i such that

$$(S, i) \models B$$

The boundary condition then amounts to an *obstacle* that obstructs the goal assertion [45], [32]; the minimality condition now states that the obstacle must be consistent with the domain.

For example, the goal ***Achieve*****[InformedParticipants Attendance]** formalized by

```
∀m: Meeting, p: Participant
Invited(p,m) ∧ Informed(p,m) ∧ Convenient(m.Date,m,p)
⇒ ◊ Participates(p,m)
```

may be obstructed by the obstacle **LastMinuteImpediment** formalized by

```
◊∃m: Meeting, p: Participant
Invited(p,m) ∧ Informed(p,m) ∧ Convenient(m.Date,m,p)
∧ □ (IsTakingPlace(m) → ¬ Convenient(m.Date,m,p))
```

This obstacle captures scenarios in which a participant has been informed of a meeting whose date is convenient to her at that time position, but no longer at subsequent time positions where the meeting is taking place.

Divergence analysis and obstacle analysis have different foci of concern. The former is aimed at coping with multiple goals that diverge, whereas the latter is aimed at coping with single goals that are overideal and/or unachievable. Obstacle identification and resolution provide the basis for defensive requirements engineering. Although there are generic similarities between some of the detection/resolution techniques, we will not discuss obstacle analysis any further here; the interested reader may refer to [32].

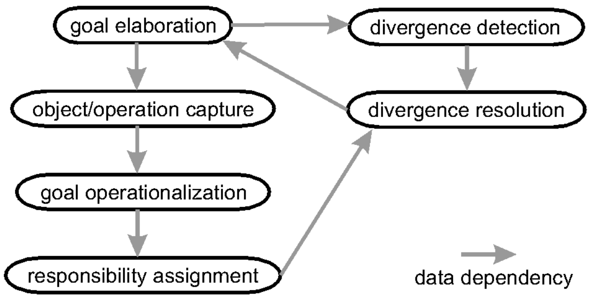

Fig. 3. Conflict management in goal-driven RE.

3.4 Integrating Conflict Management in the Requirements Elaboration Process

The integration of conflict management in the goal-driven process outlined in Section 2.2 is suggested in Fig. 3.

The main difference is the right part of it. During elaboration of the goal graph by elicitation from multiple stakeholders (asking WHY questions) and by refinement (asking HOW questions), views are captured and divergences are detected in goal specifications found in different views or within a single view. The divergences are resolved when they are identified *or* later on in the process when operational requisites have been derived from the divergent goals and responsibility assignment decisions are to be made. Resolution results in a goal structure updated with new goals and/or transformed versions of existing ones (see Section 5.1). These goals in turn may refer to new objects/operations and require specific operationalizations (see Section 5.2).

Some key questions arising here are: When exactly should divergences be identified? When exactly should an identified divergence be resolved? A definitive answer to these important questions is out of the scope of this paper; we just make a few trade-offs explicit here.

Divergences should be identified as early as possible in the goal elaboration process as they encourage the exploration of alternative OR-paths in the goal graph. However, the more specific the goals are, the more specific the boundary condition will be; divergence analysis will be much more accurate for lower-level, formalizable goals.

As will be seen in Section 5, various operators (in the sense of Fig. 1) can be followed to resolve divergences. Which operator to choose may depend on the tolerability of the divergence and of its consequences, and on the likelihood of occurrence of the boundary condition. When to apply such an operator for resolution is an open question. Too early resolution may prevent useful inferences and derivations from being made [23]; too late resolution may cause too costly backtracking to high-level goals.

The techniques discussed below for divergence detection and resolution make no assumptions about where the divergent assertions come from. They can be used for divergence analysis across multiple views or within one single view. Their presentation will therefore make no explicit use of the view mechanism presented in Section 3.2, even though many divergences arise from multiple views.

4 Detecting Divergences

Two formal techniques are proposed in this section to detect divergences (with conflicts or competitions as particular cases). The former derives boundary conditions by backward chaining; the latter relies on the use of divergence patterns.

4.1 Regressing Negated Assertions

The first technique is directly based on the common variant of the first condition characterizing divergence:

$$\{Dom, B, \bigwedge_{j \neq i} A_j\} \mid\!- \neg A_i$$

Given some goal assertion A_i, it consists of calculating preconditions for deriving the negation $\neg A_i$ backwards from the other assertions conjoined with the domain theory. Every precondition obtained yields a boundary condition. Weakest boundary conditions may be worth considering as they cover the most general combinations of circumstances to cause a conflict; however, they may sometimes be too general to find specific ways to overcome them (see examples below). This backward chaining technique amounts to a form of goal regression [54], which is the counterpart of Dijkstra's precondition calculus [20] for declarative representations. A variant of this technique can be used for obstacle derivation [32].

Let us first illustrate the idea applied to the divergent goals that appeared in Section 3.3.7, namely,

$$\forall u{:}\, User,\ r{:}\, Resource$$
$$Using(u, r) \Rightarrow o\,[\,Needs(u, r) \rightarrow Using(u, r)\,] \qquad \text{(user's view)}$$

and

$$\forall u{:}\, User,\ r{:}\, Resource$$
$$Using(u, r) \Rightarrow \Diamond_{\leq d} \neg\, Using(u, r) \qquad \text{(staff's view)}$$

The initialization step consists of taking the negation of one of these goals. Taking the staff's view, we get

$$\text{(NG)} \qquad \Diamond\, \exists\, u{:}\, User,\ r{:}\, Resource$$
$$Using(u, r) \wedge \Box_{\leq d}\, Using(u, r)$$

(Remember that there is an implicit outer $\Box$-operator in every entailment; this causes the outer $\Diamond$-operator in (NG).) For this specific user, the assertion in the first view reduces to

$$\text{(D)} \qquad o\,[Needs(u, r) \rightarrow Using(u, r)]$$

by universal instantiation and modus ponens. To make subformulas in (NG) and (D) unifiable, we rewrite them into the following logically equivalent form:

$$\text{(D')} \qquad o\,Needs(u, r) \rightarrow o\,Using(u, r)$$
$$\text{(NG')} \qquad \Diamond\, \exists\, u{:}\, User,\ r{:}\, Resource$$
$$Using(u, r) \wedge \Box_{<d}\, o\,Using(u, r)$$

(Remember that the o operator yields the nearest subsequent time position according to the smallest unit considered in the specification, see Section 2.1.2.) The subformula $o\,Using(u,r)$ in (NG') now unifies with the consequent in (D'); regressing (NG') through (D') then amounts to replacing in (NG') the matching consequent in (D') by the corresponding antecedent. We obtain:

$$\Diamond\, \exists u{:}\, User,\ r{:}\, Resource$$
$$Using(u, r) \wedge \Box_{<d}\, o\,Needs(u, r)$$

that is,

$$\Diamond\, \exists u{:}\, User,\ r{:}\, Resource)$$
$$Using(u, r) \wedge \Box_{\leq d}\, Needs(u, r)$$

We have thereby formally derived the boundary condition given in Section 3.3.7. (Note that no domain property was used in this case.)

Assuming the goals and the domain descriptions all take the form of rules $A \Rightarrow C$, the general procedure is as follows [29].

```
Initial step:   take B := ¬ $A_i$

Inductive step: let A ⇒ C be the rule selected,
                  with C matching some subformula L
                  in B;
                then  μ := mgu(L, C);
                      B := B[L/A.μ]
```

(where mgu(F1,F2) denotes the most general unifier of F1 and F2, F.μ denotes the result of applying the substitutions from unifier μ to F, and F[F1/F2] denotes the result of replacing every occurrence of F1 in formula F by F2).

Some aspects are left open in this general procedure. When several rules are applicable for the next regression, one should *first select goal assertions* prior to domain descriptions; since divergence is being sought within the assertion set, one should exploit the information from those assertions as much as possible. In the *initial step*, one should obviously select an assertion $\neg A_i$ which contains unifiable subformulas. At every step, the compatibility of the temporal scopes of the matching subformulas *L* and *C* must be checked; additional inferences and simplifications may be required to make matching subformulas refer to the same states [1]. Every iteration in the procedure above produces potentially finer boundary conditions; it is up to the requirements engineer to decide when to stop, depending on whether the boundary condition obtained is meaningful and precise enough to easily identify a scenario satisfying it and to see ways of overcoming it for divergence resolution.

In the example above, only one iteration was performed, through the other goal. The next example illustrates more iterations and the regression through domain descriptions as well. We come back to the "French reviewer" example introduced in Section 1, that is, an electronic reviewing process for a scientific journal with the following two security goals:

1) reviewers' anonymity;
2) review integrity.

(These goals could be found in different stakeholder perpectives, or within a single one.) The goals are made precise as follows:

```
Goal Maintain[ReviewerAnonymity]
 FormalDef
   ∀ r:Reviewer, p:Paper, a:Author, rep:Report
   Reviews(r, p, rep) ∧ AuthorOf(a, p)
     ⇒ □ ¬ Knows(a, Reviews[r,p,rep])
Goal Maintain[ReviewIntegrity]
 FormalDef
  ∀ r:Reviewer, p:Paper, a:Author, rep,rep':Report
  AuthorOf(a,p) ∧ Gets(a,rep,p,r)
    ⇒ Reviews(r,p,rep') ∧ rep' = rep
```

(In the formalization above, **Reviews[r,p,rep]** designates a ternary relationship capturing a reviewer **r** having produced a referee report **rep** for paper **p**; the predicate **Reviews(r,p,rep)** expresses that an instance of this relationship exists in the current state. The predicate **Gets(a,rep,p,r)** expresses that author **a** has the report **rep** by reviewer **r** for his paper **p**. The goal **ReviewerAnonymity** makes use of the KAOS built-in predicate **Knows** defined in Section 2.3.)

Let's take the goal ***Maintain*[ReviewerAnonymity]** for the initialization step. Its negation, say (**NG**), yields

```
◊ ∃ r:Reviewer, p:Paper, a:Author, rep:Report
Reviews(r,p,rep) ∧ AuthorOf(a,p)
∧ ◊ Knows(a, Reviews[r,p,rep])
```

Regressing (**NG**) through the **ReviewIntegrity** goal, whose consequent can be simplified to **Reviews(r,p,rep)** by term rewriting, yields (**NG1**):

```
◊ ∃ r:Reviewer, p:Paper, a:Author, rep:Report
AuthorOf(a,p) ∧ Gets(a, rep, p, r)
∧ ◊ Knows(a, Reviews[r,p,rep])
```

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

```
(D1) Gets(a, rep, p, r) ∧ Identifiable(r, rep)
      ⇒ ◊ Knows(a, Reviews[r,p,rep])

(D2) Reviews(r, p, rep) ∧ SignedBy(rep, r)
      ⇒ Identifiable(r, rep)
(D3) Reviews(r, p, rep) ∧ French(r)
                  ∧ ¬ ∃r' ≠ r: [Expert(r', p) ∧ French(r')]
      ⇒ Identifiable(r, rep)
```

(In the domain descriptions above, the predicate **Identifiable(r,rep)** means that the identity of reviewer **r** can be determined from the content of report **rep**; rules (**D2**) and (**D3**) provide explicit sufficient conditions for this. The predicate **SignedBy(rep,r)** means that report **rep** contains the signature of reviewer **r**; the predicate **Expert(r,p)** means that reviewer **r** is a well-known expert in the domain of paper **p**.)

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

```
◊ ∃ r:Reviewer, p:Paper, a:Author, rep:Report
AuthorOf(a, p) ∧ Gets(a, rep, p, r)
∧ Identifiable(r, rep)
```

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

```
(B) ◊∃ r:Reviewer, p:Paper, a:Author, rep:Report
    AuthorOf(a, p) ∧ Gets(a, rep, p, r)
    ∧ Reviews(r, p, rep)
    ∧ French(r) ∧ ¬ ∃r' ≠ r: Expert(r', p) ∧ French(r')
```

We have thereby formally derived the boundary condition promised in Section 1, which makes the divergent goals ***Maintain*[ReviewerAnonymity]** and ***Maintain*[ReviewIntegrity]** conflicting. It is satisfied by the scenario of a report being produced by a French reviewer who is the only well-known French expert in the domain of the paper, and then sent *unaltered* to the author (as variable ***rep*** is the same in the **Reviews** and **Gets** predicates). We will come back to this example in Section 5 to illustrate divergence resolution strategies.

Exploring the space of potential boundary conditions that can be derived from the domain theory is achieved by *backtracking* on each rule applied to select another applicable one. After having selected rule (**D3**) in the example above, one could select rule (**D2**) to derive another boundary condition:

```
(B) ◊ ∃ r:Reviewer, p:Paper, a:Author, rep:Report
    AuthorOf(a, p) ∧ Gets(a, rep, p, r)
    ∧ Reviews(r, p, rep) ∧ SignedBy(rep, r)
```

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

In practice, the domain theory does not necessarily need to be very rich at the beginning. The requirements engineer may *incrementally* elicit sufficient conditions for subformulas that are candidates to replacement in the regression process, by interaction with domain experts and clients (e.g., "what are sufficient conditions for identifiability of the reviewer from the report received?").

4.2 Using Divergence Patterns

Besides iterative regression through goal specifications and domain descriptions, one can identify frequent divergence patterns that can be formally established *once and for all* together with their associated boundary conditions. Detecting divergence between some given goals is then achieved by selecting a matching generic pattern and by instantiating it accordingly. The requirements engineer is thus relieved of the technical task of doing the formal derivations required in Section 4.1.

This approach follows the spirit of goal refinement patterns [10] or obstacle identification patterns [32]. We give a few divergence patterns here to illustrate the approach; more work is needed to reach a rich set of patterns compa-

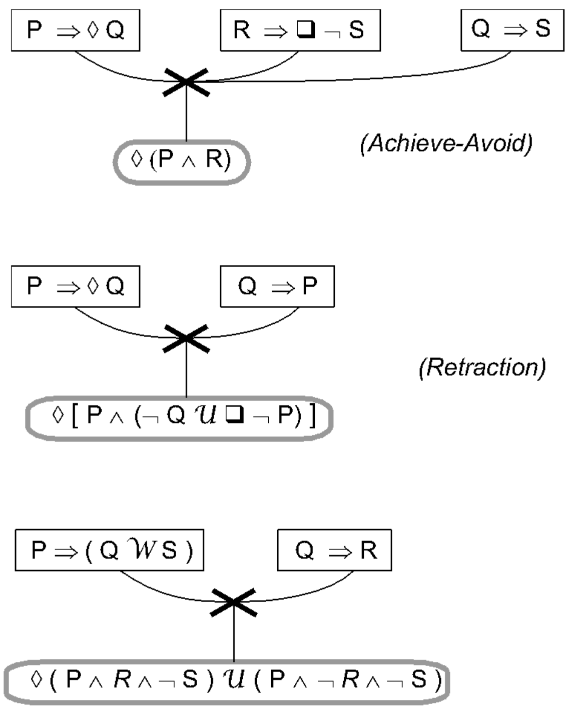

Fig. 4. Divergence patterns.

rable to the one available for goal refinement [10]. The conflicts between the formulas in the various patterns in Fig. 4 were proved formally using STeP [35].

A first obvious pattern frequently encountered involves ***Achieve*** and ***Avoid*** goals (see Fig. 4). It is easy to derive the predicate ◊ (P ∧ R) in the first pattern of Fig. 4 as a boundary condition leading to a conflict between the upper ***Achieve*** and ***Avoid*** goals conjoined with the upper right domain description (remember again that there is an implicit outer □-operator in every entailment).

As an example of reusing the **Achieve-Avoid** divergence pattern in Fig. 4, consider the following two **Satisfaction** goals in a resource management system:

```
Goal Achieve[RequestSatisfied]
  FormalDef ∀u:User, r:Resource
            Requesting(u, r) ⇒ ◊ Using(u, r)
Goal Avoid[UnReliableResourceUsed]
  FormalDef ∀u:User, r:Resource
            ¬ Reliable(r) ⇒ □ ¬ Using(u, r)
```

These two goals match the ***Achieve-Avoid*** divergence pattern in Fig. 4 with instantiations ***P*: Requesting(u,r)**, ***Q*: Using(u,r)**, ***R*: ¬Reliable(r)**, ***S*: Using(u,r)**; no domain property is necessary here. Without any formal derivation one gets the boundary condition

```
◊ ∃u:User, r:Resource
Requesting(u, r) ∧¬ Reliable(r)
```

To illustrate the use of the ***Retraction*** pattern in Fig. 4, consider a patient monitoring system with the following two assertions (expressed in propositional terms for simplicity):

(A1) Critical ⇒ ◊ Alarm

(A2) Alarm ⇒ Critical

Pairs of assertions of this form are again rather frequent. Such assertions do not appear to be divergent at first sight, although they are. They match the retraction pattern in Fig. 4 from which one may directly infer a divergence with boundary condition

$$\Diamond\,[\text{Critical} \wedge (\neg\,\text{Alarm}\;\mathcal{U}\;\Box\neg\text{Critical}\,)].$$

(Recall that $\mathcal{U}$ and $\mathcal{W}$ denote the "until" and "unless" operators, respectively.) The potential conflict between (A1) and (A2) arises from the disappearing of the critical situation before the alarm is raised. Again this has been obtained formally without any formal derivation.

Using the last pattern in Fig. 4 for a library system with the following instantiations,

P:	Borrowing(u, bc),	u: User, bc: BookCopy, lib: Library
Q:	HasPermission(u, bc) ,	
R:	Member(u, lib) ,	
S:	DueDate(u, bc) ,	

we directly infer a rather subtle potential conflict presented in [27]; the conflict arises because of a borrower losing library membership before the due date for returning her borrowed copies.

4.3 Divergence Identification Heuristics

Divergence identification heuristics are provided as rules of thumb in order to help users identify divergences without necessarily having to go through formal techniques every time. Such heuristics are derived from formal patterns of divergence or from past experience in identifying divergences. Goal classifications, as mentioned in Section 2.3, are used to define these heuristics. We give a few examples of them to illustrate the approach.

- **If** there is a **SatisfactionGoal** and a **SafetyGoal** concerning a *same* object, **then** the possibility of a divergence between those two goals should be considered. For example, a divergence between the goals **Achieve [ArrivalOnTime]** and **Avoid[TrainOnSameBlock]** might be thereby identified in a train control system.
- **If** there is a **ConfidentialityGoal** and an **InformationGoal** concerning a *same* object, **then** the possibility of a divergence between those two goals should be considered. For example, a divergence between the goals **Achieve[EditorialBoardInformed]**
- and **Maintain[ReviewerAnonymity]** might be thereby identified in an electronic journal management system because of the boundary condition of associate editors submitting papers. The same rule may lead to the identification of a divergence between the goals **Achieve[ParticipantsConstraintsKnown]** and **Maintain[ParticipantPrivacy]** in a meeting scheduler system; between the goals" **Achieve[PatientInformed]** and **Maintain[Medical Secret]** in a hospital management system; etc.
- **If** there are two **Optimize** goals interfering on a *same* object's attribute, **then** the possibility of a **Conflict OfInterest** divergence between those two goals should be considered. For example, a divergence between the goals **Maxmize[NumberOfAmbulances]** and

Minimize[CostOfService] might be thereby identified in the London Ambulance system.

- **If** there are several possible instantiations of the *same* **SatisfactionGoal** goal among *multiple* agent instances, **then** the possibility of a **Competition** divergence should be considered. (We use the notion of competition here as it is defined in Section 3.3.8.) For example, a divergence between multiple instantiations of the goal ***Achieve*[MeetingScheduled]** might be thereby identified in a meeting scheduler system; between multiple instantiations of the goal:

  ***Achieve*[NearestFreeAmbulanceDispatched]**

  in the London Ambulance system; etc.
- **If** there is an ***Achieve*** goal with a target condition ***Q*** and an ***Avoid*** goal on a condition ***S*** with ***Q*** overlapping ***S*** (that is, ***Q*** implying ***S***), **then** the two goals are divergent if it is possible for their respective preconditions ***P*** and ***R*** to hold simultaneously (see the first pattern in Fig. 4).

More specific goal subcategories on the same object (like **Confidentiality** or **Integrity** subcategories of **SecurityGoals** [2]) will result in a more focussed search for corresponding divergences and boundary conditions.

One may also identify divergences by analogy with divergences in similar systems, using analogical reuse techniques in the spirit of [36].

5 Resolving Divergences

The divergences identified need to be resolved in some way or another depending on the likelihood of occurrence of boundary conditions and on the severity of the conflict consequences in such a case. Resolution should take place sooner or later depending on the potential for eliciting further information from divergent goals. The result of the resolution process will be a transformed goal structure and, in some cases, transformed object structures (see Section 3.4); the latter transformations may need to be propagated to the corresponding domain descriptions in Dom (see Section 5.2 below).

Various strategies can be followed to resolve divergences. Each of them corresponds to a specific class of *resolution operators* (see the process level in Fig. 1).

5.1 Assertion Transformation

Under this family, we group all operators which create, delete, or modify goal assertions.

5.1.1 Avoiding Boundary Conditions

Since boundary conditions lead to conflicts, a first strategy consists of preventing them from occurring. A new goal is therefore introduced which has the *Avoid* pattern:

$$P \Rightarrow \Box \neg B,$$

where *B* denotes a boundary condition to be inhibited. AND/OR refinement and divergence analysis may then be applied to this new goal in turn.

Coming back to our "French reviewer" example, we derived by regression in Section 4.1 the following boundary condition:

```
◊∃ r:Reviewer, p:Paper, a:Author, rep:Report
AuthorOf(a, p) ∧ Gets(a, rep, p, r)
∧ Reviews(r, p, rep)
∧ French(r) ∧ ¬ ∃ r'≠r: Expert(r', p) ∧ French(r')
```

Deriving an ***Avoid*** goal from this assertion will produce a new goal:

```
∀ r:Reviewer, p:Paper, a:Author, rep,rep':Report
Reviews(r, p, rep) ∧ AuthorOf(a, p)
∧ Gets(a, rep', p, r)
⇒ (rep' ≠ rep
  ∨[French(r) ⇒∃ r'≠ r: Expert(r', p) ∧ French(r')])
```

That is, the referee report should be modified/corrected or there should be at least one other French expert in the domain of the paper. The first alternative will require the integrity goal to be weakened as discussed below.

To take another real-world example, consider the two following goals in a library system, named **CopiesEventuallyAvailable** and **CopyKeptAsLongAsNeeded**, respectively:

```
(G1)   ∀m:Member, b:Book
       Requests(m, b) ∧ InLib(b)
       ⇒ ◊ (∃bc:BookCpy)[Copy(bc,b) ∧ Available(bc)]

(G2)   ∀m:Member, b:Book, bc:BookCopy
       Borrowing(m, bc) ∧ Copy(bc, b)
       ⇒ o [Needs(m, b) → Borrowing(m, bc)]
```

Given domain properties such as

```
[(∃m:Member)Borrowing(m,bc)] ⇒ ¬ Available(bc),

Requests(m, b) ⇒
    ¬ (∃bc)Borrowing(m,bc)𝒲(∃bc)Available(bc),
```

one can show that goals (**G1**) and (**G2**) are divergent as the following boundary condition is derived by regression:

```
◊∃m:Member, b:Book
Requests(m, b) ∧ InLib(b)
∧ ∀bc:BookCopy
  [Copy (bc, b) →
  (∃m':Member)(m'≠m ∧ Borrowing(m',bc)
                          ∧ □ o Needs(m',b))]
```

This boundary condition captures the possibility of a member requesting some book registered in the library whose copies are all borrowed by other members and in the sequel permanently needed by them.

Resolving the divergence by avoiding this boundary condition might be achieved by keeping some copies of every popular book always unborrowable to make them available for direct use in the library (this strategy is often implemented in university libraries).

It may turn out, after checking with domain experts, that the assertion $P \Rightarrow \Box \neg B$ introduced for divergence resolution is not a goal/requirement but a domain property that was missing from the domain theory *Dom*, making it possible to infer the boundary condition *B* by regression. In such cases, the domain theory will be updated instead of the goal structure. For example, it might be the case in some specific resource allocation system that no user can possibly need any resource for more than the two-week limit. The divergence detected in Section 4.1 would then be resolved by adding this property to the set of domain descriptions.

5.1.2 Goal Restoration

Boundary conditions cannot always be avoided, especially when they involve agents in the external environment over which the software has no control. An alternative strategy consists of introducing a new goal stating that if the boundary condition B occurs then the divergent goal assertions A_i become true again in some reasonably near future:

$$B \Rightarrow \Diamond_{\leq d} \bigwedge_{1\leq i\leq n} A_i$$

For the library example above, this strategy would correspond to temporarily breaking goal (**G2**) by forcing some member to return her copy even if she still needs it.

5.1.3 Conflict Anticipation

This strategy can be applied when some persistent condition P can be found such that, in some context C, one inevitably gets into a conflict after some time if the condition P has persisted over a too long period:

$$C \wedge \Box_{\leq d} P \Leftrightarrow \Diamond_{\leq d} \neg \bigwedge_{1\leq i\leq n} A_i$$

In such a case, one may introduce the following new goal to avoid the conflict by anticipation:

$$C \wedge P \Rightarrow \Diamond_{\leq d} \neg P$$

An example for the particular case where $n = 1$ would be a patient monitoring system where C is instantiated to "monitoring working," P to "some monitor value exceeding its threshold," and A to "patient alive."

5.1.4 Goal Weakening

This is probably the most common strategy for resolving divergence. The principle is to weaken the formulation of one or several among the divergent goals so as to make the divergence disappear. Let us illustrate the technique on an example first. Consider the goals ***Achieve*** **[RequestSatisfied]** and ***Avoid*** **[UnReliableResourceUsed]**; these goals were seen to be divergent in Section 4.2 by use of the first divergence pattern in Fig. 4. Their assertions were:

$$\text{Requesting}(u, r) \Rightarrow \Diamond\, \text{Using}(u, r)$$
$$\neg\, \text{Reliable}(r) \Rightarrow \Box\, \neg\, \text{Using}(u, r)$$

(Universal quantifiers are again left implicit.) The boundary condition was:

$$(\Diamond\, \exists\, u\text{: User}, r\text{: Resource})\ [\text{Requesting}(u, r) \wedge \neg\, \text{Reliable}(r)]$$

The divergence can be resolved by weakening the first goal to make it cover the boundary condition. This yields:

$$\text{Requesting}(u, r) \wedge \textit{Reliable}(r) \Rightarrow \Diamond\, \text{Using}(u, r)$$

It is easy to see that the boundary condition is now covered by this weakening; converting the above assertion into disjunctive form and using the tautology $P \vee Q \equiv P \vee \neg P \wedge Q$ we get:

$$\neg\,\text{Requesting}(u,r) \vee \text{Requesting}(u,r) \wedge \neg\,\text{Reliable}(r) \vee \Diamond\,\text{Using}(u,r)$$

The goal weakening needs to be compensated by the introduction of the new goal

$$\text{Requesting}(u, r) \Rightarrow \Diamond\, (\text{Requesting}(u, r) \wedge \text{Reliable}(r))$$

to make sure that the antecedent strengthening in the weakened goal becomes true at some point so that the initial target condition can be established in spite of the antecedent strengthening.

The principle is thus to make some goals more liberal so that they cover the boundary conditions. Once more liberal goals are obtained, their weakening needs to be propagated in the goal graph to replace the older, divergent versions.

The weakening procedure is more precisely described as follows:

1) *Weaken* some goal formulations to obtain more liberal versions that cover the boundary conditions. Syntactic generalization operators [20], [29] can be used here, such as adding a disjunct, removing a conjunct, or adding a conjunct in the antecedent of an implication.
2) For each weakened goal, *propagate* the predicate changes in the goal AND-tree in which this goal is involved, by replacing every occurrence of the old predicates by the new ones.

The decision on which goal to weaken will depend on the priority and utility of the divergent goals. In the "French reviewer" example, one will clearly weaken the goal ***Maintain*** **[ReviewIntegrity]** to allow making English corrections, removing signature information, and so on. The alternative weakening of the goal ***Maintain*** **[ReviewerAnonymity]** should not be considered.

Goal weakening often needs to be compensated by some strengthening elsewhere in the specification, for example, by addition of new goals (see the first example above). A new cycle of divergence analysis may then be required.

Resolution patterns can also be identified to guide the goal weakening process. Here are a few such patterns.

- Temporal relaxation:

weaken	$\Diamond_{\leq d} A$ to $\Diamond_{\leq c} A$	$(c > d)$
weaken	$\Box_{\leq d} A$ to $\Box_{\leq c} A$	$(c < d)$

- Resolve the *Achieve-Avoid* divergence pattern

$$P \Rightarrow \Diamond Q \;\; \textit{vs.} \;\; R \Rightarrow \Box \neg Q$$

by

– weakening the first assertion: $P \wedge \neg\, R \Rightarrow \Diamond\, Q$,

– keeping the second assertion: $R \Rightarrow \Box\, \neg Q$,

– strengthening the specification by adding the new goal: $P \Rightarrow \Diamond\, (P \wedge \neg\, R)$

This resolution pattern was used in the first example of goal weakening above. An example of temporal relaxation is the extension of the date range for scheduling meetings when there is a conflict, or the extension of a paper submission deadline when there are competing conferences.

An extreme case of goal weakening is *sacrificing*, that is, deleting one of the goals to eliminate the divergence. For example, there may be two divergent goals in an automatic teller machine, namely,

1) **Avoid[CardForgottenInATM]** and
2) Maintain **[UserComfort]**.

The first goal is operationalized by asking users to get their card back before taking cash, whereas the latter is operationalized by asking users to get their card back at the very end of the session when no further operation is asked. In some countries, the conflict is resolved by sacrificing Goal 1.

As another real-world example, consider the following three goals in a conference management system:

Maintain[**ConfidentialityOfSubmissions**],
Achieve[**AuthorsInformedOfReceipt**],
Avoid[**ProgramChairOverloaded**].

One can show that these three goals are divergent when taken together. The resolution that was recently chosen in some conference was to email all authors an acknowledgement in which all email addresses of submittors appeared, thus sacrificing the confidentiality goal.

5.1.5 Alternative Goal Refinement

In case of hard divergences that cannot be satisfactorily resolved through other strategies, one should investigate alternative refinements of goals at a higher level than the level at which the divergence occurred. The aim here is to obtain alternative subgoals which are no longer divergent, or whose divergence can be resolved by use of the other strategies. Alternative goal refinements may often result in different system proposals in which more or less functionality is automated and in which the interaction between the software and its environment may be quite different.

To illustrate this, consider the meeting scheduler system again and the divergence between the goals:

Achieve[**ParticipantsConstraintsRequested**],
Achieve[**RemindersSent**],
Achieve[**ParticipantsConstraintsProvided**]

on one hand, and the goal:

Minimize[**Participant Interaction**]

on the other hand. One way to resolve such a conflict is to go up in the goal refinement graph and reconsider alternative ways of refining the parent goal:

Achieve[**ParticipantsConstraintsKnown**].

An alternative design based on the scheduler accessing participant's electronic agendas might then be explored to resolve the divergence.

5.1.6 Divergence Resolution Heuristics

Heuristics for specific divergence categories may be used as a cheap alternative to formal techniques. We just give a few examples of heuristic rules here to illustrate the principle; a complete set of such rules is outside the scope of this paper.

- **If** there is a **Competition** divergence among multiple agent instances, **then** a resolution by introduction of an **Arbitrator** agent should be considered. Using this heuristic rule would lead to the introduction of an arbitrator to resolve the conflict between two instantiations of the goal **Achieve[NearestFreeAmbulanceDispatched]** in the London Ambulance system, in case of two simultaneous accidents near each other.
- **If** there is a **Competition** divergence among multiple agent instances, **then** a resolution by introduction of a reservation policy on the object of competition should be considered.
- **If** there is a divergence between a **ConfidentialityGoal** and an **InformationGoal** concerning the *same* object, **then** a resolution by restricted goals concerning specializations of the **Known** object should be considered.
- **If** there is a divergence between a **ConfidentialityGoal** and an **InformationGoal** concerning the *same* object, **then** a resolution by restricted goals involving specializations of the **Knowing** agents should be considered.

The last two rules involve transformations of object structures; we turn to this now.

5.2 Object Transformation

Under this family, we group all operators which create, delete, or modify object types. We give a few such operators here which have proved helpful in practice. A rich set of object restructuring operators is proposed in [48].

5.2.1 Object Refinement

The idea is to specialize an object type into disjoint subtypes and to restrict the divergent assertions to the corresponding subtypes. Consider, for example, the divergence between the security goal *Maintain*[**ReviewerAnonymity**] and the information goal *Achieve*[**EditorialBoardInformed**]; the boundary condition captures the situation where members of the editorial board are submitting papers. To prevent them from knowing the full submission status file (which contains the names of the reviewers for each paper), the conflict is resolved by specializing the **PaperStatusFile** object type into two subtypes: the (complete) **EIC-StatusFile** accessible by the editor-in-chief only, and the **EdBoard-StatusFile** which contains no reviewer names. The corresponding goals are then specialized accordingly. Note that the specialization may need to be propagated to domain descriptions in *Dom*; e.g., descriptions that involved the **PaperStatusFile** object type now need to refer to the specialized types introduced if they become specific to them.

Objects can also be "weakened" by extending the range of values for some of their attributes. For example, one can resolve conflicts between competing requests for meetings by extending the date range specified in the requests.

5.2.2 Agent Refinement

Divergent assertions that involve agents can sometimes be resolved by

1) specializing the corresponding agent types into disjoint subtypes, and
2) restricting the divergent assertions to the corresponding subtypes.

The principle is thus the same as above. Consider, for example, the divergence between the goals **Achieve[Patient Informed]** and **Maintain[MedicalSecret]** in a hospital management system. One way to resolve the divergence is to let parents of the patient access some specific items of the patient's file. (Note that this resolution integrates both object refinement and agent refinement.) Again, some domain descriptions may need to be transformed accordingly.

6 CONCLUSION

Requirements engineers live in a world where inconsistencies are the rule, not the exception. There are many different kinds of inconsistency; many of them originate from the elicitation of goals and requirements from multiple stakeholders, viewpoints, and independent documents. Tolerating such inconsistencies for a while is desirable as it increases the chances of getting more relevant information and, in the end, more complete and adequate requirements. However, detecting inconsistencies and resolving them at some stage or another of the requirements engineering process is a necessary condition for successful development of the software implementing the requirements.

This paper has proposed a formal framework for clarifying various types of inconsistency that can arise in the RE process; special attention has been given to one very general kind of inconsistency that surprisingly has received no attention so far in the literature. Divergence was seen to be a frequently occurring generalization of the usual notion of conflict, with some interesting particular cases. Various formal and heuristic techniques were then proposed for detecting and resolving divergences among goals/requirements in a systematic fashion. The notion of boundary condition was seen to play a prominent role in this context.

One key principle in the paper is to start thinking about divergences as early as possible in the requirements engineering process, that is, at the *goal* level. The earlier divergence analysis is started, the more freedom is left for resolving the divergences.

Domain knowledge was seen to play an important role in some of these techniques. In fact, this knowledge delimits the space of boundary conditions that can be found. However, as we pointed out, such knowledge can be elicited stepwise during divergence analysis.

When to apply such-and-such technique may depend on the domain, on the specific application in this domain, on the kind of divergence, on the likelihood of occurrence of boundary conditions, on the severity of the consequences of the resulting conflict, and on the cost of divergence resolution. Much work remains to be done to determine precisely when it is appropriate to apply each technique.

A question that may arise is the extent to which divergence among goal assertions could be detected using model checking technology. The strength of model checking techniques is that they could, at low cost, generate scenarios satisfying the boundary conditions we are looking for; such scenarios would be produced as traces that refute the divergent assertions conjoined with the domain theory. However, we currently envision two problems in applying existing model checking techniques directly for our purpose. On one hand, we want to conduct the analysis at the goal level for reasons explained throughout the paper; model checking requires the availability of an operational description of the target system (such as a finite state transition system [6], [37], [19]) or of relational specifications [26] that do not fit our formulation of goals in terms of temporal patterns of behavior. On the other hand, for the purpose of resolution, we need to obtain a formal specification of the boundary condition rather than an instance-level scenario satisfying it. A derivation calculus on more abstract specifications seems, therefore, more appropriate, even though instance scenarios generated by a tool like Nitpick [26] could provide concrete insights for detecting divergence among relational specifications.

We hope to have convinced the reader through the wide variety of examples given in this paper that the techniques proposed are general, systematic, and effective in identifying and resolving subtle divergences. Our plan is to integrate these techniques in the KAOS/GRAIL environment [11] so that large-scale experimentation on industrial projects from our tech transfer institute can take place. In fact, several divergences were recently detected in two real projects using our techniques. The first project was the engineering of requirements for a system to support the emergency service of a major Belgian hospital. Examples of divergences that were handled with our techniques included the divergence between the goals **`Avoid[ServiceOvercrowding]`** and **`Achieve[PatientAccompanied]`**; and the divergence between the goals **`Achieve[PromptAction]`** and **`Achieve [MaximalInformationAcquired]`**. The second project concerned the development of a complex software system to support goods delivery to retailers. An interesting case detected using our technique was the divergence between the retailer's goal of one-hour delivery for every order and the company's goal of one delivery per day for every retailer (the latter obviously refined a cost reduction goal). The boundary condition found captured the situation of a retailer making more than one order on the same day. The resolution that was chosen among those identified was to reward retailers that group all their orders for the same day within one single batch to be submitted once a day.

Acknowledgments

The work reported herein was partially supported by the "Communauté Française de Belgique" (FRISCO project, Actions de Recherche Concertées Nr. 95/00-187—Direction générale de la Recherche). Partial support was also provided while the first author was on sabbatical at SRI International by the Advanced Research Projects Agency under Air Force Research Laboratory contract number F30602-97-C-0040. Many thanks to the inconsistency management groups at Imperial College, City University, and Politecnico di Milano for sharing (sometimes divergent) views on inconsistency management. Christophe Ponsard made helpful comments on an earlier version of this paper. We greatly appreciate the detailed comments and constructive suggestions made by the reviewers.

References

[1] M. Abadi, "Temporal-Logic Theorem Proving," PhD thesis, Stanford Univ., Mar. 1987.
[2] E.J. Amoroso, *Fundamentals of Computer Security*. Prentice Hall, 1994.
[3] A.I. Anton, W.M. McCracken, and C. Potts, "Goal Decomposition and Scenario Analysis in Business Process Reengineering," *Proc. Sixth Int'l Conf. Advanced Information Systems Eng.*, pp. 94-104, Lecture Notes in Computer Science 811, Springer-Verlag, 1994.
[4] B.W. Boehm, P. Bose, E. Horowitz, and M.J. Lee, "Software Requirements Negotiation and Renegotiation Aids: A Theory-W Based Spiral Approach," *Proc. ICSE-17—17th Int'l Conf. Software Eng.*, pp. 243-253, 1995.
[5] *Readings in Knowledge Representation*, R.J. Brachman and H.J. Levesque, eds. Morgan Kaufmann, 1985.
[6] E.M. Clarke and E.A. Emerson, "Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications," *ACM Trans. Programming Language Systems*, vol. 8, no. 2, pp. 244-263, 1986.
[7] G. Cugola, E. Di Nitto, A. Fuggetta and C. Ghezzi, "A Framework for Formalizing Inconsistencies and Deviations in Human-

Centered Systems," *ACM Trans. Software Eng. and Methodology*, vol. 5, no. 3, pp. 191-230, July 1996.

[8] A. Dardenne, S. Fickas and A. van Lamsweerde, "Goal-Directed Concept Acquisition in Requirements Elicitation," *Proc. IWSSD-6—Sixth Int'l Workshop Software Specification and Design*, pp. 14-21, Como, Italy, 1991.

[9] A. Dardenne, A van Lamsweerde, and S. Fickas, "Goal-Directed Requirements Acquisition," *Science of Computer Programming*, vol. 20, pp. 3-50, 1993.

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

[11] R. Darimont, E. Delor, P. Massonet, and A. van Lamsweerde, "GRAIL/KAOS: An Environment for Goal-Driven Requirements Engineering," *Proc. ICSE'98—20th Int'l Conf. Software Eng.*, vol. 2, pp. 58-62, Kyoto, Japan, Apr. 1998.

[12] E. Dubois, N. Levy, and J. Souquières, "Formalising Restructuring Operators in a Specification Process," *Proc. ESEC-87—First European Software Eng. Conf.*, pp. 161-171, Sept. 1987.

[13] S. Easterbrook, "Domain Modelling with Hierarchies of Alternative Viewpoints," *Proc. RE'93—First Int'l Symp. Requirements Eng.*, San Diego, Calif., 1993.

[14] M. Feather, "Language Support for the Specification and Development of Composite Systems," *ACM Trans. Programming Languages and Systems*, vol. 9, no. 2, pp. 198-234, Apr. 1987.

[15] M. Feather, S. Fickas, A. van Lamsweerde, and C. Ponsard, "Reconciling System Requirements and Runtime Behavior," *Proc. IWSSD'98—Ninth Int'l Workshop Software Specification and Design*, pp. 50-59, Ise-Shima, Japan, Apr. 1998.

[16] S. Fickas and M. Feather, "Requirements Monitoring in Dynamic Environments," *Proc. RE'95—Second Int'l Symp. Requirements Eng.*, York, U.K., 1995.

[17] A. Finkelstein and H. Fuks, "Multi-Party Specification," *Proc. Fifth Int'l Workshop Software Specification and Design*, Pittsburgh, Pa., May 1989.

[18] A. Finkelstein, "The London Ambulance System Case Study," *Proc. IWSSD8—Eighth Int'l Workshop Software Specification and Design, ACM Software Engineering Notes*, Sept. 1996.

[19] R. Gerth, D. Peled, M. Vardi, and P. Wolper, "Simple On-the-Fly Automatic Verification of Linear Temporal Logic," *Proc. IFIP WG6.1 Symp. Protocol Specification, Testing and Verification*, North Holland, 1995.

[20] D. Gries, *The Science of Programming*. Springer-Verlag, 1981.

[21] M.P. Heimdahl and N.G. Leveson, "Completeness and Consistency Analysis of State-Based Requirements," *Proc. ICSE'95—17th Int'l Conf. Software Eng.*, pp. 3-14, Seattle, Wash., 1995.

[22] C. Heitmeyer, R. Jeffords, and B. Labaw, "Automated Consistency Checking of Requirements Specifications," *ACM Trans. Software Eng. and Methodology*, vol. 5, no. 3, pp. 231-261, July 1996.

[23] A. Hunter and B. Nuseibeh, "Analyzing Inconsistent Specifications," *Proc. RE'97—Third Int'l Symp. Requirements Eng.*, pp. 78-86, Annapolis, Md., 1997.

[24] M. Jackson and P. Zave, "Domain Descriptions," *Proc. RE'93—First Int'l IEEE Symp. Requirements Eng.*, pp. 56-64, Jan. 1993.

[25] D. Jackson, "Structuring Z Specifications with Views," *ACM Trans. Software Eng. and Methodology*, vol. 4, no. 4, pp. 365-389, Oct. 1995.

[26] D. Jackson, "Elements of Style: Analyzing a Software Design Feature with a Counterexample Detector," *Proc. ACM Int'l Symp. Software Testing and Analysis*, pp. 239-249, San Diego, Calif., 1996.

[27] A.J. Jones and M. Sergot, "On the Characterization of Law and Computer Systems: the Normative System Perspective," *Deontic Logic in Computer Science—Normative System Specification*, J.C. Meyer and R.J. Wieringa, eds. Wiley, 1993.

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

[29] A. van Lamsweerde, "Learning Machine Learning," *Introducing a Logic Based Approach to Artificial Intelligence*, A. Thayse, ed., vol. 3, pp. 263-356. Wiley, 1991.

[30] A. van Lamsweerde, R. Darimont, and P. Massonet, "Goal-Directed Elaboration of Requirements for a Meeting Scheduler: Problems and Lessons Learned," *Proc. RE'95—Second Int'l Symp. Requirements En.*, York, U.K., 1995.

[31] A. van Lamsweerde, "Divergent Views in Goal-Driven Requirements Engineering," *Proc. Viewpoints'96—ACM SIGSOFT Workshop Viewpoints in Software Development*, Oct. 1996.

[32] A. van Lamsweerde and E. Letier, "Integrating Obstacles in Goal-Driven Requirements Engineering," *Proc. ICSE'98—20th Int'l Conf. Software Eng.*, Kyoto, Japan, Apr. 1998.

[33] A. van Lamsweerde and L. Willemet, "Inferring Declarative Requirements Specifications from Operational Scenarios," *IEEE Trans. Software Eng.*, vol. 24, no. 12, Dec. 1998. to appear

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

[35] Z. Manna and the STep Group, "STeP: Deductive-Algorithmic Verification of Reactive and Real-Time Systems," *Proc. CAV'96—Eighth Int'l Conf. Computer-Aided Verification*, pp. 415-418, July 1996.

[36] P. Massonet and A. van Lamsweerde, "Analogical Reuse of Requirements Frameworks," *Proc. RE-97—Third Int'l Symp. Requirements Eng.*, pp. 26-37, Annapolis, Md., 1997.

[37] K.L. McMillan, *Symbolic Model Checking: An Approach to the State Explosion Problem*. Kluwer, 1993.

[38] B. Meyer, "On Formalism in Specifications," *IEEE Software*, vol. 2, no. 1, pp. 6-26, Jan. 1985.

[39] J. Mylopoulos, L. Chung, and B. Nixon, "Representing and Using Nonfunctional Requirements: A Process-Oriented Approach," *IEEE Trans. Software. Eng.*, vol. 18, no. 6, pp. 483-497, June 1992.

[40] J. Mylopoulos and R. Motschnig-Pitrik, "Partitioning Information Bases with Contexts," *Proc. Third Int'l Conf. Cooperative Information Systems*, Vienna, May 1995.

[41] C. Niskier, T. Maibaum, and D. Schwabe, "A Pluralistic Knowledge-Based Approach to Software Specification," *Proc. ESEC-89—Second European Software Eng. Conf.*, pp. 411-423, Sept. 1989.

[42] B. Nuseibeh, J. Kramer, and A. Finkelstein, "A Framework for Expressing the Relationships Between Multiple Views in Requirements Specifications," *IEEE Trans. Software Eng.*, vol. 20, no. 10, pp. 760-773, Oct. 1994.

[43] B. Potter, J. Sinclair, and D. Till, *An Introduction to Formal Specification and Z*. Prentice Hall, 1991.

[44] C. Potts, K. Takahashi, and A.I. Anton, "Inquiry-Based Requirements Analysis," *IEEE Software*, vol. 11, no. 2, pp. 21-32, Mar. 1994.

[45] C. Potts, "Using Schematic Scenarios to Understand User Needs," *Proc. DIS'95—ACM Symp. Designing interactive Systems: Processes, Practices, and Techniques*, Univ. of Michigan, Aug. 1995.

[46] W.N. Robinson,, "Integrating Multiple Specifications Using Domain Goals," *Proc. IWSSD-5—Fifth Int'l Workshop Software Specification and Design*, pp. 219-225, 1989.

[47] W.N. Robinson, "Negotiation Behavior During Requirement Specification," *Proce. ICSE12—12th Int'l Conf. Software Eng.*, pp. 268-276, Mar. 1990.

[48] W.N. Robinson and S. Volkov, "A Meta-Model for Restructuring Stakeholder Requirements," *Proc. ICSE19—19th Int'l Conf. Software Eng.*, pp. 140-149, Boston, May 1997.

[49] D.T. Ross and K.E. Schoman, "Structured Analysis for Requirements Definition," *IEEE Trans. Software Eng.*, vol. 3, no. 1, pp. 6-15, 1977.

[50] K.S. Rubin and A. Goldberg, "Object Behavior Analysis," *Comm. ACM*, vol. 35, no. 9, pp. 48-62, Sept. 1992.

[51] K. Ryan and S. Greenspan, "Requirements Engineering Group Report," *Proc. IWSSD8—Eighth Int'l Workshop Software Specification and Design, ACM Software Eng. Notes*, pp. 22-25, Sept. 1996.

[52] I. Sommerville and P. Sawyer, *Requirements Engineering: A Good Practice Guide*. Wiley, 1997.

[53] G. Spanoudakis and A. Finkelstein, "Interference in Requirements Engineering: the Level of Ontological Overlap," Report TR-1997/01, City Univ., London, U.K., 1997.

[54] R. Waldinger, "Achieving Several Goals Simultaneously," *Machine Intelligence*, vol. 8, E. Elcock and D. Michie, eds. Ellis Horwood, 1977.

[55] K. Yue, "What Does It Mean to Say that a Specification is Complete?" *Proc. IWSSD-4, Fourth Int'l Workshop Software Specification and Design*, 1987.

[56] P. Zave and M. Jackson, "Conjunction as Composition," *ACM Trans. Software Eng. and Methodology*, vol. 2, no. 4, pp. 379-411, 1993.

[57] P. Zave and M. Jackson, "Four Dark Corners of Requirements Engineering." *ACM Trans. Software Eng. and Methodology*, vol. 6, no. 1, pp. 1-30, 1996.

Axel van Lamsweerde received the MS degree in mathematics from the Université Catholique de Louvain, Belgium, and the PhD degree in computing science from the University of Brussels. He is a full professor of computing science at the Université Catholique de Louvain, Belgium. From 1970 to 1980, he was a research associate with the Philips Research Laboratory in Brussels, where he worked on proof methods for parallel programs and knowledge-based approaches to automatic programming. He was then a professor of software engineering at the Universities of Namur and Brussels until he joined UCL in 1990. He is co-founder of the CEDITI technology transfer institute, partially funded by the European Union. He has also been a visitor at the University of Oregon and the Computer Science Laboratory of SRI International, Menlo Park.

Dr. van Lamsweerde's professional interests are in lightweight formal methods and tools for assisting software engineers in knowledge-intensive tasks. His current focus is on constructive, technical approaches to requirements engineering and, more generally, to formal reasoning about software engineering products and processes. His recent papers can be found at http://www.info.ucl.ac.be/people/avl.html.

Dr. van Lamsweerde was program chair of the Third European Software Engineering Conference (ESEC '91), program co-chair of the Seventh IEEE Workshop on Software Specification and Design (IWSSD-7), and program co-chair of the ACM-IEEE 16th International Conference on Software Engineering (ICSE-16). He is a member of the editorial boards of the *Automated Software Engineering Journal* and the *Requirements Engineering Journal*. Since 1995, he has been editor-in-chief of the *ACM Transactions on Software Engineering and Methodology (TOSEM)*. He is a member of the IEEE, ACM, and AAAI.

Robert Darimont received the MS and PhD degrees in applied sciences (computing science orientation) from the Université Catholique de Louvain, Belgium. He is the manager of the Software Engineering Group at CEDITI, the IT transfer center of the University of Louvain. He works on the development of GRAIL, the tool supporting the KAOS methodology for goal-oriented requirements engineering; leads industrial projects using KAOS; and helps companies to adopt up to date and precompetitive software engineering technologies. Dr. Darimont is a member of the IEEE and the ACM and the information director of the *ACM Transactions on Software Engineering and Methodology (TOSEM)*.

Emmanuel Letier received the degree of engineer in applied mathematics in 1995 from the Université Catholique de Louvain, Belgium. He is currently pursuing PhD research on formal reasoning about agents during requirements elaboration at the Département d'Ingénierie Informatique of this university.